

C++ Programmierung

Benjamin Buch

[Wikibooks.org](https://de.wikibooks.org/)

1. Juli 2012

On the 28th of April 2012 the contents of the English as well as German Wikibooks and Wikipedia projects were licensed under Creative Commons Attribution-ShareAlike 3.0 Unported license. An URI to this license is given in the list of figures on page 267. If this document is a derived work from the contents of one of these projects and the content was still licensed by the project under this license at the time of derivation this document has to be licensed under the same, a similar or a compatible license, as stated in section 4b of the license. The list of contributors is included in chapter Contributors on page 263. The licenses GPL, LGPL and GFDL are included in chapter Licenses on page 271, since this book and/or parts of it may or may not be licensed under one or more of these licenses, and thus require inclusion of these licenses. The licenses of the figures are given in the list of figures on page 267. This PDF was generated by the \LaTeX typesetting software. The \LaTeX source code is included as an attachment (`source.7z.txt`) in this PDF file. To extract the source from the PDF file, we recommend the use of <http://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/utility> or clicking the paper clip attachment symbol on the lower left of your PDF Viewer, selecting `Save Attachment`. After extracting it from the PDF file you have to rename it to `source.7z`. To uncompress the resulting archive we recommend the use of <http://www.7-zip.org/>. The \LaTeX source itself was generated by a program written by Dirk Hünninger, which is freely available under an open source license from http://de.wikibooks.org/wiki/Benutzer:Dirk_Huenniger/wb2pdf. This distribution also contains a configured version of the `pdflatex` compiler with all necessary packages and fonts needed to compile the \LaTeX source included in this PDF file.

Inhaltsverzeichnis

1	Wie lese ich dieses Buch?	3
2	Es war einmal.	5
2.1	Der Name „C++“	6
2.2	Weiterentwicklung der Programmiersprache C++	6
2.3	Berücksichtigung aktueller technischer Gegebenheiten	6
3	Compiler	9
3.1	Was passiert beim Übersetzen	9
3.2	Hilfsmittel	10
3.3	Die GCC	10
3.4	Für Fortgeschrittenere	12
3.5	Makefiles	14
3.6	Boost Build	15
4	GUIs und C++	17
4.1	Was ist ein GUI?	17
4.2	C++ und GUIs	17
4.3	Einsatz von GUI	17
5	Hallo, du schöne Welt!	19
6	Einfache Ein- und Ausgabe	21
6.1	Einfache Ausgabe	21
6.2	Einfache Eingabe	22
7	Kommentare	25
8	Rechnen (lassen)	27
8.1	Einfaches Rechnen	27
8.2	Die großen Vier	28
8.3	Zusammengesetzte Operatoren	29
8.4	Inkrement und Dekrement	30
9	Variablen, Konstanten und ihre Datentypen	33
9.1	Datentypen	33
9.2	Variablen	39
9.3	Konstanten	41
9.4	Literale und ihre Datentypen	42
10	Rechnen mit unterschiedlichen Datentypen	45
10.1	Ganzzahlen	45
10.2	Gleitkommarechnen	47
10.3	Casting	47
10.4	Ganzzahlen und Gleitkommazahlen	49
10.5	Rechnen mit Zeichen	49
11	Verzweigungen	51
11.1	Falls	51
11.2	Andernfalls	52

11.3	Vergleichsoperatoren	54
11.4	Logische Operatoren	55
11.5	Bedingter Ausdruck	58
12	Schleifen	61
12.1	Die <code>while</code> -Schleife	63
12.2	Die <code>do-while</code> -Schleife	65
12.3	Die <code>for</code> -Schleife	66
12.4	Die <code>break</code> -Anweisung	68
12.5	Die <code>continue</code> -Anweisung	68
12.6	Kapitelanhang	69
13	Auswahl	73
13.1	<code>switch</code> und <code>break</code>	74
13.2	Nur Ganzzahlen	76
13.3	Zeichen und mehrere <code>case</code> -Zweige	76
14	Ein Taschenrechner wird geboren	79
14.1	Die Eingabe	79
14.2	Die 4 Grundrechenarten	79
14.3	Das ganze Programm	80
15	Zusammenfassung	81
15.1	Ein- und Ausgabe	81
15.2	Kommentare	82
15.3	Rechnen	82
15.4	Variablen	82
15.5	Kontrollstrukturen	84
16	Prozeduren und Funktionen	87
16.1	Parameter und Rückgabewert	87
16.2	Übergabe der Argumente	90
16.3	Default-Parameter	93
16.4	Funktionen überladen	94
16.5	Funktionen mit beliebig vielen Argumenten	95
16.6	Inline-Funktionen	97
17	Lebensdauer und Sichtbarkeit von Objekten	99
17.1	Statische Variablen	99
17.2	Alter Text	103
17.3	Sichtbarkeit	105
18	Rekursion	107
18.1	Fakultät	107
18.2	Fibonacci-Zahlen	108
18.3	Merge sort	109
19	Zeiger	111
19.1	Grundlagen zu Zeigern	111
19.2	Zeiger und <code>const</code>	112
19.3	Zeiger und Funktionen	113
19.4	Zeigerarithmetik	114
19.5	Negativbeispiele	116
19.6	<code>void</code> -Zeiger (anonyme Zeiger)	117
19.7	Funktionszeiger	118

20	Referenzen	121
20.1	Grundlagen zu Referenzen	121
20.2	Anwendung von Referenzen	122
21	Felder	127
21.1	Zeiger und Arrays	128
21.2	Mehrere Dimensionen	129
21.3	Arrays und Funktionen	131
21.4	Lesen komplexer Datentypen	133
22	Zeichenketten	135
22.1	Einleitung	135
22.2	Wie entsteht ein <code>string</code> -Objekt?	135
22.3	<code>string</code> und andere Datentypen	137
22.4	Zuweisen und Verketteten	138
22.5	Nützliche Methoden	138
22.6	Zeichenzugriff	139
22.7	Manipulation	140
22.8	Vergleiche	142
22.9	Zahl zu <code>string</code> und umgekehrt	144
22.10	C-Strings	144
23	Vorarbeiter des Compilers	144
23.1	<code>#include</code>	145
23.2	<code>#define</code> und <code>#undef</code>	146
23.3	<code>#</code>	147
23.4	<code>##</code>	147
23.5	<code>#if</code> , <code>#ifdef</code> , <code>#ifndef</code> , <code>#else</code> , <code>#elif</code> und <code>#endif</code>	148
23.6	<code>#error</code> und <code>#warning</code>	149
23.7	<code>#line</code>	149
23.8	<code>#pragma</code>	149
23.9	Vordefinierte Präprozessor-Variablen	150
24	Headerdateien	151
24.1	Was ist eine Headerdatei?	151
24.2	Namenskonventionen	152
24.3	Schutz vor Mehrfacheinbindung	152
24.4	Inline-Funktionen	153
25	Das Klassenkonzept	155
25.1	Ein eigener Datentyp	156
26	Erstellen und Zerstören	161
26.1	Konstruktor	161
26.2	Destruktor	167
26.3	Beispiel mit Ausgabe	167
26.4	Explizite Konstruktoren	168
27	Privat und öffentlich	171
27.1	Strukturen	171
28	Operatoren überladen	173
28.1	Definition der überladenen Operatoren	175
28.2	Spaß mit „falschen“ Überladungen	190
28.3	Präfix und Postfix	191

29	Klassen als Datenelemente einer Klasse	195
29.1	Verweise untereinander	195
30	Rechnen mit Brüchen	197
30.1	Was diese Klasse bieten soll	197
30.2	Was diese Klasse nicht kann	197
30.3	Ein erster Blick	197
31	Die Methoden	199
31.1	Zugriff	199
31.2	ggT()	200
31.3	kgV()	201
31.4	kuerzen()	201
32	Die Rechenoperationen	203
32.1	Addition	203
32.2	Subtraktion	203
32.3	Multiplikation	203
32.4	Division	204
32.5	Kombination	204
32.6	Abschluss	204
33	Umwandlung aus anderen Datentypen	207
33.1	Gleitkommazahl wird Bruch	208
33.2	(K)ein Kopierkonstruktor	209
34	Ein- und Ausgabe	211
34.1	Ausgabe	211
34.2	Eingabe	211
35	Umwandlung in andere Datentypen	213
36	Der Taschenrechner geht zur Schule	215
37	Zusammenfassung	217
38	Methoden	221
38.1	Konstante Methoden	221
38.2	Sinn und Zweck konstanter Objekte	222
38.3	Zugriffsmethoden	222
38.4	Überladen	223
38.5	Code-Verdopplung vermeiden	224
38.6	Call by reference	225
38.7	Wir empfehlen inline	227
39	Vererbung	229
39.1	Einleitung	229
39.2	Die Ausgangslage	229
39.3	Gemeinsam und doch getrennt	230
39.4	Erläuterung des Beispiels	230
39.5	protected	231
40	Methoden (nicht) überschreiben	233
40.1	Einleitung	233
40.2	Zwei gleiche Namen, ein Aufruf	233
40.3	Die Methode des Vaters aufrufen	233
41	Private und geschützte Vererbung	235
41.1	Einleitung	235
41.2	Private Vererbung	235

41.3	Geschützte Vererbung	235
41.4	Wann wird was benutzt?	235
42	Funktionstemplates	237
42.1	Spezialisierung	239
42.2	Zwischen Spezialisierung und Überladung	240
42.3	Überladen von Template-Funktionen	241
42.4	Templates mit mehreren Parametern	244
42.5	Nichttypargumente	246
42.6	<code>inline</code>	247
43	Container	249
43.1	Containerklassen	249
44	Interne Zahlendarstellung	253
44.1	Gleitkommazahlen	260
45	Autoren	263
	Abbildungsverzeichnis	267
46	Licenses	271
46.1	GNU GENERAL PUBLIC LICENSE	271
46.2	GNU Free Documentation License	272
46.3	GNU Lesser General Public License	272

1 Wie lese ich dieses Buch?

Egal, ob Sie schon C++ können, eine andere Programmiersprache beherrschen oder kompletter Programmieranfänger sind. Wenn Sie in C++ programmieren wollen, werden Sie mit diesem Buch etwas anfangen können. Es steht Ihnen frei, das Buch ganz normal von Anfang an zu lesen oder sich die Kapitel herauszupicken, die Sie interessieren. Allerdings sollten Sie beim Herauspickern darauf achten, dass Sie den Stoff der vorherigen Kapitel beherrschen, andernfalls werden Sie wahrscheinlich Schwierigkeiten haben, das Kapitel Ihres Interesses zu verstehen.

Dieses Buch gliedert sich in verschiedene Abschnitte, von denen jeder einen eigenen Schwierigkeitsgrad und entsprechend eine eigene Zielgruppe hat. Die meisten Abschnitte vermitteln Wissen über C++ oder sollen die praktische Anwendung zeigen, um das bisherige Wissen zu festigen. Andere weisen auf Besonderheiten und „Stolpersteine“ hin oder vermitteln Hintergrundwissen.

- Jeder Abschnitt enthält eine Anzahl von Kapiteln, welche einen komplexeren Zusammenhang erklären. Die Abschnitte sind in sich abgeschlossen, setzen aber voraus, dass Sie den Stoff der vorherigen Abschnitte verstanden haben.
- Am Ende jedes Kapitels stehen meistens ein paar Fragen und/oder Aufgaben, die Ihnen helfen sollen zu überprüfen, ob Sie verstanden haben, was im entsprechenden Kapitel stand. Die Antworten auf die Fragen lassen sich einfach ausklappen, für die Aufgaben gibt es je einen Verweis auf eine Seite mit einer Musterlösung. Diese Musterlösung ist natürlich nicht die einzige Variante, um die Aufgabe korrekt zu erfüllen, sie dient lediglich als Beispiel, wie Sie es machen könnten. Zu beachten ist allerdings, dass das im Kapitel erworbene Wissen genutzt werden sollte, um die Aufgabe zu erfüllen.
- Jeder Abschnitt schließt mit einer Zusammenfassung ab, die das vermittelte Wissen kurz und bündig auf den Punkt bringt. Wer schnell etwas nachschlagen will, kann sich in der Regel an die Zusammenfassungen halten. Auf Abweichungen von dieser Regel wird explizit hingewiesen.

„Für Programmieranfänger“ ist der nächste Abschnitt. Wenn Sie schon in einer anderen Programmiersprache Programme geschrieben haben, können Sie diesen Abschnitt getrost überspringen. Es geht nicht speziell um C++, sondern eher darum, zu begreifen, was überhaupt eine Programmiersprache ist und wie man dem Rechner sagen kann, was er zu tun hat.

Wenn Sie eine andere Programmiersprache schon sehr sicher beherrschen, reicht es wahrscheinlich, wenn Sie von den nächsten Abschnitten nur die Zusammenfassungen lesen, aber es kann Ihnen in keinem Fall schaden, auch die kompletten Abschnitte zu lesen. Die Zusammenfassungen sollen Ihnen zwar einen groben Überblick geben, was in diesem Abschnitt vermittelt wurde, aber es ist nun einmal nicht möglich, zur gleichen Zeit kurz und vollständig zu sein.

Sofern Sie C++ schon sicher beherrschen, können Sie sich einfach ein Kapitel ausgucken, es lesen und wenn nötig, in einem früheren Kapitel nachschlagen. Umsteigern ist dieses Verfahren hingegen nicht zu empfehlen, weil C++ eben doch etwas anderes ist als die meisten anderen Programmiersprachen.

In jedem Fall hoffen wir, dass dieses Buch Ihnen hilft, Programmieren zu lernen und Ihre Fähigkeiten zu verbessern, aber vergessen Sie niemals, dass ein Buch nur eine Stütze sein kann. *Programmieren lernt man durchs Programmieren!*

2 Es war einmal...

C++ wurde von Bjarne Stroustrup ab 1979 in den AT&T Bell Laboratories entwickelt. Ausgangspunkt waren Untersuchungen des UNIX-Betriebssystemkerns in Bezug auf verteiltes Rechnen.

Auf die Idee für eine neue Programmiersprache war Stroustrup schon durch Erfahrungen mit der Programmiersprache Simula im Rahmen seiner Doktorarbeit an der Cambridge University gekommen. Simula erschien zwar geeignet für den Einsatz in großen Software-Projekten, die Struktur der Sprache erschwerte aber die für viele praktische Anwendungen erforderliche Erzeugung hocheffizienter Programme. Demgegenüber ließen sich effiziente Programme zwar mit der Sprache BCPL schreiben, für große Projekte war BCPL aber wiederum ungeeignet.

Mit den Erfahrungen aus seiner Doktorarbeit erweiterte Stroustrup nun die Programmiersprache C um ein Klassenkonzept, für das die Sprache Simula-67 das primäre Vorbild war. Die Wahl fiel auf die Programmiersprache C, eine Mehrzwecksprache, die schnellen Code produzierte und einfach auf andere Plattformen zu portieren war. Als dem Betriebssystem UNIX beiliegende Sprache hatte C außerdem eine nicht unerhebliche Verbreitung. Zunächst fügte er der Sprache Klassen (mit Datenkapselung) hinzu, dann abgeleitete Klassen, ein strengeres Typsystem, Inline-Funktionen und Standard-Argumente.

Während Stroustrup „C with Classes“ („C mit Klassen“) entwickelte (woraus später C++ wurde), schrieb er auch cfront, einen Compiler, der aus C with Classes zunächst C-Code als Zwischenresultat erzeugte. Die erste kommerzielle Version von cfront erschien im Oktober 1985.

1983 wurde C with Classes in C++ umbenannt. Erweiterungen darin waren: virtuelle Funktionen, Überladen von Funktionsnamen und Operatoren, Referenzen, Konstanten, änderbare Freispeicherverwaltung und eine verbesserte Typüberprüfung. Die Möglichkeit von Kommentaren, die an das Zeilenende gebunden sind, wurde wieder aus BCPL übernommen (//).

1985 erschien die erste Version von C++, die eine wichtige Referenzversion darstellte, da die Sprache damals noch nicht standardisiert war. 1989 erschien die Version 2.0 von C++. Neu darin waren Mehrfachvererbung, abstrakte Klassen, statische Elementfunktionen, konstante Elementfunktionen und die Erweiterung des Schutzmodells um `protected`. 1990 erschien das Buch „The Annotated C++ Reference Manual“, das als Grundlage für den darauffolgenden Standardisierungsprozess diente.

Relativ spät wurden der Sprache Templates, Ausnahmen, Namensräume, neuartige Typumwandlungen und boolesche Typen hinzugefügt.

Im Zuge der Weiterentwicklung der Sprache C++ entstand auch eine gegenüber C erweiterte Standardbibliothek. Erste Ergänzung war die Stream-I/O-Bibliothek, die Ersatz für traditionelle C-Funktionen wie zum Beispiel `printf()` und `scanf()` bietet. Eine der wesentlichen Erweiterungen der Standardbibliothek kam später durch die Integration großer Teile der bei Hewlett-Packard entwickelten Standard Template Library (STL) hinzu.

Nach jahrelanger Arbeit wurde schließlich 1998 die endgültige Fassung der Sprache C++ (ISO/IEC 14882:1998) genormt. 2003 wurde ISO/IEC 14882:2003 verabschiedet, eine Nachbesserung der Norm von 1998. Die nächste größere Überarbeitung der Sprache C++ erscheint frühestens 2010.

Die vorhandenen Performance-Probleme der Sprache C++, die Rechenzeit und Speicherplatz betreffen, werden auf hohem Niveau zusammen mit Lösungen im Technical Report ISO/IEC TR 18015:2006 diskutiert. Das Dokument ist zum Download von ISO freigegeben.

2.1 Der Name „C++“

Der Name ist eine Wortschöpfung von Rick Mascitti und wurde zum ersten Mal im Dezember 1983 benutzt. Der Name kommt von der Verbindung der Vorgängersprache C und dem Inkrement-Operator „++“, der den Wert einer Variable um eins erhöht.

2.2 Weiterentwicklung der Programmiersprache C++

Um mit den aktuellen Entwicklungen der sich schnell verändernden Computer-Technik Schritt zu halten, aber auch zur Ausbesserung bekannter Schwächen, erarbeitet das C++-Standardisierungskomitee derzeit die nächste größere Revision von C++, die inoffiziell mit C++0x abgekürzt wird, worin die Ziffernfolge eine grobe Einschätzung des möglichen Erscheinungstermins andeuten soll. Im November 2006 wurde der Zieltermin für die Fertigstellung auf das Jahr 2009 festgelegt und die Abkürzung C++09 dafür eingeführt.

Die vorrangigen Ziele für die Weiterentwicklung von C++ sind Verbesserungen im Hinblick auf die Systemprogrammierung sowie zur Erstellung von Programmbibliotheken. Außerdem soll die Erlernbarkeit der Sprache für Anfänger verbessert werden.

2.3 Berücksichtigung aktueller technischer Gegebenheiten

C++ deckt einige typische Problemfelder der Programmierung noch nicht ausreichend ab, zum Beispiel die Unterstützung von Nebenläufigkeit (Threads), deren Integration in C++, insbesondere für die Verwendung in Mehrprozessorumgebungen, die eine Überarbeitung der Sprache unumgänglich macht. Die hierfür nötige Überarbeitung des C++-Speichermodells soll Garantien der Sprache für den nebenläufigen Betrieb festlegen, um Mehrdeutigkeiten in der Abarbeitungsreihenfolge sowohl aufzulösen, als auch in bestimmten Fällen aufrechtzuerhalten und dadurch Spielraum für Optimierungen zu schaffen.

2.3.1 Verbesserungen am Sprachkern

Zu den weiter reichenden Spracherweiterungen gehören die "Konzepte" (engl. concepts) zur Verbesserung der Handhabung von Templates, Typinferenz zur Ableitung von Ergebnistypen aus Ausdrücken und die R-Wert-Referenzen, mit deren Hilfe sich als Ergänzung zu dem bereits vorhandenen Kopieren von Objekten dann auch ein Verschieben realisieren lässt.

2.3.2 Erweiterung der Programmbibliothek

Im April 2006 gab das C++-Standardisierungskomitee den so genannten ersten technischen Report (TR1) heraus, eine nichtnormative Ergänzung zur aktuell gültigen, 1998 definierten Bibliothek, mit der Erweiterungsvorschläge vor einer möglichen Übernahme in die C++-Standardbibliothek auf ihre Praxistauglichkeit hin untersucht werden sollen. Viele Compiler-Hersteller liefern den TR1 mit ihren Produkten aus.

Enthalten sind im TR1 u.a. reguläre Ausdrücke, verschiedene intelligente Zeiger, ungeordnete assoziative Container, eine Zufallszahlenbibliothek, Hilfsmittel für die C++-Metaprogrammierung, Tupel sowie numerische und mathematische Bibliotheken. Die meisten dieser Erweiterungen stammen aus der Boost-Bibliothek, woraus sie mit minimalen Änderungen übernommen wurden. Außerdem sind viele Bibliothekserweiterungen der 1999 überarbeiteten Programmiersprache C (C99) in einer an C++ angepassten Form enthalten.

Mit Ausnahme der numerischen und mathematischen Bibliotheken wurde die Übernahme aller TR1-Erweiterungen in die kommende Sprachnorm C++0x bereits vom C++-Standardisierungskomitee beschlossen. Ein weiterer Bibliothekszusatz namens TR2, mit Funktionalität für den nebenläufigen Betrieb (Threads), ist in Vorbereitung und soll nach dem kommenden Standard veröffentlicht werden.

Der aktuelle Entwurf des kommenden C++-Standards C++0x kann unter <http://www.open-std.org/JTC1/SC22/WG21/>¹ (unter „current draft“) eingesehen werden. Die Veröffentlichung der Sprachnorm bis Ende 2009, dafür steht die 0 in C++0x, wurde bereits verfehlt.

¹ <http://www.open-std.org/JTC1/SC22/WG21/>

3 Compiler

Dieses Kapitel ist noch nicht vollständig

3.1 Was passiert beim Übersetzen

3.1.1 Der Präprozessor

In C++ gibt es so wie bei C einen Präprozessor, der vor dem Lauf des Übersetzers gewisse Zeilen des Codes umformt. Das Ergebnis wird dann dem Übersetzer übergeben. Diese Zeilen beginnen mit einem Doppelkreuz (#), gefolgt von einer Anweisung. Beispiele sind

```
#include "Dateiname.Erweiterung"  
#include <Dateiname.Erweiterung>  
#define  
#ifndef  
#endif  
#pragma once
```

Der Präprozessor führt Ersetzungen auf Textbasis aus, das heißt, es findet keinerlei kontextabhängige Prüfung statt, ob dabei korrektes C++ entsteht.

3.1.2 Übersetzer

Der Übersetzer (engl. *compiler*) ist das Programm, das aus dem vom Präprozessor erzeugten C++-Code Objektdatei¹en erzeugt. Diese Dateien beinhalten betriebssystemnahen Maschinencode, der aber auch auf externe Funktionen zugreift, wobei extern bedeutet, dass sie nicht in derselben Datei vorhanden sind. Sollten syntaktische Fehler im Code enthalten sein, gibt der Übersetzer mehr oder weniger präzise Fehlermeldungen aus – je nachdem, von was für einem Hersteller er stammt, welcher Modus eingestellt ist, usw. Der Fehlerberichtmodus bestimmt, welche Fehlermeldungen ausgegeben werden. Bei zurückhaltender Einstellung werden nur Fehlermeldungen ausgegeben. Wird die Ausgabemenge durch einen andern Modus erweitert, werden auch Warnungen und dann auch Hinweise hinzugefügt, die auf einen eventuellen Fehler verweisen.

3.1.3 Der Bindelader

Der Bindelader (engl. *linker*, selten auch *Binder* genannt) bindet die vom Übersetzer gelieferten Objektdateien zu einem ausführbaren Code, indem alle Verweise auf externe oder auf Betriebssystem-

¹ <http://de.wikipedia.org/wiki/Objektdatei>

temfunktionen aufgelöst werden. Sollte diese Auflösung nicht gelingen, meldet der Binder, dass versucht wird, auf unauffindbare Funktionen oder Objekte zuzugreifen.

3.2 Hilfsmittel

1. Compiler Liste (mit Betriebssystem)
2. IDE Liste (mit Betriebssystem und möglichen Compilern)

3.3 Die GCC

GCC steht für **G**nu **C**ompiler **C**ollection und ist eine Sammlung von freien Compilern. GCC stellt neben dem C++-Compiler namens "g++" noch viele andere Compiler für verschiedene Sprachen bereit. Der C-Compiler ist etwa gcc und der Java-Compiler heißt gcj.

GCC steht unter der GNU General Public License (GNU GPL²), das heißt, GCC ist Open Source und jeder ist berechtigt, den Quellcode einzusehen, zu verändern und natürlich auch GCC zu verwenden. Da GCC auf sehr viele Plattformen portiert wurde und unter nahezu allen Betriebssystemen verfügbar ist, eignet er sich prächtig, um ihn in einem Buch wie diesem zu verwenden.

Auf fast allen Linux-, Unix- und unixartigen Systemen ist GCC schon standardmäßig installiert. Falls nicht, können Sie ihn mit dem Paketmanager Ihrer (Linux-)Distribution nachinstallieren oder ihn von gcc.gnu.org³ herunterladen und manuell installieren. Auf Windows können Sie MinGW (eine Portierung von GCC) verwenden. Sie ist zwar nicht so aktuell wie die originale GCC-Version, aber für den Anfang immer noch ausreichend. Alternativ können Sie GCC auf Windows auch über Cygwin⁴ nutzen.

Aber jetzt ist Schluss mit der Theorie. Schauen wir uns erst einmal an, wie g++ benutzt wird. Im folgenden wird angenommen, dass eine Datei mit dem Namen *prog.cpp* vorhanden ist. In dieser Datei könnte zum Beispiel folgendes stehen:

```
#include <iostream>

using namespace std;

int main(){
    cout << "Hallo Welt\n";
}
```

Ausgabe

```
Hallo Welt
```

2 <http://de.wikipedia.org/wiki/GNU%20General%20Public%20License>

3 <http://gcc.gnu.org>

4 <http://de.wikipedia.org/wiki/Cygwin>

Dieses Programm müssen Sie noch nicht verstehen, im Kapitel Hallo, du schöne Welt!⁵ wird es erklärt. Nun geben Sie auf der Kommandozeile (Linux: Shell, Windows: Eingabeaufforderung) folgendes ein:

```
g++ prog.cpp
```

`g++` ist der Name des Programms, welches aufgerufen wird, also der Compiler `g++`. `prog.cpp` ist der Name der Datei, die kompiliert werden soll. Wenn Sie diesen Befehl ausführen, sehen Sie entweder Fehlermeldungen auf dem Bildschirm, oder aber Sie bekommen eine Datei mit dem Namen `a.out`. Manchmal bekommen Sie auch sogenannte „warnings“, also Warnungen. Bei diesen Warnungen wird der Code zwar kompiliert, aber Sie sollten versuchen, warnungsfreie Programme zu schreiben. Ein Beispiel für eine Warnung könnte z.B. sein:

```
g++ prog.cpp
prog.cpp: In function 'int main()':
prog.cpp:17: warning: comparison between signed and unsigned integer
expressions
```

In diesem Beispiel würde die Warnung bedeuten, dass wir eine Zahl ohne Vorzeichen (`unsigned`) und eine mit Vorzeichen (`signed`) vergleichen, und zwar innerhalb der Funktion `'int main()'`, genauer gesagt in Zeile 17. Was `unsigned` und `signed` ist, erfahren Sie im Kapitel Variablen, Konstanten und ihre Datentypen⁶.

Es gibt auch einige, zum Teil hilfreiche Warnungen, die nicht angezeigt werden. Um diese zu sehen, müssen Sie die Option `-Wall` hinzufügen.

```
g++ -Wall prog.cpp
```

Um sich noch mehr Warnungen anzeigen zu lassen (`-Wall` zeigt auch nicht alle), können Sie auch noch `-Wextra` benutzen:

```
g++ -Wall -Wextra prog.cpp
```

Es wird empfohlen, diese Möglichkeit zu nutzen, vor allem wenn Sie auf der Suche nach Fehlern in Ihrem Programm sind.

Möchten Sie, dass das fertige Programm einen anderen Namen als `a.out` hat, so können Sie es natürlich jedesmal umbenennen. Dies ist aber umständlich und somit nicht zu empfehlen. Ein viel eleganterer Weg ist das Benutzen der Option `-o`.

```
g++ -o tollername prog.cpp
```

Der Dateiname nach der Option `-o` gibt an, in welche Datei das kompilierte Programm gespeichert werden soll. So erhalten Sie in diesem Beispiel die ausführbare Datei `tollername`.

⁵ Kapitel 5 auf Seite 19

⁶ Kapitel 9 auf Seite 33

Sollten Sie sich einmal nicht mehr erinnern, was eine bestimmte Funktion bewirkte oder wie sie hieß, so können Sie `g++` einfach nur mit der Option `--help` aufrufen.

```
g++ --help
```

`g++` gibt dann eine kurze Auflistung der Optionen aus. Wer gerne eine detailliertere Version hätte, kann unter Linux auch das Manualprogramm „man“ benutzen:

```
man g++
```

Allerdings ist diese Hilfe etwas lang (über 15 000 Zeilen...). Außerdem gibt es unter Linux noch eine übersichtliche Dokumentation bei `info`.

```
info g++
```

3.4 Für Fortgeschrittenere

Jetzt sollten Sie erst einmal den Rest des Kapitels überspringen und mit einigen der folgenden Buchkapitel weitermachen, denn für den Rest ist ein wenig Vorwissen sehr hilfreich. Sobald Sie mit den ersten paar Kapiteln fertig sind, können Sie hierher zurückkehren und den Rest lesen.

Was passiert überhaupt, wenn Sie `g++` aufrufen? Nun, als erstes wird der Code vom Präprozessor durchgeschaut und bearbeitet. (Natürlich bleibt Ihre Quelldatei, wie sie ist.) Dabei werden beispielsweise Makros ersetzt und Kommentare gelöscht. Dieser bearbeitete Code wird dann vom Compiler in die Assemblersprache übersetzt. Die Assemblersprache ist auch eine Programmiersprache, welche aber nur die Maschinensprache so darstellt, dass ein Mensch sie (leichter) lesen kann. Schließlich wird diese Assemblersprache von einem Assembler in Maschinencode umgewandelt. Zum Schluss wird noch der Linker aufgerufen, der die einzelnen Programmdateien und die benutzten Bibliotheken "verbindet".

Darum, dass dies alles in korrekter Reihenfolge richtig ausgeführt wird, brauchen Sie sich nicht zu kümmern; `g++` erledigt das alles für Sie. Allerdings kann es manchmal nützlich sein, dass das Programm noch nicht gelinkt wird, etwa wenn Sie mehrere Quelldateien haben. Dann kann man einfach dem `g++` die Option `-c` mitgeben.

```
g++ -c -o prog.o prog.cpp
```

Durch diesen Aufruf erhalten wir eine Datei `prog.o`, die zwar kompiliert und assembliert, aber noch nicht gelinkt ist. Deswegen wurde die Datei auch `prog.o` genannt, da ein kompiliertes und assembliertes, aber nicht gelinktes Programm als Objektdatei vorliegt. Objektdateien bekommen üblicherweise die Endung `*.o`. Ohne die `-o` Option hätten Sie möglicherweise eine Datei gleichen Namens erhalten, aber das ist nicht absolut sicher. Es ist besser den Namen der Ausgabedatei immer mit anzugeben; so können Sie sicher sein, dass die ausgegeben Dateien auch wirklich den von Ihnen erwarteten Namen haben.

Bei g++ gibt es Unmengen von Optionen, mit denen Sie fast alles kontrollieren können. So gibt es natürlich auch eine Option, durch die kompiliert, aber nicht assembliert wird. Diese Option heißt `-S`.

```
g++ -S prog.cpp
```

Diese Option wird allerdings fast nie benötigt, es sei denn Sie interessieren sich dafür, wie Ihr Compiler Ihren Code in Assembler umsetzt. Die Option `-E` ist schon nützlicher. Mit ihr wird nur der Präprozessor ausgeführt:

```
g++ -E prog.cpp
```

So können Sie z. B. sehen, ob mit den Makros alles ordnungsgemäß geklappt hat oder die Headerdateien eventuell in einer von Ihnen unerwarteten Reihenfolge inkludiert wurden. Eine Warnung sollen Sie hier aber mit auf den Weg bekommen. Wenn der Präprozessor durchgelaufen ist, stehen auch alle mittels `#include` eingebundenen Headerdateien der Standardbibliothek mit im Quelltext, die Ausgabe kann also ziemlich lang werden. Allein das Einbinden von `iostream` produziert bei g++ 4.6.3 knapp 19 000 Zeilen Code. Welche Zeit es in Anspruch nimmt, den Quellcode nach dem Übersetzen in Assembler (Option `-s`) zu lesen, dürfte damit geklärt sein.

Sie sollten auch die Option `-ansi` kennen. Da g++ einige C++ Erweiterungen beinhaltet, die nicht im C++-Standard definiert sind oder sogar mit ihm im Konflikt stehen, ist es nützlich, diese Option zu verwenden, wenn Sie ausschließlich mit Standard-C++ arbeiten wollen. In der Regel ist das zu empfehlen, da sich solcher Code *viel* leichter auf andere Compiler portieren lässt. Im Idealfall müssten Sie gar keine Änderungen mehr vornehmen. Da aber weder g++ noch irgendein anderer Compiler absolut standardkonform sind, ist das selten der Fall. Ein Negativbeispiel für standardkonforme Compiler kommt immer mal wieder aus Redmond. Die dort ansässige Firma produziert in der Regel Produkte, die zu ihren selbstdefinierten Standards in einer bestimmten Version (also nicht mal untereinander) kompatibel sind. Beschwerden Sie sich bitte nicht, wenn einige Programmbeispiele mit deren Compiler nicht kompiliert werden. Der Fairness halber soll trotzdem angemerkt werden, dass langsam aber doch sicher Besserung in Sicht ist.

```
g++ -ansi -o prog prog.cpp
```

Dies bewirkt, dass diese nicht standardkonformen Erweiterungen des g++-Compilers abgeschaltet werden. Solcher Code ist in der Regel die bessere Wahl, da er auch in Zukunft, wenn es neue Compiler oder Compiler-Versionen geben wird, noch mit ihnen übersetzt werden kann.

Möchten Sie eine Warnung erhalten, wenn nicht standardkonforme Erweiterungen von g++ verwendet werden, dann nutzen Sie die Option `-pedantic`:

```
g++ -pedantic -o prog prog.cpp
```

Um die Optimierung zuzuschalten, nutzen Sie die Option `-Ox`, wobei das x für eine Zahl von 1 bis 3 steht. 3 bedeutet stärkste Optimierung.

```
g++ -O3 -o prog prog.cpp
```

Programme, die mit Optimierung übersetzt wurden, sind kleiner und laufen schneller. Der Nachteil besteht in der höheren Zeit, welche für die Übersetzung an sich benötigt wird. Daher sollten Sie die Optimierung nur zuschalten, wenn Sie eine Programmversion erstellen, die Sie auch benutzen möchten. Beim Austesten während der Entwicklung ist es besser, ohne Optimierung zu arbeiten, da häufig kompiliert wird. Eine lange Wartezeit, nur um dann einen Programmdurchlauf zu machen, ist schlicht nervtötend.

3.5 Makefiles

3.5.1 Was ist make?

Bei make handelt es sich um ein Werkzeug, mit dem man die Abhängigkeiten eines Build Prozesses auflösen kann. Dieses Stück Software ist schon sehr alt und in unterschiedlichsten Implementierungen verfügbar, die verbreitetsten sind wohl GNU make⁷ und BSD make⁸. Leider sind die verschiedenen Varianten untereinander nicht ganz kompatibel.

3.5.2 Makefiles per Hand erstellen

make kann sehr viel mehr, als hier beschrieben werden könnte, es folgt daher lediglich eine kurze Erläuterung, um was es eigentlich geht. In einem Makefile lassen sich Regeln beschreiben, wie bestimmte "Targets" (Ziele) erstellt werden können. Diese können von anderen Zielen oder Dateien abhängen.

Beispielsweise erstellt man eine Objektdatei aus einer Sourcedatei, indem man sie kompiliert. Dazu muss aber natürlich die Sourcedatei vorhanden sein. Eine solche Regel könnte zum Beispiel so aussehen:

```
hello.o: hello.c
    $(CC) -c $(CFLAGS) hello.c
```

Die erste Zeile besagt, dass zum Erstellen von `hello.o` die Datei `hello.c` benötigt wird. Die Zweite sagt aus, wie das Erstellen von `hello.o` zu bewerkstelligen ist. Variablen werden mit `$` eingeleitet. So beinhaltet zum Beispiel `$(CC)` in der Regel den Namen des C Compilers.

Derartige Regeln kann man auch mit Wildcards⁹ versehen, so kann man eine Regel erstellen, mit der man ausdrücken kann, wie generell aus einer `*.c` eine `*.o` Datei zu erstellen ist:

```
%.o: %.c
    $(CC) -c $(CFLAGS) $<
```

7 <http://www.gnu.org/software/make/>

8 <http://www.freebsd.org/cgi/man.cgi?query=make&sektion=1>

9 http://de.wikipedia.org/wiki/Wildcard_%28Informatik%29

Dabei steht die spezielle Variable `$<` für den Namen der tatsächlichen Source Datei, wie zum Beispiel `hello.c`.

Beispiel:

```
CC = gcc
OBJECTS = cbg.o
LIBS = -lcurl
CFLAGS = -Wall -O2
BINDIR = $(DESTDIR)/usr/bin
NAME = cbg
cbg: $(OBJECTS)
    $(CC) -o $(NAME) $(OBJECTS) $(LIBS)
%.o: %.c
    $(CC) -c $(CFLAGS) $<
install:
    install --mode=755 $(NAME) $(BINDIR)/
clean:
    rm *.o $(NAME)
uninstall:
    rm $(BINDIR)/$(NAME)
```

Damit ist es nun möglich, die einzelnen Ziele zu erstellen:

```
make install
make clean
make uninstall
```

Wird kein Ziel angegeben, so wird das erste Ziel erstellt, in obigen Beispiel also `cbg`.

3.5.3 Automake

3.6 Boost Build

Eine sehr gute alternative zu Makefiles. <http://www.boost.org/doc/tools/build/index.html>

4 GUIs und C++

4.1 Was ist ein GUI?

GUI kommt vom englischen „Graphical User Interface“, was soviel heißt wie „Graphische Benutzerschnittstelle“. Grob gesagt ist es das, was der Benutzer von den meisten Programmen zu sehen bekommt. Also: Die Menüleiste, Textfelder, Buttons u.s.w.

Weiterhin gibt es natürlich noch das „Command Line Interface“ (kurz CLI, zu Deutsch "Kommandozeilenschnittstelle"), das vielen als Fenster mit schwarzem Hintergrund und weißer Schrift bekannt ist.

4.2 C++ und GUIs

C++ bringt von Haus aus kein GUI mit, da es als hardwarenahe plattformunabhängige Sprache erstellt wurde und GUIs stark vom verwendeten Betriebssystem abhängen. Dieser „Mangel“ könnte einige Neueinsteiger vielleicht auf den ersten Blick abschrecken, da sie sich einen Einstieg in C++ oder gar in die gesamte Programmierung mit vielen Buttons, Textfeldern, Statusanzeigen und Menüeinträgen erhofft haben.

Wer jedoch einmal damit angefangen hat, kleinere Programme zu schreiben, wird schnell merken, dass der Reiz hier nicht von bunten Farben und blinkenden Symbolen ausgeht. Ganz im Gegenteil. Gerade für den Anfänger ist es am besten, wenn die eigentliche Funktion nicht durch Tonnen von "Design-Code" überdeckt und unleserlich gemacht wird. Hat man erst einmal die Grundlagen erlernt, so ist es nicht schwer auch ansprechende Oberflächen zu erstellen.

Es gibt eine ganze Reihe von Bibliotheken, die eine weitgehend plattformübergreifende Programmierung mit C++ als Sprache ermöglichen. Aber auch die Programmierung über die API (für engl. „application programming interface“, deutsch: „Schnittstelle zur Anwendungsprogrammierung“) des spezifischen Betriebssystems ist natürlich möglich.

4.3 Einsatz von GUI

Sobald man mit C++ dann doch ein GUI programmieren will, sollte auf externe Bibliotheken zurückgegriffen werden. Alle GUIs erfordern ein Verständnis der C++ Grundlagen. Nachfolgend sind einige Bibliotheken zur GUI-Programmierung aufgelistet. Viele der Bibliotheken lassen sich auch mit anderen Programmiersprachen einsetzen.

4.3.1 Qt

Qt ist eine leistungsstarke plattformübergreifende Klassenbibliothek mit Meta-Object Compiler (gennant: *moc*), die von der norwegischen Firma Trolltech (inzwischen Qt Software und Teil von Nokia) entwickelt wird. Die Klassenbibliothek ist unter der GNU General Public License (GPL) und der GNU Lesser General Public License (LGPL) lizenziert. Es gibt außerdem eine proprietäre Lizenz, die allerdings lediglich zusätzlichen technischen Support beinhaltet.

Buchempfehlung

Das Buch „Qt für C++-Anfänger“^a gibt einen Einstieg in die Programmierung mit Qt. Es behandelt nicht viele Themen und ist somit wirklich nur zum Kennenlernen des Frameworks geeignet. Dafür sind die vorhandenen Seiten aber auch gut gefüllt.

^a <http://de.wikibooks.org/wiki/Qt%20f%FCr%20C%2B%2B-Anf%E4nger>

4.3.2 GTK+

Das GIMP-Toolkit (abgekürzt: GTK+) ist eine freie Komponentenbibliothek, die in C geschrieben ist und unter der GNU Lesser General Public License (LGPL) steht. Um dieses Framework in C++ zu nutzen gibt es ein Framework namens Gtkmm¹. Wie Qt ist es für viele Plattformen verfügbar.

4.3.3 wxWidgets

wxWidgets ist ein auf C++ basierendes Open-Source-Framework zur plattformunabhängigen Entwicklung von Anwendungen mit grafischer Benutzeroberfläche (GUI). Die wxWidgets-Lizenz ist eine leicht modifizierte LGPL und erlaubt daher ebenfalls die freie Verwendung in proprietärer und freier Software und den weiteren Vertrieb unter einer selbst gewählten Lizenz.

4.3.4 MFC

Die Microsoft Foundation Classes (MFC) sind ein C++ Framework, vor allem zur GUI-Programmierung unter Microsoft Windows. Es verwendet Entwurfsmuster, insbesondere Document-View (Model-View-Controller), Singleton, Bridge und Observer, um Funktionalität der Windows-API zu kapseln. Außerdem gibt es eine enorme Anzahl an Hilfsmakros zur Unterstützung bei RTTI („Runtime Type Information“), Nachrichtenbehandlung, dynamischer Instanziierung etc.. Da sie seit 1992 in jeder -nicht Express- Version von Microsoft Visual C++ enthalten sind ist ihre Verbreitung auf der Windows-Plattform sehr hoch. Die Verwendung ist an eine kommerzielle Lizenz gebunden, den Besitz einer kostenpflichtigen Version von Visual C++, die MFC-Laufzeitumgebung ist kostenlos. Für die GUI-Entwicklung mit C++ -nicht C++/CLI- auf Windows sind die MFC das am weitesten verbreitete Framework das direkt auf der Windows-API aufsetzt.

¹ <http://www.gtkmm.org/de/index.html>

5 Hallo, du schöne Welt!

Es ist eine alte Tradition, eine neue Programmiersprache mit einem „Hello-World“-Programm einzuweihen. Auch dieses Buch soll mit der Tradition nicht brechen, hier ist das „Hello-World“-Programm in C++:

```
#include <iostream> // Ein- und Ausgabebibliothek

int main() { // Hauptfunktion
    std::cout << "Hallo, du schöne Welt!" << std::endl; // Ausgabe

    return 0; // Optionale Rückgabe an das
    Betriebssystem
}
```

Ausgabe

```
Hallo, du schöne Welt!
```

Zugegebenermaßen ist es nicht die Originalversion, sondern eine Originellversion von „Hello-World“. Wenn Sie das Programm ausführen, bekommen Sie den Text „Hallo, du schöne Welt!“ am Bildschirm ausgegeben. Sie wissen nicht, wie Sie das Programm ausführen können? Dann lesen Sie doch einmal das Kapitel über Compiler¹.

`#include <iostream>` stellt die nötigen Befehle zur Ein- und Ausgabe bereit. Als nächstes beginnt die „Hauptfunktion“ `main()`. Diese Hauptfunktion wird beim Ausführen des Programms aufgerufen. Sie ist also der zentrale Kern des Programms. Wenn Funktionen im Text erwähnt werden, stehen dahinter übrigens immer Klammern, um sie besser von anderen Sprachbestandteilen wie beispielsweise Variablen unterscheiden zu können.

`std::cout` beschreibt den Standardausgabe-Strom. Dabei wird der Text meist in einem Terminal angezeigt (wenn die Ausgabe nicht in eine Datei oder an ein anderes Programm umgeleitet wird). Die beiden Pfeile (`<<`) signalisieren, dass der dahinterstehende Text auf die Standardausgabe „geschoben“ wird. Das `std::endl` gibt einen Zeilenumbruch aus und sorgt dafür, dass der Text jetzt am Bildschirm ausgegeben wird.

`return 0` beendet das Programm und zeigt dem Betriebssystem an, dass es erfolgreich ausgeführt wurde. Auf die Einzelheiten wird in den folgenden Kapiteln (oder Abschnitten) noch ausführlich eingegangen. Diese Zeile ist optional; wird sie nicht angegeben, gibt der Compiler implizit 0 zurück.

Im Moment sollten Sie sich merken, dass jeder C++-Befehl mit einem Semikolon (`;`) abgeschlossen wird und dass geschweifte Klammern (`{ . . . }`) Zusammengehörigkeit symbolisieren – so auch oben in der Hauptfunktion. Alles was zwischen den geschweiften Klammern steht, gehört zu ihr.

¹ Kapitel 3 auf Seite 9

Leerzeichen, Tabulatorzeichen und Zeilenumbrüche spielen für den C++-Compiler keine Rolle. Sie können das folgende Programm genauso übersetzen wie seine gut lesbare Version von weiter oben:

```
#include <iostream>
int
main (
) {std ::
cout
<< "Hallo, du schöne Welt!"<<std
:: endl
;return 0;
}
```

Ausgabe

```
Hallo, du schöne Welt!
```

Vorsichtig sollten Sie bei Zeilen sein, die mit # beginnen. Leerzeichen und Tabulatoren sind zwar auch hier bedeutungslos, aber Zeilenumbrüche dürfen nicht stattfinden.

Es ist übrigens (für Ihren Rechner) auch irrelevant, ob Sie etwas wie // Ein- und Ausgabebibliothek mit in Ihr Programm schreiben oder nicht. Es handelt sich dabei um sogenannte Kommentare, die Sie in Kürze auch genauer kennenlernen werden. Beachten sollten Sie übrigens, dass bei C++ die Groß- und Kleinschreibung relevant ist. Schlüsselwörter und Namen der Standardbibliothek werden stets kleingeschrieben. Für die Groß-/Kleinschreibung von selbstdefinierten Namen gibt es gewisse Konventionen, auf welche in einem späteren Kapitel eingegangen wird.

Hinweis

Bei allen in diesem Buch beschriebenen Programmen handelt sich um so genannte Kommandozeilenprogramme. Falls Sie eine IDE^a zur Entwicklung benutzen und das Programm direkt aus dieser heraus aufrufen, kann es Ihnen passieren, dass Sie nur einen kurzen Blitz von Ihrem Programm sehen, weil sich das Kommandozeilenfenster nach der Programmbeendigung sofort schließt.

In diesem Fall haben Sie drei Optionen:

- Rufen Sie eine Kommandozeile auf und führen Sie das Programm von dort manuell aus.
- Schauen Sie nach, ob es in Ihrer IDE eine Funktion gibt, die das Fenster nach Programmende noch einen Tastendruck lang offen hält. (Eine solche Option muss nicht vorhanden sein!)
- Fügen Sie einen Befehl zu ihrem Programm hinzu, der auf eine Eingabe wartet: `std::cin.get();`

^a <http://de.wikipedia.org/wiki/Integrierte%20Entwicklungsumgebung>

6 Einfache Ein- und Ausgabe

Ein- und Ausgaberroutinen geben Ihnen die Möglichkeit, mit einem Programm zu interagieren. Dieses Kapitel beschäftigt sich mit der Eingabe über die Tastatur und der Ausgabe auf der Konsole in C++-typischer Form. Die C-Variante werden Sie später noch kennenlernen.

Um die C++ Ein- und Ausgabe nutzen zu können, müssen Sie die Bibliothek `iostream` einbinden. Das geschieht mit:

```
#include <iostream>
```

Danach müssen die Befehle daraus bekannt gegeben werden, da sie sich in einem speziellen *Namensraum* befinden. Was Namensräume sind und wofür man sie einsetzt, werden Sie später noch erfahren. Um nun die Ein- und Ausgabebefehle nutzen zu können, müssen sie dem Compiler sagen: Benutze den Namensraum `std`. Da der Compiler das aber so nicht versteht, können Sie folgende gleichbedeutende Zeile benutzen:

```
using namespace std;
```

Der Namensraum „`std`“ heißt so viel wie Standard. Wenn etwas in diesem Namensraum steht, dann gehört es zur C++-Standardbibliothek und die sollten Sie (in der Regel) wann immer möglich einsetzen. In den Programmen dieses Buches wird der Namensraum immer direkt angegeben. Das heißt, wenn beispielsweise ein Objekt benutzt werden soll das im Namensraum „`std`“ liegt, wird ihm „`std::`“ vorangestellt. Die beiden Doppelpunkte heißen Bereichsoperator.

6.1 Einfache Ausgabe

Nun wollen wir aber endlich auch mal was Praktisches tun. Zugegebenermaßen nichts Weltbewegendes und im Grunde nicht einmal etwas wirklich Neues, denn Text haben wir ja schon im „Hello-World“-Programm ausgegeben.

```
#include <iostream>

int main(){
    std::cout << "Dieser Text steht nun in der Kommandozeile!";
    std::cout << "Dieser Text schließt sich direkt an...";
}
```

Ausgabe

Dieser Text steht nun in der Kommandozeile!Dieser Text schließt sich direkt an...

Wie der Text bereits selbst sagte, er erscheint in der Kommandozeile und der zweite schließt sich sehr direkt an. Wenn sie innerhalb einer Zeichenkette einen Zeilenumbruch einfügen möchten, gibt es zwei Möglichkeiten. Sie können die *Escape-Sequenz* `\n` in die Zeichenkette einfügen oder den *Manipulator* `endl` benutzen. Was genau Escape-Sequenzen oder Manipulatoren sind ist Thema eines späteren Kapitels, aber folgendes Beispiels demonstriert schon mal die Verwendung für den Zeilenumbruch:

```
#include <iostream>

int main(){
    std::cout << "Text in der Kommandozeile!\n";           // Escape-Sequenz \n
    std::cout << "Dieser Text schließt sich an...\n";       // Das steht in einer eigenen
    Zeile

    std::cout << std::endl;                                   // Leerzeile mittels endl

    std::cout << "Text in der Kommandozeile!" << std::endl; // Zeilenumbruch mit endl
    std::cout << "Dieser Text schließt sich an..." << std::endl; // Das steht in einer eigenen
    Zeile
}
```

Ausgabe

```
Text in der Kommandozeile!
Dieser Text schließt sich an...

Text in der Kommandozeile!
Dieser Text schließt sich an...
```

Beide Methoden haben scheinbar den gleichen Effekt, wo der kleine Unterschied liegt, ist allerdings auch Thema eines späteren Kapitels und im Moment noch nicht relevant.

6.2 Einfache Eingabe

Für die Eingabe muss ein wenig vorgegriffen werden, denn um etwas einzulesen, ist ja etwas nötig, worin das Eingelesene gespeichert werden kann. Dieser „Behälter“ nennt sich Variable. Eine Variable muss zunächst einmal angelegt werden, am besten lässt sich die Eingabe an einem Beispiel erklären:

```
#include <iostream>

int main(){
    int Ganzzahl;

    std::cout << "Benutzereingabe: ";
```

```
std::cin >> Ganzzahl;

std::cout << "Sie haben " << Ganzzahl << " eingegeben.";
}
```

Ausgabe

```
Benutzereingabe: 643
Sie haben 643 eingegeben.
```

Es wird, wie bereits erwähnt, erst eine Variable angelegt (`int Ganzzahl;`), welcher dann ein Wert zugewiesen wird (`cin >> Ganzzahl;`). Diese zweite Zeile bedeutet so viel wie: lies eine ganze Zahl von der Tastatur und speichere sie in der Variablen `Ganzzahl`.

`cin` ist sozusagen die Tastatur, `Ganzzahl` ist der Behälter und `>>` bedeutet so viel wie „nach“. Zusammen ergibt sich „Tastatur nach Behälter“, es wird also der „Inhalt“ der Tastatur in die Variable `Ganzzahl` verschoben. Dass eine `Ganzzahl` von der Tastatur gelesen wird, ist übrigens vom Datentyp der Variable abhängig, aber dazu später mehr.

Um den Inhalt der Variable wieder auszugeben, müssen Sie nichts weiter tun, als sie mit einem weiteren Schiebeoperator (`<<`) hinter `cout` anzuhängen. Es ist ohne Weiteres möglich, mehrere solcher Schiebeoperatoren hintereinander zu schalten, solange Sie nur die letzte Ein- oder Ausgabe mit einem Semikolon (`;`) abschließen. Bei der Eingabe muss natürlich der `>>`-Operator statt dem `<<`-Operator benutzt werden. Die Reihenfolge der Ein- oder Ausgabe bei solchen Konstruktionen entspricht der eingegebenen Folge im Quelltext. Was zuerst hinter `cout` oder `cin` steht, wird also auch zuerst ausgeführt.

Fragen

Frage 1

Was ist hier verkehrt:

```
int Ganzzahl;  
std::cin << Ganzzahl;
```

Antwort

Der Operator muss >> sein.

Frage 2

Funktioniert folgendes Beispiel:

```
#include <iostream>  
  
int main(){  
    int Ganzzahl;  
  
    std::cin >> Ganzzahl;  
  
    std::cout << "Sie haben "  
              << Ganzzahl  
              << " eingegeben.";  
}
```

Antwort

Das Beispiel funktioniert hervorragend. Zeilenumbrüche sind kein Problem, solange sie nicht innerhalb einer Zeichenkette stehen. Wenn Sie allerdings beabsichtigen, einen Zeilenumbruch bei der Ausgabe zu erzeugen, müssen Sie `\n` oder `endl` benutzen.

7 Kommentare

In allen Programmiersprachen gibt es die Möglichkeit, im Quelltext Notizen zu machen. Für andere oder auch für sich selbst, denn nach ein paar Wochen werden Sie möglicherweise Ihren eigenen Quelltext nicht mehr ohne Weiteres verstehen. Kommentare helfen Ihnen und anderen besser und vor allem schneller zu verstehen, was der Quelltext bewirkt. In C++ gibt es zwei Varianten, um Kommentare zu schreiben:

```
// Ein Kommentar, der mit zwei Schrägstrichen eingeleitet wird, geht bis zum Zeilenende

/* Ein Kommentar dieser Art kann
   sich über mehrere Zeilen
   erstrecken oder ... */

a = b /* ... vor dem Zeilenende enden. */ + c;
```

Die erste, bis zum Zeilenende geltende Sorte, ist die moderne Art des Kommentars. Sie ist in der Regel vorzuziehen, da sie einige Vorteile gegenüber der alten, noch aus C stammenden Variante hat. Die zweite Sorte (manchmal auch als C-Kommentar bezeichnet) in seiner mehrzeiligen Form sollte nur an Stellen verwendet werden, an denen längere Textpassagen stehen und möglichst nicht zwischen Code. Anwendung finden solche Kommentare oft am Dateianfang, um den Inhalt kurz zusammenzufassen oder Lizenzrechtliches zu regeln.

Der hauptsächliche Nachteil bei den mehrzeiligen Kommentaren besteht darin, dass man sie nicht „verschachteln“ kann. Oft werden beispielsweise Teile des Quellcodes zu Testzwecken kurzweilig auskommentiert. Folgendes Beispiel soll dies demonstrieren:

```
#include <iostream>                                     /* Ein- und Ausgabebibliothek */

int main(){                                           /* Hauptfunktion */
/*
   std::cout << "Hallo, du schöne Welt!" << std::endl; /* Ausgabe */
*/
}
```

Das würde nicht funktionieren. Wenn hingegen die anderen Kommentare benutzt werden, gibt es solche Probleme nicht:

```
#include <iostream>                                     // Ein- und Ausgabebibliothek
```

```
int main() {                                     // Hauptfunktion
    /*
    std::cout << "Hallo, du schöne Welt!" << std::endl; // Ausgabe
    */
}
```

Im ersten Beispiel wird die Einleitung von `/* Ausgabe */` einfach ignoriert. Der abschließende Teil beendet den Kommentar, der die Code-Zeile auskommentieren soll und der eigentliche Abschluss führt zu einem Kompilierfehler. Zugegeben, das ist nicht weiter schlimm, denn solch ein Fehler ist schnell gefunden, aber ihn von vorn herein zu vermeiden, ist eben noch zeitsparender. Im übrigen muss bei einzeiligen Kommentaren oft weniger geschrieben werden. Eine Ausnahme bilden Kommentare, die einfach zu lang sind, um sie auf eine Zeile zu schreiben. Dennoch sollten auch sie durch `//`-Kommentare realisiert werden.

```
// Ich bin ein Beispiel für einen langen Kommentar, der durch doppelte
// Schrägstriche über mehrere Zeilen geht. Würde ich am Dateianfang stehen,
// hätte man mich wahrscheinlich mit anderen Kommentarzeichen ausgestattet,
// aber da ich hier eindeutig von einer Riesenmenge Quelltext umgeben bin,
// hat man sich trotz der erhöhten Schreibarbeit für // entschieden
```

Tip

Viele Texteditoren enthalten eine Tastenkombination, über die sich Text ein- und auskommentieren lässt. Besonders für längere Codepassagen ist dies nützlich.

8 Rechnen (lassen)

In diesem Kapitel soll unser Rechner einmal das tun, was er ohnehin am besten kann: Rechnen. Wir werden uns derweil zurücklehnen und zusehen oder besser gesagt werden wir das tun, nachdem wir ihm mitgeteilt haben, was er rechnen soll.

8.1 Einfaches Rechnen

```
#include <iostream>

int main(){
    std::cout << "7 + 8 = " << 7 + 8 << std::endl; // Ausgabe einer Rechnung
}
```

Ausgabe

```
7 + 8 = 15
```

Zugegebenermaßen hätten Sie diese Rechnung wahrscheinlich auch im Kopf lösen können, aber warum sollten Sie sich so unnötig anstrengen. Ihr Rechner liefert doch auch das richtige Ergebnis und wenn Sie dies mit der Eingabe von Zahlen kombinieren, können Sie sogar bei jedem Programm-durchlauf zwei unterschiedliche Zahlen addieren:

```
#include <iostream>

int main(){
    int Summand1, Summand2; // Anlegen von zwei Variablen

    std::cin >> Summand1 >> Summand2; // Zwei Zahlen eingeben

    std::cout << Summand1 << " + " << Summand2 // beide durch " + " getrennt wieder ausgeben
              << " = " // " = " ausgeben
              << Summand1 + Summand2 // Ergebnis berechnen und ausgeben
              << std::endl; // Zeilenumbruch
}
```

Ausgabe

```
Benutzereingabe: 774
Benutzereingabe: 123
774 + 123 = 897
```

Das Ergebnis lässt sich natürlich auch in einer Variable zwischenspeichern. Folgendes Beispiel demonstriert diese Möglichkeit:

```
#include <iostream>

int main(){
    int Summand1, Summand2, Ergebnis;           // Anlegen von drei Variablen

    std::cin >> Summand1 >> Summand2;         // Zwei Zahlen eingeben

    Ergebnis = Summand1 + Summand2;           // Ergebnis berechnen

    std::cout << Summand1 << " + " << Summand2 // beide durch " + " getrennt wieder ausgeben
              << " = "                          // " = " ausgeben
              << Ergebnis                          // Ergebnis ausgeben
              << std::endl;                       // Zeilenumbruch
}

```

Ausgabe

```
Benutzereingabe: 400
Benutzereingabe: 300
400 + 300 = 700

```

8.2 Die großen Vier

C++ beherrscht die vier Grundrechenarten: Addition (+), Subtraktion (-), Multiplikation (*) und Division (/). Genau wie in der Mathematik gilt auch in C++ die Regel Punktrechnung geht vor Strichrechnung und Klammern gehen über alles. Das folgende Beispiel soll eine komplexere Rechnung demonstrieren:

```
#include <iostream>

int main(){
    int Ergebnis;                               // Anlegen einer Variable

    Ergebnis = ((3 + 3*4)/5 - 1)*512 - 768;     // Ergebnis berechnen

    std::cout << "((3 + 3*4)/5 - 1)*512 - 768 = " // Aufgabe ausgeben
              << Ergebnis                          // Ergebnis ausgeben
              << std::endl;                       // Zeilenumbruch
}

```

Ausgabe

$$((3 + 3*4)/5 - 1)*512 - 768 = 256$$

Gerechnet wird in dieser Reihenfolge:

3 * 4 = 12
3 + 12 = 15
15 / 5 = 3
3 - 1 = 2
2 * 512 = 1024
1024 - 768 = 256

Sie sollten darauf achten, immer die gleiche Anzahl öffnende und schließende Klammern zu haben, denn dies ist ein beliebter Fehler, der von Anfängern meist nicht so schnell gefunden wird. Compiler bringen in solchen Fällen nicht selten Meldungen, die einige Zeilen unter dem eigentlichen Fehler liegen.

8.3 Zusammengesetzte Operatoren

C++ ist eine Sprache für schreibfaule Menschen, daher gibt es die Möglichkeit, die Rechenoperatoren mit dem Zuweisungsoperator zu kombinieren. Dies sieht dann folgendermaßen aus:

```
Zahl = 22;  
Zahl += 5; // Zahl = Zahl + 5;  
Zahl -= 7; // Zahl = Zahl - 7;  
Zahl *= 2; // Zahl = Zahl * 2;  
Zahl /= 4; // Zahl = Zahl / 4;
```

Als Kommentar sehen Sie die Langfassung geschrieben. Diese Kurzschreibweise bedeutet nicht mehr, als dass die vor (!) dem Zuweisungsoperator stehende Rechenoperation mit der Variable auf der linken Seite und dem Wert auf der rechten Seite ausgeführt und das Ergebnis der Variable auf der linken Seite zugewiesen wird. Sie sollten diese Kurzschreibweise der ausführlichen vorziehen, da sie nicht nur die Finger schont, sondern auch noch ein wenig schneller ist.

Am besten werden Sie dies wahrscheinlich verstehen, wenn Sie es einfach ausprobieren. Stehen auf der rechten Seite noch weitere Rechenoperationen, so werden diese zuerst ausgeführt. Das Ganze stellt sich dann also folgendermaßen dar:

```
Zahl = 5;  
Zahl *= 3 + 4; // Zahl = Zahl * (3 + 4);
```

8.4 Inkrement und Dekrement

Inkrementieren bedeutet, den Wert einer Variable um 1 zu erhöhen, entsprechend bedeutet dekrementieren 1 herunterzählen. Dem Inkrementoperator schuldet C++ übrigens seinen Namen. Die beiden Operatoren gibt es jeweils in der Präfix und der Postfix Variante. Insgesamt ergeben sich also vier Operatoren:

```
Zahl = 5;

Zahl++; // Inkrement Postfix (Zahl == 6)
++Zahl; // Inkrement Präfix (Zahl == 7)
Zahl--; // Dekrement Postfix (Zahl == 6)
--Zahl; // Dekrement Präfix (Zahl == 5)
```

Der Unterschied zwischen Inkrement (++) und Dekrement (--) ist sofort ohne größeres Nachdenken erkennbar. Der Sinn von Präfix und Postfix ergibt sich hingegen nicht sofort von selbst. C++ schuldet seinen Namen der Postfix-Variante.

Der Unterschied zwischen Präfix und Postfix besteht im Rückgabewert. Die Präfix-Variante erhöht den Wert einer Zahl um 1 und gibt diesen neuen Wert zurück. Die Postfix-Variante erhöht den Wert der Variable ebenfalls um 1, gibt jedoch den Wert zurück, den die Variable vor der Erhöhung hatte.

Das folgende kleine Programm zeigt den Unterschied:

```
#include <iostream>

int main(){
    int Zahl; // Anlegen einer Variable

    std::cout << "Zahl direkt ausgeben:\n";

    Zahl = 5; // Zahl den Wert 5 zuweisen

    std::cout << Zahl << ' '; // Zahl ausgeben

    Zahl++; // Inkrement Postfix (Zahl == 6)
    std::cout << Zahl << ' '; // Zahl ausgeben

    ++Zahl; // Inkrement Präfix (Zahl == 7)
    std::cout << Zahl << ' '; // Zahl ausgeben

    Zahl--; // Dekrement Postfix (Zahl == 6)
    std::cout << Zahl << ' '; // Zahl ausgeben

    --Zahl; // Dekrement Präfix (Zahl == 5)
    std::cout << Zahl << ' ';
```

```

std::cout << "\nRückgabewert des Operators ausgeben:\n";

Zahl = 5;                                // Zahl den Wert 5 zuweisen

std::cout << Zahl << ' ';                // Zahl ausgeben
std::cout << Zahl++ << ' ';              // Inkrement Postfix (Zahl == 6)
std::cout << ++Zahl << ' ';              // Inkrement Präfix (Zahl == 7)
std::cout << Zahl-- << ' ';              // Dekrement Postfix (Zahl == 6)
std::cout << --Zahl << ' ';              // Dekrement Präfix (Zahl == 5)

std::cout << "\nEndwert von Zahl: " << Zahl << std::endl;
}

```

Ausgabe

```

Zahl direkt ausgeben:
5 6 7 6 5
Rückgabewert des Operators ausgeben:
5 5 7 7 5
Endwert von Zahl: 5

```

Thema wird später näher erläutert...

In einem späteren Kapitel werden Sie noch ein paar zusätzliche Informationen erhalten, was beim Rechnen schief gehen kann und wie Sie es vermeiden. Wenn Sie beim Herumexperimentieren mit Rechenoperationen plötzlich scheinbar unerklärliche Ergebnisse erhalten, dann ist es an der Zeit einen Blick auf dieses Kapitel zu werfen.

Aufgaben

Aufgabe 1

Rechnen Sie die Ergebnisse der folgenden Aufgaben aus und schreiben Sie ein Programm, welches das Gleiche tut.

```

Zahl = (500 - 100*(2 + 1))*5
Zahl = (Zahl - 700)/3
Zahl += 50*2
Zahl *= 10 - 8
Zahl /= Zahl - 200

```

Musterlösung

```

#include <iostream>

int main(){
    int Zahl;

    Zahl = (500 - 100*(2 + 1))*5; // 1000
    Zahl = (Zahl - 700)/3;      // 100

```

Rechnen (lassen)

```
Zahl += 50*2;           // 200
Zahl *= 10 - 8;        // 400
Zahl /= Zahl - 200;    // 2
```

```
std::cout << "Zahl: " << Zahl;
```

```
}
```

Ausgabe

```
Zahl: 2
```

9 Variablen, Konstanten und ihre Datentypen

Variablen sind Behälter für Werte, sie stellen gewissermaßen das Gedächtnis eines Programms bereit. Konstanten sind Variablen, die ihren Wert nie verändern. Der Datentyp einer Variable oder Konstante beschreibt, wie der Inhalt zu verstehen ist. Ein Rechner kennt nur zwei Zustände: 0 und 1. Durch eine Aneinanderreihung solcher Zustände lassen sich verschiedene Werte darstellen. Mit acht aufgereihten Zuständen (= 8 Bit = 1 Byte) lassen sich bereits 256 verschiedene Werte darstellen. Diese Werte kann man als Ganzzahl, Zeichen, Wahrheitswert oder Gleitkommazahl interpretieren. Der Datentyp gibt Auskunft darüber, um was es sich handelt.

Der C++-Standard schreibt nicht vor, dass ein Byte aus genau 8 Bit bestehen muss, es kann auch größer sein. Es ist also möglich, dass auf einer speziellen Prozessorarchitektur eine Anzahl von 10 Bit als ein Byte festgelegt sind. Dies ist jedoch äußerst selten der Fall, daher werden wir im folgenden annehmen, dass ein Byte aus 8 Bit besteht.

9.1 Datentypen

Zunächst sollen die Datentypen von C++ beschrieben werden, denn sie sind grundlegend für eine Variable oder Konstante. Es gibt vier Gruppen von Datentypen: Wahrheitswerte, Zeichen, Ganzzahlen und Gleitkommazahlen.

9.1.1 Wahrheitswerte

Der Datentyp für Wahrheitswerte heißt in C++ `bool`, was eine Abkürzung für `boolean` ist. Er kann nur zwei Zustände annehmen: `true` (wahr) oder `false` (falsch). Obwohl eigentlich 1 Bit ausreichen würde, hat `bool` mindestens eine Größe von einem Byte (also 8 Bit), denn 1 Byte ist die kleinste adressierbare Einheit und somit die Minimalgröße für jeden Datentyp. Es ist auch durchaus möglich, dass ein `bool` beispielsweise 4 Byte belegt, da dies auf einigen Prozessorarchitekturen die Zugriffsgeschwindigkeit erhöht.

9.1.2 Zeichen

Zeichen sind eigentlich Ganzzahlen. Sie unterscheiden sich von diesen nur bezüglich der Ein- und Ausgabe. Jeder Zahl ist ein Zeichen zugeordnet, mit den Zahlen lässt sich ganz normal rechnen, aber bei der Ausgabe erscheint das zugeordnete Zeichen auf dem Bildschirm. Welches Zeichen welcher Zahl zugeordnet ist, wird durch den verwendeten Zeichensatz festgelegt.

Die meisten Zeichensätze beinhalten den sogenannten *ASCII-Code* (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange), welcher die Zeichen 0 – 127 belegt. Er enthält 32 Steuerzeichen (0 – 31) und 96 druckbare Zeichen (32 – 127).

`char` ist der Standard-Datentyp für Zeichen. Er ist in der Regel 1 Byte groß und kann somit 256 verschiedene Zeichen darstellen. Diese genügen für einen erweiterten ASCII-Code, welcher zum Beispiel auch deutsche Umlaute definiert. `wchar_t` ist ein Datentyp für Unicodezeichen und hat gewöhnlich eine Größe von 2 Byte, ist aber zunehmend auch mit 4 Byte zu finden. Die folgende Liste enthält einige nützliche Links zu Artikeln der Wikipedia:

- ASCII¹
- ISO 8859-1² (Latin-1)
- Codepage 437³ (DOS / DOS-Box)
- Unicode⁴

9.1.3 Ganzzahlen

Für ganze Zahlen sind die Datentypen `short`, `int` und `long` definiert. Weiterhin sind, wie schon gesagt, auch `char` und `wchar_t` ganzzahlige Datentypen. Mit Ausnahme von `wchar_t` kann jedem dieser Datentypen ein `signed` oder `unsigned` vorangestellt werden. `signed` bedeutet mit, `unsigned` ohne Vorzeichen, entsprechend hat der Datentyp dann einen negativen und einen positiven (`signed`) oder nur einen positiven (`unsigned`) Wertebereich, welcher dann aber doppelt so groß ist. `wchar_t` entspricht meistens dem Datentyp `unsigned short`, dies ist in ISO-C++ aber nicht vorgeschrieben.

ISO-C++ schreibt auch nicht die genaue Größe der Datentypen vor, es gibt lediglich die Reihenfolge bezüglich der Größe vor:

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

Außerdem ist festgelegt, dass `char` genau 1 Byte, `short` mindestens 2 Byte und `long` mindestens 4 Byte lang sein müssen. `int` hat (üblicherweise) auf 16-Bit-Rechnern eine Größe von 2 Byte und auf 32-Bit-Rechnern (auch 64-Bit-Rechner) eine Größe von 4 Byte. Die nachfolgende Tabelle zeigt die ganzzahligen Datentypen mit ihrer *üblichen Größe* und dem entsprechendem Wertebereich:

Typ	Speicherplatz	Wertebereich (dezimal)
<code>char</code>	1 Byte	- 128 bis +127 bzw. 0 bis 255
<code>signed char</code>	1 Byte	-128 bis +127
<code>unsigned char</code>	1 Byte	0 bis 255
<code>short</code>	2 Byte	-32768 bis +32767
<code>unsigned short</code>	2 Byte	0 bis 65535
<code>int</code>	4 Byte	- 2147483648 bis +2147483647
<code>unsigned int</code>	4 Byte	0 bis 4294967295
<code>long</code>	4 Byte	- 2147483648 bis +2147483647

1 <http://de.wikipedia.org/wiki/ASCII>
 2 <http://de.wikipedia.org/wiki/ISO%208859-1>
 3 <http://de.wikipedia.org/wiki/Codepage%20437>
 4 <http://de.wikipedia.org/wiki/Unicode>

Typ	Speicherplatz	Wertebereich (dezimal)
unsigned long	4 Byte	0 bis 4294967295

Auffällig ist, dass in der Tabelle nur für `char` eine `signed` Variante vorkommt. Das liegt daran, dass die übrigen Datentypen ohne den Zusatz `signed` immer vorzeichenbehaftet sind, für `char` ist dies hingegen nicht festgelegt. Dennoch können Sie das Schlüsselwort `signed` natürlich auch in Verbindung mit den anderen Datentypen (`signed short`, `signed int` und `signed long`) benutzen. In bestimmten Situationen kann dies zum Beispiel helfen die Übersicht zu erhöhen. Wenn Sie `char` für Zeichen benutzen, ist eine Angabe von `signed` oder `unsigned` nicht nötig, möchten Sie eine Variable dieses Datentyps zum Rechnen benutzen, sollten Sie die Schlüsselworte hingegen angeben.

Das waren jetzt viele Informationen auf wenig Raum. Merken Sie sich einfach in etwa die Größe der vier Datentypen, den Wertebereich können Sie dann entsprechend ableiten ($2^{\text{Anzahl der Bits}}$, 1 Byte = 8 Bit). Merken Sie sich weiterhin dass `signed` vorzeichenbehaftet, `unsigned` vorzeichenlos bedeutet und dass im Falle einer fehlenden Angabe `signed` angenommen wird, mit Ausnahme von `char` für den es eine compilerabhängige Festlegung gibt.

Wählen Sie ein `int`, wenn dieser Typ alle Zahlen des nötigen Wertebereichs aufnehmen kann, bei vorzeichenlosen Zahlen verwenden Sie `unsigned`. Reicht dieser Wertebereich nicht aus und ist `long` größer, dann nehmen Sie `long` (bzw. `unsigned long`). `short` und `unsigned short` sollte nur Verwendung finden, wenn Speicherplatz knapp ist, etwa bei Verwendung großer Arrays, oder wenn low-level-Datenstrukturen festgelegter Größe benutzt werden müssen. Achten Sie darauf, dass der theoretisch größte Wert, welcher in Ihrer Variable gespeichert wird, den größten möglichen Wert nicht überschreitet. Selbiges gilt natürlich auch für die Unterschreitung des kleinstmöglichen Wertes.

Ein Unter- oder Überlauf ist übrigens durchaus möglich. Die meisten Compiler bieten zwar eine Option an, um in einem solchem Fall einen Fehler zu erzeugen, aber diese Option ist standardmäßig nicht aktiv. Im folgenden kleinen Beispiel werden die Datentypen `short` (min: -32768, max: 32767) und `unsigned short` benutzt und bei beiden wird je ein Unter- und ein Überlauf ausgeführt:

```
#include <iostream>

int main(){
    short Variable1=15000;
    unsigned short Variable2=15000;

    std::cout << "short Variable:          " << Variable1 << std::endl
              << "unsigned short Variable: " << Variable2 << std::endl
              << "+30000\n\n";

    Variable1 += 30000;
    Variable2 += 30000;

    std::cout << "short Variable:          " << Variable1 << " (Überlauf)" << std::endl
              << "unsigned short Variable: " << Variable2 << std::endl
              << "+30000\n\n";
```

```
Variable1 += 30000;
Variable2 += 30000;

std::cout << "short Variable:          " << Variable1 << std::endl
          << "unsigned short Variable: " << Variable2 << " (Überlauf)" << std::endl
          << "-30000\n\n";

Variable1 -= 30000;
Variable2 -= 30000;

std::cout << "short Variable:          " << Variable1 << std::endl
          << "unsigned short Variable: " << Variable2 << " (Unterlauf)" << std::endl
          << "-30000\n\n";

Variable1 -= 30000;
Variable2 -= 30000;

std::cout << "short Variable:          " << Variable1 << " (Unterlauf)" << std::endl
          << "unsigned short Variable: " << Variable2 << std::endl;
}
```

Ausgabe

```
short Variable:          15000
unsigned short Variable: 15000
+30000

short Variable:          -20536 (Überlauf)
unsigned short Variable: 45000
+30000

short Variable:          9464
unsigned short Variable: 9464 (Überlauf)
-30000

short Variable:          -20536
unsigned short Variable: 45000 (Unterlauf)
-30000

short Variable:          15000 (Unterlauf)
unsigned short Variable: 15000
```

Verständlicher wird dieses Phänomen, wenn man die Zahlen binär (Duales Zahlensystem⁵) darstellt. Der Einfachheit halber beginnen wir mit der Darstellung der `unsigned short Variable`:

⁵ <http://de.wikipedia.org/wiki/Dualsystem>

```

Addition im Dualsystem mit unsigned short als Datentyp

Rechnung 1
  |0011101010011000| 15000
+  |0111010100110000| 30000
-----
Merker |111      11      |
-----
=  |1010111111001000| 45000

Rechnung 2
  |1010111111001000| 45000
+  |0111010100110000| 30000
-----
Merker 1|1111111      |
-----
= 1|0010010011111000| 9464

```

Die beiden Rechnungen weisen keinerlei Besonderheiten auf. Da nur die letzten 16 Ziffern beachtet werden (2 Byte = 16 Bit), entfällt in der zweiten Rechnung die 1 vor dem Vertikalstrich, wodurch das Ergebnis (in dezimaler Schreibweise) 9464 und nicht 75000 lautet.

```

Subtraktion im Dualsystem mit unsigned short als Datentyp

Rechnung 3
  |0010010011111000| 9464
-  |0111010100110000| 30000
-----
Merker...1|1111111      |
-----
=...1|1010111111001000| 45000

Rechnung 4
  |1010111111001000| 45000
-  |0111010100110000| 30000
-----
Merker      |111      11      |
-----
=  |0011101010011000| 15000

```

In diesem Fall ist die zweite Rechnung unauffällig. In der ersten Rechnung wird hingegen eine große Zahl von einer kleineren angezogen, was zu einem negativen Ergebnis führt oder besser führen würden, denn die Untergrenze ist in diesem Fall ja 0. Dort wo die 3 Punkte stehen, folgt eine unendliche Anzahl von Einsen. Dies ist keineswegs nur bei Dualzahlen der Fall, wenn Sie im dezimalen System eine große Zahl von einer kleineren nach den üblichen Regeln der schriftlichen Subtraktion abziehen, so erhalten Sie ein ähnliches Ergebnis:

```

      24
     - 31
     ---
Merker...1
     ---
     =...993

```

Die duale Darstellung der vier Rechnungen mit der `short`-Variable wird Ihnen sehr bekannt vorkommen:

```

Addition im Dualsystem mit short als Datentyp

Rechnung 1
  |0|011101010011000| 15000
+  |0|111010100110000| 30000
-----
Merker |1|11      11      |
-----
=  |1|010111111001000| -20536

Rechnung 2
  |1|010111111001000| -20536
+  |0|111010100110000| 30000
-----
Merker 1|1|111111      |
-----
= 1|0|010010011111000| 9464

Subtraktion im Dualsystem mit short als Datentyp

Rechnung 3
  |0|010010011111000| 9464
-  |0|111010100110000| 30000
-----
Merker...1|1|111111      |
-----
=...1|1|010111111001000| -20536

Rechnung 4
  |1|010111111001000| -20536
-  |0|111010100110000| 30000
-----
Merker |1|11      11      |
-----
=  |0|011101010011000| 15000

```

Die dualen Ziffern sind die gesamte Zeit über exakt die gleichen, der einzige Unterschied besteht darin, dass die erste Ziffer (welche durch einen weiteren Vertikalstrich abgegrenzt wurde) nun als Vorzeichenbit interpretiert wird (0 – Positiv, 1 – Negativ). Dies führt zu einer veränderten Darstellung im Dezimalsystem.

9.1.4 Gleitkommazahlen

Eine Gleitkommavariablen kann sich eine bestimmte Anzahl Ziffern merken und dazu die Position des Kommas. Das Wissen über den internen Aufbau einer solchen Zahl werden Sie wahrscheinlicher eher selten bis nie brauchen, daher sei an dieser Stelle auf den Wikipediaartikel über Gleitkommazahlen⁶ verwiesen. In C++ werden Sie Gleitkommazahlen/-variable für das Rechnen mit Kommazahlen verwenden. Es gibt drei Datentypen für Gleitkommazahlen, die in der folgenden Tabelle mit ihren üblichen Werten aufgelistet sind:

Typ	Speicherplatz	Wertebereich	kleinste positive Zahl	Genauigkeit
float	4 Byte	$\pm 3,4 \cdot 10^{38}$	$1,2 \cdot 10^{-38}$	6 Stellen
double	8 Byte	$\pm 1,7 \cdot 10^{308}$	$2,3 \cdot 10^{-308}$	15 Stellen

⁶ <http://de.wikipedia.org/wiki/Gleitkommazahl>

Typ	Speicherplatz	Wertebereich	kleinste positive Zahl	Genauigkeit
long double	10 Byte	$\pm 1,1 \cdot 10^{4932}$	$3,4 \cdot 10^{-4932}$	19 Stellen

Die Auswahl eines Gleitkommatentyps ist weniger einfach als die einer Ganzzahl. Wenn Sie nicht genau wissen, was Sie nehmen sollen, ist `double` in der Regel eine gute Wahl. Sobald Sie erst einmal ausreichend Erfahrung haben, wird es Ihnen leichter fallen abzuschätzen, ob `float` oder `long double` für Ihr Problem vielleicht eine bessere Wahl sind.

9.2 Variablen

Bevor eine Variable verwendet werden kann, muss sie dem Compiler bekannt gegeben werden. Dies bezeichnet man als Deklaration der Variable. Das eigentliche Anlegen einer Variable, so dass der Compiler Speicherplatz für Sie reserviert, wird Definition genannt. Eine Definition ist immer auch eine Deklaration und für Variablen ist der Unterschied zwischen Deklaration und Definition etwas kompliziert um ihn an dieser Stelle bereits zu erklären. Sie werden die Begriffe zunächst in Zusammenhang mit Funktionen kennenlernen und später für Variablen. Sie sollten sich jedoch jetzt bereits merken, dass es sich beim Anlegen der Variablen, die wir verwenden, immer um Definitionen handelt. Dies ist insofern wichtig, als dass eine Definition immer nur einmal geschrieben werden darf, während eine Deklaration beliebig oft vorgenommen werden kann. Um einen Vergleich zur realen Welt zu ziehen, wollen wir das Entstehen neuen Lebens betrachten. Sie können beliebig oft erzählen, dass ein bestimmtes Kind geboren wird, aber tatsächlich geschehen kann dies aber nur einmal. Im vorherigen Kapitel haben wir bereits mit Variablen vom Typ `int` gerechnet. Nun sollen Sie lernen, wie Variablen in C++ angelegt werden. Die allgemeine Syntax lautet:

```
<Datentyp> <Name>;
```

Außerdem ist es möglich, mehrere Variablen des gleichen Typs hintereinander anzulegen:

```
<Datentyp> <Variable1>, <Variable2>, <Variable3>;
```

Auch kann man einer Variable einen Anfangswert geben, dies bezeichnet man als Initialisierung. Es gibt zwei syntaktische Möglichkeiten (Schreibweisen) für Initialisierungen, welche anhand einer `int` Variable gezeigt werden soll:

```
int Zahl=100; // Möglichkeit 1
int Zahl(100); // Möglichkeit 2
```

Die erste Variante ist weit verbreitet, aber nicht zwingend besser. Bei den fundamentalen Datentypen von C++ spielt es keine Rolle welche Variante Sie verwenden, aber bei komplexeren Datentypen (Klassen) kann es zu Verwechslungen mit dem Zuweisungsoperator kommen, wenn Sie Möglichkeit 1 benutzen. Den genauen Unterschied zwischen einer Initialisierung und einer Zuweisung werden Sie kennen lernen, sobald es um Klassen geht. Für den Moment sollten Sie sich für eine der beiden

Variante entscheiden. Für Möglichkeit 1 spricht die große Verbreitung und die damit verbundene intuitive Nutzung. In diesem Buch werden wir diese erste Methode verwenden und in Zusammenhang mit Klassen auch eine Empfehlung geben, wann diese Methode zur besseren Übersicht im Quellcode beitragen kann. Für Möglichkeit 2 spricht hingegen, dass diese Syntax für Initialisierungen immer gültig ist. Nebenbei wird bei dieser Methode deutlich, dass es sich um eine Initialisierung handelt. Variablen mit Anfangswerten können natürlich auch hintereinander angelegt werden, sofern sie den gleichen Datentyp besitzen, allerdings ist davon aus Gründen der Übersichtlichkeit abzuraten.

```
int Zahl1(77), Zahl2, Zahl3=58; // Drei int-Variablen von denen zwei Anfangswerte haben
```

Wenn Sie einer Variable keinen Anfangswert geben, müssen Sie ihr später im Programm noch einen Wert zuweisen, bevor Sie mit ihr arbeiten (also damit rechnen oder den Inhalt ausgeben lassen). Weisen Sie einer solchen Variable keinen Wert zu und benutzen sie, so kann der Inhalt zufällig sein. Genaugenommen handelt es sich dann um die Bitfolge, die an der Stelle im Speicher stand, an der Ihre Variable angelegt wurde. Es gibt in C++ Regeln, in welchen Fällen der Compiler eine Variable ohne explizite Initialisierung implizit mit 0 initialisiert und wann stattdessen einfach der aktuelle Speicherinhalt stehen bleibt. Allerdings sind diese Regeln so kompliziert, dass es sich nicht lohnt sie sich zu merken, denn sollte je ein anderer Programmierer Ihren Code lesen, so muss auch dieser die Regeln kennen um den Code sofort verstehen zu können. Das nachfolgende kleine Programm zeigt einen Fall, in dem C++ besagt, dass keine implizite Initialisierung mit 0 stattfindet.

```
#include <iostream> // Ein-/Ausgabe

int main() {
    int Zahl; // Ganzzahlige Variable
    double Kommazahl1, Kommazahl2; // Gleitkommavariablen
    char Zeichen; // Zeichenvariable

    std::cout << "Zahl: " << Zahl << std::endl // Ausgabe der Werte
               << "Kommazahl1: " << Kommazahl1 << std::endl // welche jedoch
               << "Kommazahl2: " << Kommazahl2 << std::endl // nicht festgelegt
               << "Zeichen: " << Zeichen << std::endl; // wurden
}
```

Ausgabe

```
Zahl: -1211024315
Kommazahl1: 4.85875e-270
Kommazahl2: -3.32394e-39
Zeichen: f
```

Die Ausgabe kann bei jedem Ausführen des Programms anders lauten. Sollte dies bei Ihnen nicht der Fall sein, so stehen nur zufällig die gleichen Werte an der Stelle im Speicher welchen die jeweilige Variable belegt. Spätestens nach einem Neustart Ihres Rechners haben Sie höchstwahrscheinlich wieder eine andere Ausgabe. Variablen keinen Anfangswert zu geben ist beispielsweise sinnvoll, wenn Sie vorhaben über `std::cin` einen Wert in die Variable einzulesen. Dennoch würde es auch in diesem Fall keinen Schaden anrichten, wenn Sie die Variablen explizit mit 0 initialisieren.

```

#include <iostream> // Ein-/Ausgabe

int main(){
    int Zahl; // Ganzzahlige Variable
    double Kommazahl1, Kommazahl2; // Gleitkommavariablen
    char Zeichen; // Zeichenvariable

    std::cout << "Geben Sie bitte durch Leerzeichen getrennt eine Ganzzahl, zwei Kommazahlen "
        "und ein Zeichen ein:\n";

    std::cin >> Zahl // Eingabe von Werten
        >> Kommazahl1 // mit denen die vier
        >> Kommazahl2 // Variablen gefüllt
        >> Zeichen; // werden

    std::cout << "Zahl: " << Zahl << std::endl // Ausgabe der Werte
        << "Kommazahl1: " << Kommazahl1 << std::endl // welche zuvor
        << "Kommazahl2: " << Kommazahl2 << std::endl // eingegeben
        << "Zeichen: " << Zeichen << std::endl; // wurden
}

```

Ausgabe

```

Geben Sie bitte durch Leerzeichen getrennt eine Ganzzahl, zwei Kommazahlen und ein Zeichen ein:
Benutzereingabe: 6 8.4 6.0 g
Zahl: 6
Kommazahl1: 8.4
Kommazahl2: 6
Zeichen: g

```

9.3 Konstanten

Konstanten sind, wie schon oben beschrieben, Variablen, welche ihren Wert nicht verändern. Daraus folgt logisch, dass eine Konstante immer mit einem Anfangswert initialisiert werden muss, andernfalls hätten Sie eine Konstante mit einem zufälligen Wert und das ergibt keinen Sinn. Das Schlüsselwort, um eine Variable zu einer Konstante zu machen, ist `const`. Es gehört immer zu dem, was links davon steht, es sei denn, links steht nichts mehr, dann gehört es zu dem Teil auf der rechten Seite. Dies klingt zwar kompliziert, ist es aber eigentlich gar nicht. Für uns bedeutet es im Moment nur, dass Sie zwei Möglichkeiten haben, eine Variable zu einer Konstante zu machen:

```

const int Zahl(400); // Alternativ: const int Zahl=400;
// oder
int const Zahl(400); // Alternativ: int const Zahl=400;

```

Beides hat die gleiche Wirkung, wieder ist die erste Variante weit verbreitet und wieder ist die zweite Variante der besseren Lesbarkeit bei komplexeren Datentypen (Arrays von Zeigern Konstante auf Memberfunktionen. . .) vorzuziehen. Entscheiden Sie sich für die Variante, die Ihnen besser gefällt und verwenden Sie diese. Wichtig ist, dass Sie der Variante, für die Sie sich entscheiden, treu bleiben, wenigstens für die Dauer eines Projekts. Denn Code, in dem sich der Schreibstil ständig ändert, ist schwerer zu lesen, als alles andere.

9.4 Literale und ihre Datentypen

Ein Literal⁷ ist eine Zeichenkette, die zu Darstellung des Wertes einer der oben beschriebenen Datentypen dient.

```
#include <iostream>                // Ein-/Ausgabe

int main() {
    std::cout << 100 << std::endl // 100 ist ein int-Literal
               << 4.7 << std::endl // 4.7 ist ein double-Literal
               << 'h' << std::endl // 'h' ist ein char-Literal
               << true << std::endl; // true ist ein bool-Literal
}
```

Ausgabe

```
100
4.7
h
1
```

Bei der Ausgabe ist zu beachten, dass Boolean-Werte als 0 (*false*) bzw. 1 (*true*) ausgegeben werden.

Da die Bestimmung des Typs für einen ganzzahligen Wert etwas schwieriger ist als bei den übrigen, werden wir diese zuletzt behandeln. Bei Gleitkommalliteralen ist festgelegt, dass es sich um `double`-Werte handelt. Um einen Gleitkommalliteral mit einem anderen Typ zu erhalten, ist ein so genanntes Suffix nötig.

```
5.0 // double
6.7f // float
2.6F // float
9.4l // long double
4.0L // long double
```

Eine Zahl mit Komma (.) ist also ein `double`-Wert. Folgt der Zahl ein `f` oder ein `F` wird sie zu einem `float`-Wert und folgt ihr ein `l` oder ein `L` wird sie zu einem `long double`-Wert. Gleitpunktzahlen können auch in der wissenschaftlichen Schreibweise⁸ dargestellt werden.

⁷ <http://de.wikipedia.org/wiki/Literal>

⁸ <http://de.wikipedia.org/wiki/Wissenschaftliche%20Notation>

```
5.7e10
3.3e-3
8.7e876L
4.2e-4F
```

Wie die letzten beiden Beispiele zeigen, können auch hierbei die Suffixe für den Datentyp genutzt werden. Um ein Zeichen beziehungsweise eine Zeichenketten als `wchar_t` zu kennzeichnen, stellt man ihr ein `L` voran:

```
'a'           // char
L'b'         // wchar_t
"Ich bin ein Text" // char const*
L"Ich bin ein Text" // wchar_t const*
```

Was das Sternchen (*) hinter dem Datentyp im Kommentar bedeutet, werden Sie in einem späteren Kapitel erfahren. `bool` kann nur zwei Zustände annehmen, entsprechend gibt es auch nur zwei `bool`-Literale: `true` und `false`.

Nun zu den Ganzzahlen. Neben der dezimalen Darstellung von Zahlen gibt es in C++ auch die Möglichkeit der Oktalen⁹ und Hexadezimalen¹⁰ Darstellung. Um eine Zahl als Oktal zu kennzeichnen, wird ihr eine `0` (Null) vorangestellt, für eine Hexadezimalzahl wird `0x` (zu empfehlen, da deutlich besser lesbar) oder `0X` vorangestellt. Die Groß-/Kleinschreibung der hexadezimalen Ziffern spielt keine Rolle.

```
756           // Dezimal,      Dezimal: 756
046           // Oktal,       Dezimal: 38
0757          // Oktal,       Dezimal: 495
0xffff        // Hexadezimal, Dezimal: 65535
0x1234ABcd    // Hexadezimal, Dezimal: 305441741
```

Der Datentyp wird durch die Größe des Wertes bestimmt, wobei die folgende Reihenfolge gilt: `int`, `unsigned int`, `long`, `unsigned long`. Weiterhin kann jeder Ganzzahl das Suffix `u` oder `U` für `unsigned` und `l` oder `L` für `long` angehängt werden. Auch beide Suffixe gleichzeitig sind möglich. Die Reihenfolge ändert sich entsprechend den durch die Suffixe festgelegten Kriterien.

Das Wissen über die Datentypen von Literalen werden Sie wahrscheinlich eher selten benötigen, daher reicht es „mal etwas davon gehört zu haben“ und es, wenn nötig, nachzuschlagen.

⁹ <http://de.wikipedia.org/wiki/Oktalsystem>

¹⁰ <http://de.wikipedia.org/wiki/Hexadezimalsystem>

10 Rechnen mit unterschiedlichen Datentypen

Sie kennen nun die Datentypen in C++ und haben auch schon mit `int`-Variablen gerechnet. In diesem Kapitel erfahren Sie, wie man mit Variablen unterschiedlichen Typs rechnet. Es geht also weniger um das Ergebnis selbst, als viel mehr darum wie der Ergebnistyp lautet.

10.1 Ganzzahlen

Das Rechnen mit Ganzzahlen ist leicht zu begreifen. Die „kleinen“ Datentypen werden als `int` behandelt. Bei den größeren entscheidet der größte Datentyp über den Ergebnistyp. Die folgende Liste zeigt die Zusammenhänge:

```
char          + char          => int          | wchar_t      +
  char        => int
char          + wchar_t       => int          | wchar_t      +
  wchar_t     => int
char          + signed char    => int          | wchar_t      +
  signed char => int
char          + unsigned char  => int          | wchar_t      +
  unsigned char => int
char          + short          => int          | wchar_t      +
  short        => int
char          + unsigned short => int          | wchar_t      +
  unsigned short => int
char          + int            => int          | wchar_t      +
  int          => int
char          + unsigned int   => unsigned int | wchar_t      +
  unsigned int => unsigned int
char          + long           => long          | wchar_t      +
  long         => long
char          + unsigned long  => unsigned long | wchar_t      +
  unsigned long => unsigned long

signed char   + char          => int          | unsigned char +
  char        => int
signed char   + wchar_t       => int          | unsigned char +
  wchar_t     => int
signed char   + signed char    => int          | unsigned char +
  signed char => int
signed char   + unsigned char  => int          | unsigned char +
  unsigned char => int
signed char   + short          => int          | unsigned char +
  short        => int
signed char   + unsigned short => int          | unsigned char +
  unsigned short => int
signed char   + int            => int          | unsigned char +
  int          => int
signed char   + unsigned int   => unsigned int | unsigned char +
  unsigned int => unsigned int
signed char   + long           => long          | unsigned char +
  long         => long
signed char   + unsigned long  => unsigned long | unsigned char +
```

```

unsigned long => unsigned long

short      + char      => int      | unsigned short +
char      => int
short      + wchar_t   => int      | unsigned short +
wchar_t   => int
short      + signed char => int      | unsigned short +
signed char => int
short      + unsigned char => int    | unsigned short +
unsigned char => int
short      + short     => int      | unsigned short +
short     => int
short      + unsigned short => int   | unsigned short +
unsigned short => int
short      + int       => int      | unsigned short +
int       => int
short      + unsigned int => unsigned int | unsigned short +
unsigned int => unsigned int
short      + long      => long     | unsigned short +
long      => long
short      + unsigned long => unsigned long | unsigned short +
unsigned long => unsigned long

int        + char      => int      | unsigned int  +
char      => unsigned int
int        + wchar_t   => int      | unsigned int  +
wchar_t   => unsigned int
int        + signed char => int     | unsigned int  +
signed char => unsigned int
int        + unsigned char => int    | unsigned int  +
unsigned char => unsigned int
int        + short     => int      | unsigned int  +
short     => unsigned int
int        + unsigned short => int   | unsigned int  +
unsigned short => unsigned int
int        + int       => int      | unsigned int  +
int       => unsigned int
int        + unsigned int => unsigned int | unsigned int  +
unsigned int => unsigned int
int        + long      => long     | unsigned int  +
long      => long oder unsigned long
int        + unsigned long => unsigned long | unsigned int  +
unsigned long => unsigned long

long       + char      => long     | unsigned long +
char      => unsigned long
long       + wchar_t   => long     | unsigned long +
wchar_t   => unsigned long
long       + signed char => long    | unsigned long +
signed char => unsigned long
long       + unsigned char => long   | unsigned long +
unsigned char => unsigned long
long       + short     => long     | unsigned long +
short     => unsigned long
long       + unsigned short => long  | unsigned long +
unsigned short => unsigned long
long       + int       => long     | unsigned long +
int       => unsigned long
long       + unsigned int => unsigned long | unsigned long +
unsigned int => unsigned long
long       + long      => long     | unsigned long +
long      => unsigned long
long       + unsigned long => unsigned long | unsigned long +
unsigned long => unsigned long

```

Zugegebenermaßen wirkt dies erst einmal erschlagend, aber es ist eigentlich nicht schwer zu begreifen. Bei jeder Rechenoperation hat jeder der 2 Operanden, sowie das Ergebnis der Rechnung einen Datentyp:

```
#include <iostream>

int main(){
    char  Zahl1=22;
    short Zahl2=40;

    std::cout << Zahl1 * Zahl2 << std::endl; // 22 * 40 = 880
                                           // char + short => int
}
```

10.2 Gleitkommarechnen

Beim Rechnen mit Gleitkommazahlen gelten im Grunde die gleichen Regeln wie bei Ganzzahlen. Der Ergebnistyp entspricht auch hier dem des Operanden mit dem „größeren“ Typ. Die Reihenfolge lautet: float, double, long double. Es gilt also:

```
float    + float    => float
float    + double   => double
float    + long double => long double

double   + float    => double
double   + double   => double
double   + long double => long double

long double + float    => long double
long double + double   => long double
long double + long double => long double
```

10.3 Casting

Casting bedeutet in diesem Zusammenhang, die Umwandlung eines Datentyps in einen anderen. Diese Typumwandlung kann sowohl automatisch (implizit) stattfinden, als auch vom Programmierer angegeben (explizit) werden.

10.3.1 Implizite Typumwandlung

Mit impliziter Typumwandlung hatten Sie bereits reichlich zu tun, denn es kann ausschließlich mit Zahlen gerechnet werden die den gleichen Typ besitzen.

Beispiele:

```
char + int          => int          | int          + int          =>
```

```
int
short + unsigned int => unsigned int | unsigned int + unsigned int =>
unsigned int
float + double      => double      | double      + double      =>
double
```

Umformungsregeln

Viele binäre Operatoren, die arithmetische oder Aufzählungsoperanden erwarten, verursachen Umwandlungen und ergeben Ergebnistypen auf ähnliche Weise. Der Zweck ist, einen gemeinsamen Typ zu finden, der auch der Ergebnistyp ist. Dieses Muster wird "die üblichen arithmetischen Umwandlungen" genannt, die folgendermaßen definiert sind:

- Wenn ein Operand vom Typ `long double` ist, dann wird der andere zu `long double` konvertiert.
- Andernfalls, wenn ein Operand vom Typ `double` ist, dann wird der andere zu `double` konvertiert.
- Andernfalls, wenn ein Operand vom Typ `float` ist, dann wird der andere zu `float` konvertiert.
- Ansonsten werden die integralen Umwandlungen auf beide Operanden angewendet:
- Wenn ein Operand vom Typ `unsigned long` ist, dann wird der andere zu `unsigned long` konvertiert.
- Andernfalls, wenn ein Operand vom Typ `long` und der andere vom Typ `unsigned int`, dann wird, falls ein `long` alle Werte eines `unsigned int` darstellen kann, der `unsigned int`-Operand zu `long` konvertiert; andernfalls werden beide Operanden zu `unsigned long` konvertiert.
- Andernfalls, wenn ein Operand vom Typ `long` ist, dann wird der andere zu `long` konvertiert.
- Andernfalls, wenn ein Operand vom Typ `unsigned int` ist, dann wird der andere zu `unsigned int` konvertiert.

Hinweis: Der einzig verbleibende Fall ist, dass beide Operanden vom Typ `int` sind.

Diese Regeln wurden so aufgestellt, dass dabei stets ein Datentyp in einen anderen Datentyp mit "größerem" Wertebereich umgewandelt wird. Das stellt sicher, dass bei der Typumwandlung keine Wertverluste durch Überläufe entstehen. Es können allerdings bei der Umwandlung von Ganzzahlen in `float`-Werte Rundungsfehler auftreten:

```
std::cout << "17000000 + 1.0f = " << std::fixed << 17000000 + 1.0f << std::endl;
```

Ausgabe

```
17000000 + 1.0f = 17000000.000000
```

Für die Berechnung werden zunächst beide Operanden in den Datentyp `float` konvertiert und anschließend addiert. Das Ergebnis ist wiederum ein `float` und somit aber nicht in der Lage, Zahlen in der Größenordnung von 17 Millionen mit der nötigen Genauigkeit zu speichern, um zwischen 17000000 und 17000001 zu unterscheiden. Das Ergebnis der Addition ist daher wieder 17000000.

10.3.2 Explizite Typumwandlung

In C++ gibt es dafür zwei Möglichkeiten. Zum einen den aus C übernommenen Cast (Typ) Wert und zum anderen die vier (neuen) C++ Casts.

```
static_cast< Zieltyp >(Variable)
const_cast< Zieltyp >(Variable)
dynamic_cast< Zieltyp >(Variable)
reinterpret_cast< Zieltyp >(Variable)
```

Tip

Die Leerzeichen zwischen dem Zieltyp und den spitzen Klammern sind nicht zwingend erforderlich, Sie sollten sich diese Notation jedoch angewöhnen. Speziell wenn Sie später mit Templates oder Namensräumen arbeiten, ist es nützlich Datentypen ein wenig von ihrer Umgebung zu isolieren. Sie werden an den entsprechenden Stellen noch auf die ansonsten möglichen Doppeldeutigkeiten hingewiesen.

Thema wird später näher erläutert...

Im Moment benötigen Sie nur den `static_cast`. Was genau die Unterschiede zwischen diesen Casts sind und wann man welchen einsetzt erfahren Sie Kapitel Casts^a. Auf C++-Programmierung/ Nützliches/ Casts#Gefährliche C-Casts^b wird in diesem Kapitel ebenfalls eingegangen, merken Sie sich jedoch schon jetzt, dass Sie diese nicht einsetzen sollten. Natürlich müssen Sie sie als C++-Programmierer dennoch kennen, falls Sie einmal auf einen solchen stoßen sollten.

- ^a <http://de.wikibooks.org/wiki/C%2B%2B-Programmierung%2F%20N%C3%9Ctzliches%2F%20Casts>
- ^b <http://de.wikibooks.org/wiki/C%2B%2B-Programmierung%2F%20N%C3%9Ctzliches%2F%20Casts%23Gef%C4%9Crliche%20C-Casts>

10.4 Ganzzahlen und Gleitkommazahlen

Wird mit einer Ganzzahl und einer Gleitkommazahl gerechnet, so ist das Ergebnis vom gleichen Typ wie die Gleitkommazahl.

10.5 Rechnen mit Zeichen

Mit Zeichen zu rechnen ist besonders praktisch. Um beispielsweise das gesamte Alphabet auszugeben zählen Sie einfach vom Buchstaben 'A' bis einschließlich 'Z':

```
#include <iostream>

int main(){
```

```
for(char i = 'A'; i <= 'Z'; ++i){
    std::cout << i;
}
}
```

Ausgabe

ABCDEFGHIJKLMNOPQRSTUVWXYZ

Für eine Erklärung des obigen Quellcodes, lesen Sie bitte das Kapitel Schleifen¹.

Wenn Sie binäre Operatoren auf Zeichen anwenden, ist das Ergebnis (mindestens) vom Typ `int`. Im folgenden Beispiel wird statt eines Buchstabens, der dazugehörige ASCII²-Wert ausgegeben. Um also wieder ein Zeichen auszugeben, müssen Sie das Ergebnis wieder in den Zeichentyp `casten`. (Beachten Sie im folgenden Beispiel, dass die Variable `i` - im Gegensatz zum vorherigen Beispiel - nicht vom Typ `char` ist):

```
#include <iostream>

int main(){
    char zeichen = 'A';

    for(int i = 0; i < 26; ++i){
        std::cout << zeichen + i << ' ';           // Ergebnis int
    }

    std::cout << std::endl;

    for(int i = 0; i < 26; ++i){
        std::cout << static_cast< char >(zeichen + i); // Ergebnis char
    }
}
```

Ausgabe

65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
ABCDEFGHIJKLMNOPQRSTUVWXYZ

¹ Kapitel 12 auf Seite 61

² <http://de.wikipedia.org/wiki/ASCII-Tabelle>

11 Verzweigungen

Eine Verzweigung (bedingte Anweisung, *conditional statement*) dient dazu, ein Programm in mehrere Pfade aufzuteilen. Beispielsweise kann so auf Eingaben des Benutzers reagiert werden. Je nachdem, was der Benutzer eingibt, ändert sich der Programmablauf.

11.1 Falls

Verzweigungen werden mit dem Schlüsselwort `if` begonnen. In der einfachsten Form sieht das so aus:

```
if («Bedingung») «Anweisung»
```

Wenn die Bedingung erfüllt ist, wird die Anweisung ausgeführt, ansonsten wird sie übersprungen. Sollen nicht nur eine, sondern mehrere Anweisungen ausgeführt werden, fassen Sie diese mit `{ . . . }` zu einer Blockanweisung zusammen:

```
if («Bedingung»){  
    «Anweisungen»  
}
```

Als Bedingung darf jeder Ausdruck verwendet werden, der einen `bool` zurückgibt oder dessen Ergebnis sich in einen `bool` umwandeln lässt. Ganzzahlige und Gleitkommatentypen lassen sich nach `bool` umwandeln, die Regel lautet einfach: Ist eine Zahl gleich 0, so wird sie als `false` ausgewertet, andernfalls als `true`.

```
int i;  
cin >> i;  
if(i){  
    cout << "Der Benutzer hat einen Wert ungleich 0 eingegeben\n";  
}
```

11.2 Andernfalls

Das Schlüsselwort `else` erweitert die Einsatzmöglichkeiten der Verzweigung. Während ein normales (also einzelnes) `if` einen bestimmten Teil des Codes ausführt, falls eine Bedingung erfüllt ist, stellt `else` eine Erweiterung dar, anderen Code auszuführen, falls die Bedingung nicht erfüllt ist.

```
int i;
cin >> i;
if (i)
    cout << "Sie haben einen Wert ungleich 0 eingegeben!\n";
else
    cout << "Sie haben 0 eingegeben!\n";
```

Natürlich könnten auch hier, sowohl für die `if`-Anweisung, als auch für die `else`-Anweisung, ein Anweisungsblock stehen. Wenn Sie Pascal oder eine ähnliche Programmiersprache kennen, wird Ihnen auffallen, dass auch die Anweisung vor dem `else` mit einem Semikolon abgeschlossen wird. Da auf eine `if`- oder `else`-Anweisung immer nur eine Anweisung *oder* ein Anweisungsblock stehen kann, muss zwangsläufig direkt danach ein `else` stehen, um dem `if` zugeordnet zu werden.

Sie können in einer Verzweigungsanweisung auch mehr als zwei Alternativen angeben:

```
int i;
cin >> i;
if (i == 10)
    cout << "Sie haben zehn eingegeben\n";
else
    if (i == 11)
        cout << "Sie haben elf eingegeben\n";
    else
        cout << "Sie haben weder zehn noch elf eingegeben\n";
```

Es können beliebig viele Zweige mit `else if` vorkommen. Allerdings ist es üblich, eine andere Einrückung zu wählen, wenn solche „`if-else`-Bäume“ ausgebaut werden:

```
int i;
cin >> i;
if (i == 10)
    cout << "Sie haben zehn eingegeben\n";
else if (i == 11)
    cout << "Sie haben elf eingegeben\n";
else
    cout << "Sie haben weder zehn noch elf eingegeben\n";
```

Außerdem ist es zu empfehlen, auch bei einer Anweisung einen Anweisungsblock zu benutzen. Letztlich ist die Funktionalität immer die gleiche, aber solche Blöcke erhöhen die Übersichtlichkeit

und wenn Sie später mehrere Anweisungen, statt nur einer angeben möchten, brauchen Sie sich um etwaige Klammern keine Gedanken zu machen, weil sie sowieso schon vorhanden sind.

```
int i;
cin >> i;
if (i == 10) {
    cout << "Sie haben zehn eingegeben\n";
} else if(i == 11) {
    cout << "Sie haben elf eingegeben\n";
} else {
    cout << "Sie haben weder zehn noch elf eingegeben\n";
}
```

Sie werden für die Positionierung der Klammern übrigens auch oft auf eine andere Variante treffen:

```
int i;
cin >> i;
if (i == 10)
{
    cout << "Sie haben zehn eingegeben\n";
}
else
{
    if (i == 11)
    {
        cout << "Sie haben elf eingegeben\n";
    }
    else
    {
        cout << "Sie haben weder zehn noch elf eingegeben\n";
    }
}
}
```

Einige Programmierer finden dies übersichtlicher, für dieses Buch wurde jedoch die Variante mit den öffnenden Klammern ohne Extrazeile zu verwenden. Das hat den Vorteil, dass weniger Platz benötigt wird und da die Einrückung ohnehin die Zugehörigkeit andeutet, ist eine zusätzliche Kennzeichnung nicht unbedingt nötig.

Die Einrückung von Quelltextzeilen hat für den Compiler übrigens keine Bedeutung. Sie ist lediglich eine grafische Darstellungshilfe für den Programmierer. Auch die in diesem Buch gewählte Einrückungstiefe von vier Leerzeichen ist optional, viele Programmierer verwenden etwa nur zwei Leerzeichen. Andere hingegen sind davon überzeugt, dass acht die ideale Wahl ist. Aber egal, wofür Sie sich entscheiden, wichtig ist, dass Sie Ihren Stil einhalten und nicht ständig ihren Stil wechseln. Das verwirrt nicht nur, sondern sieht auch nicht schön aus.

Tip

Wenn es Sie nicht stört, in Ihrem Texteditor Tabulatorzeichen und Leerzeichen anzeigen zu lassen, dann sollten Sie für die Einrückung Tabulatorzeichen verwenden und für alles hinter der normalen Einrückung (etwa den Abstand bis zum Kommentar) Leerzeichen. Das hat den Vorteil, dass Sie die Einrückungstiefe jederzeit ändern können, indem Sie angeben wie viele Leerzeichen einem Tabulatorzeichen entsprechen.

11.3 Vergleichsoperatoren

Im obigen Beispiel kam schon der Vergleichsoperator `==` zum Einsatz. In C++ gibt es insgesamt sechs Vergleichsoperatoren. Sie liefern jeweils den Wert `true`, wenn die beiden Operanden (die links und rechts des Operators stehen) dem Vergleichskriterium genügen, ansonsten den Wert `false`.

<code>==</code>	identisch
<code><=</code>	ist kleiner (oder) gleich
<code>>=</code>	ist größer (oder) gleich
<code><</code>	ist kleiner
<code>></code>	ist größer
<code>!=</code>	ist ungleich

Hinweis

Der Vergleichsoperator `==` wird von Anfängern oft mit dem Zuweisungsoperator `=` verwechselt. Da es absolut legal ist, eine Zuweisung innerhalb einer `if`-Bedingung zu machen, führt das oft zu schwer zu findenden Fehlern:

```
int a = 5, b = 8;
if (a = b) cout << "5 ist gleich 8.";
```

Ausgabe

```
5 ist gleich 8.
```

Das weist `b` an `a` (`a = 8`) zu und wertet dann die Rückgabe von `a` (also `8`) zu `true` aus. Prüfen Sie bei seltsamen Verhalten also immer ob vielleicht der Zuweisungsoperator `=` statt des Gleichheitsoperators `==` verwendet wurde.

Eine weitere Falle ist der Ungleichheitsoperator `!=`, wenn er falsch herum geschrieben wird (`=!`). Letzteres sind in Wahrheit zwei Operatoren, nämlich die Zuweisung `=` und die logische Negierung `!`, die Sie gleich kennen lernen werden. Um das zu unterscheiden, machen Sie sich einfach klar, was das in Worten heißt:

- `!=` – nicht gleich
- `=!` – gleich nicht

11.4 Logische Operatoren

Mit logischen Operatoren können Sie mehrere Bedingungen zu einem Ausdruck verknüpfen. C++ bietet folgende Möglichkeiten:

<code>!</code>	Logisches Nicht	Resultat wahr, wenn der Operand falsch ist
<code>&&</code>	Logisches Und	Resultat wahr, wenn beide Operanden wahr sind
<code> </code>	Logisches Oder	Resultat wahr, wenn mindestens ein Operand wahr ist (<i>inclusive-or</i>)

Bei den Operatoren `&&` (Logik-und) und `||` (Logik-oder) werden die Teilausdrücke *von links nach rechts* bewertet, und zwar nur so lange, bis das Resultat feststeht. Wenn z. B. bei einer `&&`-Verknüpfung schon die erste Bedingung falsch ist, wird die zweite nicht mehr untersucht, da beide Bedingungen `true` sein müssen. Der Rückgabewert dieser Operatoren ist genau wie bei den Vergleichsoperatoren von Typ `bool`.

```
int i = 10, j = 20;
if (i == 10 && j == 10) {
    cout << "Beide Werte sind gleich zehn\n";
}
if (i == 10 || j == 10) {
    cout << "Ein Wert oder beide Werte sind gleich zehn\n";
}
```

```
}
```

Ausgabe

Ein Wert oder beide Werte sind gleich zehn

Aus Gründen der Lesbarkeit sollten Vergleichsausdrücke grundsätzlich von Klammern umgeben sein. Der obige Code würde folglich so aussehen:

```
int i = 10, j = 20;
if ((i == 10) && (j == 10)) {
    cout << "Beide Werte sind gleich zehn\n";
}
if ((i == 10) || (j == 10)) {
    cout << "Ein Wert oder beide Werte sind gleich zehn\n";
}
```

Ausgabe

Ein Wert oder beide Werte sind gleich zehn

Hinweis

Beachten Sie bitte, dass der UND-Operator (&&) eine höhere Priorität als der ODER-Operator (||) hat. Das heißt, Sie müssen bei Ausdrücken wie dem Folgenden vorsichtig sein.

```
int i = 10, j = 20;
// Erwartete Reihenfolge      (((i == 10) || (j == 20)) && (j == 20)) && (i == 5))
// Tatsächliche Reihenfolge ((i == 10) || ((j == 20) && (j == 20)) && (i == 5))
if (i == 10 || j == 20 && j == 20 && i == 5) {
    cout << "i ist Zehn und fünf oder (j ist Zwanzig und i fünf)!\n";
} else {
    cout << "i ist nicht Zehn oder (j ist Zwanzig oder i nicht fünf)!\n";
}
```

Ausgabe

```
i ist Zehn und fünf oder (j ist Zwanzig und i fünf)!
```

Die Ausgabe ist von der Logik her falsch, weil der Ausdruck in der Reihenfolge `((i == 10) || ((j == 20) && (j == 20)) && (i == 5))` ausgewertet wird. Solche Fehler sind sehr schwer zu finden, also sollten Sie sie auch nicht machen. Daher der Tipp: Verwenden Sie bei solch komplexen Bedingungen *immer* Klammern, um klar zu machen, in welcher Reihenfolge Sie die Ausdrücke auswerten wollen. Das ist unmissverständlich, und der menschliche Leser liest den Ausdruck genauso wie der Compiler. Um zu erkennen, an welcher Stelle eine Klammer wieder geschlossen wird, beherrschen die meisten Editoren das sogenannte Bracket Matching. Dabei hebt der Editor (automatisch oder über einen bestimmten Hotkey) die schließende Klammer hervor.

Die Operatoren lassen sich übersichtlich mit Wahrheitstafeln beschreiben:

Logisches Und				
a	true	true	false	false
b	true	false	true	false
a && b	true	false	false	false

Logisches Oder				
a	true	true	false	false
b	true	false	true	false
a b	true	true	true	false

Logisches Nicht		
a	true	false
!a	false	true

Tip

Wenn Sie mit mehreren `&&` und `||` arbeiten, dann schreiben Sie den Ausdruck, der am wahrscheinlichsten zutrifft, auch am weitesten links, also **vor** den anderen Ausdrücken. Das ist eine (zugegebenermaßen) sehr geringfügige Optimierung, aber es gibt Situationen in denen sie trotzdem sinnvoll ist. Beispielsweise wenn die Bedingung innerhalb einer Schleife^a sehr oft ausgeführt wird.

Hinweis für fortgeschrittene Leser: Beachten Sie bitte, dass für überladene Operatoren andere Regeln gelten. Alle diejenigen, die noch nicht wissen, was überladene Operatoren sind, brauchen sich um diesen Hinweis (noch) nicht zu kümmern.

^a Kapitel 12 auf Seite 61

11.5 Bedingter Ausdruck

Häufig werden Verzweigungen eingesetzt, um abhängig vom Wert eines Ausdrucks eine Zuweisung vorzunehmen. Das können Sie mit dem Auswahloperator `? ... :` auch einfacher formulieren:

```
min = a < b ? a : b;  
// Alternativ ginge:  
if (a < b) {  
    min = a;  
} else {  
    min = b;  
}
```

Grafisch sieht das so aus:

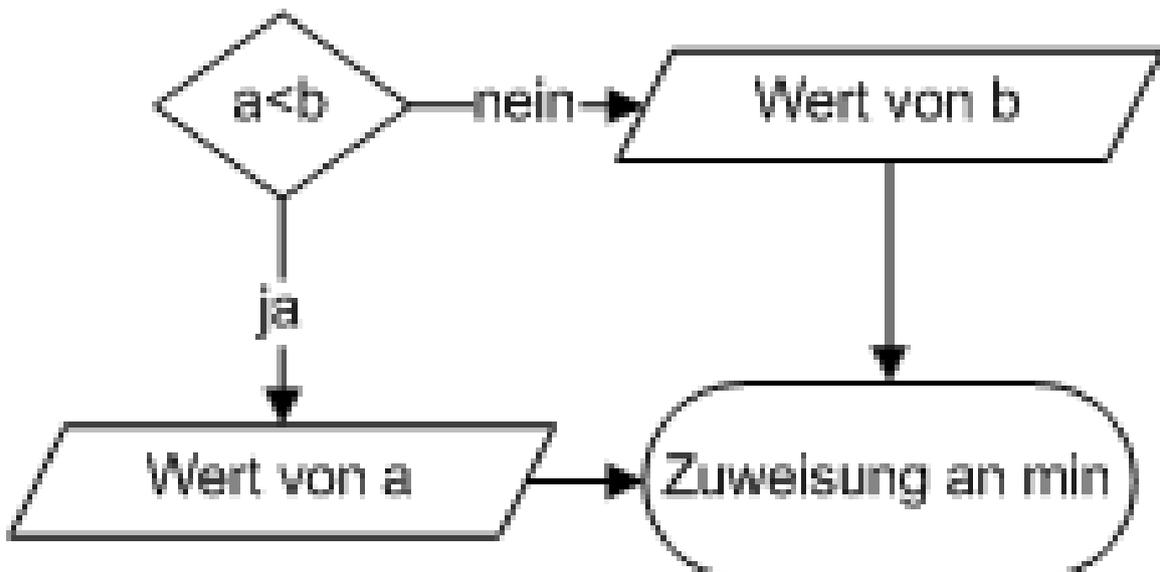


Abb. 1

Der Variablen `min` wird der kleinere, der beiden Werte `a` und `b` zugewiesen. Analog zum Verhalten der logischen Operatoren wird nur derjenige „Zweig“ bewertet, der nach Auswertung der Bedingung (`a < b`) tatsächlich ausgeführt wird.

12 Schleifen

Mit dem, was Sie bis jetzt gelernt haben, sollte es für Sie eine leichte Übung sein, die Zahlen von eins bis zehn ausgeben zu lassen. So könnte ein Programm aussehen, das dies tut:

```
#include <iostream>

using namespace std;

int main() {
    cout << "1\n";
    cout << "2\n";
    cout << "3\n";
    cout << "4\n";
    cout << "5\n";
    cout << "6\n";
    cout << "7\n";
    cout << "8\n";
    cout << "9\n";
    cout << "10\n";
}
```

Ausgabe

```
1
2
3
4
5
6
7
8
9
10
```

Dieses Programm ist einfach – aber was wäre, wenn die Zahlen eins bis einer Million ausgegeben werden sollen? Oder schlimmer noch - Ja, es geht noch schlimmer - die Ausgabe hängt von der Benutzereingabe ab. Dann müssten Sie von der größtmöglichen Zahl ausgehen (bei `unsigned int` üblicherweise 4.294.967.295) und auch noch nach jeder Eingabe überprüfen, ob die vom Benutzer eingegebene Zahl erreicht ist.

```
#include <iostream>
```

```
using namespace std;

int main() {
    unsigned int i=1, benutzer; // 2 Variablen anlegen

    cin >> benutzer;           // Benutzer gibt Zahl ein

    if (i <= benutzer) {       // Benutzereingabe erreicht?
        cout << "1\n";         // Ausgabe der Zahl 1
        ++i;                   // Ausgegebene Zahlen mitzählen
    } else return 0;           // Anwendung beenden
    if (i <= benutzer) {       // Benutzereingabe erreicht?
        cout << "2\n";         // Ausgabe der Zahl 2
        ++i;                   // Ausgegebene Zahlen mitzählen
    } else return 0;           // Anwendung beenden
    // ...
    if (i <= benutzer) {       // Benutzereingabe erreicht?
        cout << "4294967295\n"; // Ausgabe der Zahl 4294967295
        ++i;                   // Ausgegebene Zahlen mitzählen
    } else return 0;           // Anwendung beenden

    // Wenn Ihr Compiler diese Stelle erreichen soll, brauchen Sie einen leistungsstarken
    // Rechner und ein robustes Betriebssystem.
    // Um dieses Problem zu lösen, setzen Sie sich einfach in die nächste Zeitmaschine
    // und bestellen Sie sich einen Rechner aus dem Jahr 2020!
}
```

Es sei Ihnen überlassen, den fehlenden Code von Hand einzutragen und wenn Sie fertig sind können Sie mit Stolz behaupten, dass Sie ein simples Problem mit einer Anzahl von über 17,1 *Milliarden* Zeilen Code gelöst haben. Die GCC¹ hat gerade einmal etwas mehr als 2,1 *Millionen* Zeilen Code, aber da geht es ja auch nur darum, Compiler zu schreiben und nicht Zahlen auszugeben. Trotzdem, die Unterschiede zwischen dem Compilerbau-Projekt und unserem hoch komplexen Programm zum Ausgeben einer durch den Benutzer angegebenen Menge von Zahlen ist dermaßen horrend, dass es da doch eigentlich noch einen Trick geben muss.

Und tatsächlich, denn an diesem Punkt kommen Schleifen ins Spiel. Bis jetzt wurden alle Programme einfach der Reihe nach abgearbeitet und zwischendurch wurde eventuell mal eine Abzweigung genommen. Mit einer Schleife können Sie erreichen, dass ein Programmteil mehrfach abgearbeitet wird. C++ stellt 3 Schleifenkonstrukte zur Verfügung. Die kopfgesteuerte `while`-Schleife, die fußgesteuerte `do-while`-Schleife und die ziemlich universelle (ebenfalls kopfgesteuerte) `for`-Schleife. Was kopf- und fußgesteuerte Schleife bedeutet erfahren Sie in Kürze. Vielleicht wissen Sie es aber auch schon aus dem Kapitel „Grundlegende Elemente“, aus dem Abschnitt „Für Programmieranfänger“.

¹ <http://de.wikipedia.org/wiki/GNU%20Compiler%20Collection>

12.1 Die while-Schleife

Eine while-Schleife hat die folgende allgemeine Form:

```
while («Bedingung») «Anweisung»
```

Natürlich können Sie, wie bei den Verzweigungen auch, hier wieder mehrere Anweisungen zu einem Anweisungsblock zusammenfassen, da diese als eine Anweisung gilt:

```
while («Bedingung») {
    «Anweisungen»
}
```

Solange die Bedingung erfüllt ist, wird die Anweisung oder der Anweisungsblock ausgeführt. Da es sich hier um eine kopfgesteuerte Schleife handelt, wird erst die Bedingung ausgewertet. Ist diese erfüllt, so wird die Anweisung ausgeführt und dann erneut die Bedingung überprüft. Ist die Bedingung nicht erfüllt, wird der Schleifeninhalt kurzerhand übersprungen und mit dem Quelltext nach der Schleife fortgesetzt. Was eine fußgesteuerte Schleife macht, erfahren Sie unter der Überschrift `do-while`. Eine Gegenüberstellung der beiden Schleifen gibt es in der Zusammenfassung dieses Kapitels.

Tip

Wie die Bedingungen ausgewertet werden, können Sie im Kapitel „Verzweigungen“^a nachlesen. Einige von Ihnen werden sich vielleicht noch daran erinnern, dass dies das vorherige Kapitel war, aber für die Gedächtnisschwachen unter Ihnen und jene, die es einfach nicht lassen können in der Mitte mit dem Lesen zu beginnen, steht es hier noch mal Schwarz auf Grün.

^a Kapitel 11 auf Seite 51

Hinweis

Beachten Sie, dass Schleifen so lange ausgeführt werden, bis die Bedingung nicht mehr erfüllt ist. Wenn Sie also nicht innerhalb der Schleife dafür sorgen, dass die Bedingung irgendwann nicht mehr erfüllt ist, dann haben Sie eine sogenannte *Endlosschleife*. Das heißt, der *Schleifenrumpf* (so nennt man die Anweisung oder den Anweisungsblock einer Schleife) wird immer wieder ausgeführt. Das Programm wird also nie beendet. In Kombination mit einem Schlüsselwort, das Sie in Kürze kennen lernen werden, kann eine solche Endlosschleife durchaus gewollt und sinnvoll sein, aber in der Regel entsteht so etwas versehentlich. Wenn Ihr Programm also mal „abgestürzt“ ist, im Sinne von „Es reagiert nicht mehr! Wie schrecklich!“, dann haben Sie fast todsicher irgendwo eine Endlosschleife fabriziert.

Nun aber zurück zu unserer Schleifenaufgabe mit 17,1 Milliarden Zeilen Code. Da Ihr Compiler immer noch nicht damit fertig ist, die oben vorgestellte Lösung zu übersetzen, versuchen wir jetzt mal das ganze mit einer while-Schleife zu lösen.

```
#include <iostream>
```

```
using namespace std;

int main() {
    unsigned int i=1, benutzer; // 2 Variablen anlegen

    cin >> benutzer;           // Benutzer gibt Zahl ein

    while (i <= benutzer) {    // Benutzereingabe erreicht?
        cout << i << endl;     // Ausgabe von i und einem Zeilenumbruch
        i++;                   // i um eins erhöhen (-> ausgegebene Zahlen mitzählen)
    }
}
```

Nun ja, das sieht dem Programm von oben doch irgendwie ähnlich, nur die knapp 4,3 Milliarden `if`-Anweisungen sind weggefallen und haben einer vom Aufbau fast identischen `while`-Schleife platz gemacht. Nun haben wir natürlich ein Problem: Ihr Superrechner aus dem Jahr 2020 ist immer noch mit der Übersetzung der ersten Programmversion beschäftigt und wird es wohl auch noch bis zu seiner Erfindung bleiben. Aber zum Glück haben Sie ja noch einen alten Rechner von 1980. Also versuchen Sie das Programm auf diesem zu übersetzen und auszuführen. Tatsächlich. Es funktioniert. Erstaunlich, dass so eine alte Kiste einen Rechner überholt, den Sie sich extra aus der Zukunft haben liefern lassen, um die maximale Leistungsstärke zu bekommen.

Wenn Sie Schwierigkeiten haben das Beispiel nachzuvollziehen, dann sehen Sie sich noch mal die Schleifen-Version für die Zahlen von 1 bis 10 an.

```
#include <iostream>

using namespace std;

int main() {
    int i = 1;                 // Anlegen von i

    while (i <= 10) {         // Ist i noch kleiner-gleich 10?
        cout << i << endl;     // Ausgabe von i und neue Zeile
        i++;                   // i um eins erhöhen
    }
}
```

Ausgabe

```
1
2
3
4
5
6
7
8
```

9
10

So sieht das Ganze schon viel kompakter und vielleicht auch übersichtlicher aus als die Version von ganz oben. Die Ausgabe dagegen ist völlig identisch. Hier wird `i` am Anfang auf eins gesetzt. Somit ist die Bedingung (eins ist kleiner oder gleich zehn) erfüllt. Damit wird der Schleifenrumpf ausgeführt. „1“ wird ausgegeben. Es folgt ein Zeilenumbruch. Danach wird der Wert von `i` um eins erhöht. Danach wird wieder geprüft, ob die Bedingung erfüllt ist. Da `i` jetzt zwei ist, lautet die Bedingung „zwei ist kleiner oder gleich zehn“, da dies eine wahre Aussage ist, wird wieder der Schleifenrumpf ausgeführt. Das wiederholt sich bis `i` schließlich den Wert elf hat. Die Bedingung lautet dann „elf ist kleiner oder gleich zehn“, diese Aussage ist zweifellos falsch. Daher wird der Schleifenrumpf nun übersprungen und mit dem Code dahinter weitergemacht. Da in unserem Beispiel dort aber kein Code mehr folgt, wird das Programm beendet.

Zusammengefasst:

1. Quelltext vor der Schleife
2. Schleifen Bedingung
 - Erfüllt:
 - a) Schleifenrumpf
 - b) weiter mit 2
 - Nicht Erfüllt:
 - a) weiter mit 3
3. Quelltext nach der Schleife

12.2 Die do-while-Schleife

Wie versprochen lüften wir nun das Geheimnis um die fußgesteuerten Schleifen. `do-while` ist eine fußgesteuerte Schleife, das heißt, als erstes wird der Schleifenrumpf ausgeführt, danach die Bedingung überprüft und dann abhängig von der Bedingung, wieder der Rumpf ausgeführt (Bedingung erfüllt) oder mit dem Quelltext nach der Schleife fortgesetzt (Bedingung nicht erfüllt). Eine fußgesteuerte Schleife zeichnet sich also dadurch aus, dass der Schleifenrumpf mindestens ein mal ausgeführt wird.

```
do «Anweisung» while («Bedingung»);
```

Bei dieser Schleife finden wir die obige Syntax nur selten, da der Schleifenrumpf hier zwischen den beiden Schlüsselwörtern `do` und `while` steht, wird fast immer ein Anweisungsblock benutzt, auch wenn nur eine Anweisung vorhanden ist. Für Ihren Compiler spielt das natürlich keine Rolle, aber für einen Menschen der den Quelltext liest, ist es übersichtlicher.

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

Unser anfängliches Riesenprogramm sieht mit einer `do-while` Schleife so aus:

```
#include <iostream>

using namespace std;

int main() {
    unsigned int i=1, benutzer; // 2 Variablen anlegen

    cin >> benutzer;           // Benutzer gibt Zahl ein

    do {                        // Schleifenanfang
        cout << i << endl;      // Ausgabe von i
        ++i;                    // Ausgegebene Zahlen mitzählen
    } while (i <= benutzer);    // Benutzereingabe erreicht?
}
```

Sie werden feststellen das die Ausgabe dieses Programms mit der Ausgabe in der `while`-Schleifen Version übereinstimmt. Den Unterschied bemerken Sie, wenn Sie 0 eingeben, während die `while`-Version keine Ausgabe macht, gibt diese `do-while`-Version „1“ und einen Zeilenumbruch aus, denn der Schleifenrumpf wird im erst einmal ausgeführt, erst danach wird die Bedingung überprüft und entschieden ob er noch einmal ausgeführt werden muss.

Zusammengefasst:

1. Quelltext vor der Schleife
2. Schleifenrumpf
3. Schleifen Bedingung
 - Erfüllt:
 - a) weiter mit 2
 - Nicht Erfüllt:
 - a) weiter mit 4
4. Quelltext nach der Schleife

12.3 Die `for`-Schleife

Die `for`-Schleife ist etwas komplexer als die vorherigen beiden Schleifen. Sie gliedert sich in Teile:

```
for(«Initialisierungsteil»; «Bedingungsteil»; «Anweisungsteil») «Schleifenrumpf»
```

Um etwas Platz zu sparen, wurde in dieser Beschreibung einfach Schleifenrumpf geschrieben, anstatt wieder einzeln aufzuführen, dass sowohl eine einzelne (durch Semikolon abgeschlossene) Anweisung, als auch ein Anweisungsblock möglich sind. Der Bedingungsteil verhält sich genau wie bei der `while`- und der `do-while`-Schleife oder sagen wir fast genau so, denn einen kleinen aber feinen Unterschied gibt es doch. Während `while` und `do-while` immer eine Bedingung

erwarten, muss bei einer `for`-Schleife nicht unbedingt eine Bedingung angegeben werden. Wenn keine Bedingung angegeben ist, wird einfach angenommen dass die Bedingung immer erfüllt ist, Sie erhalten eine Endlosschleife. Wie das sinnvoll eingesetzt wird erfahren Sie in Kürze.

Im Anweisungsteil können Sie eine beliebige Anweisung ausführen, dieser Teil wird oft verwendet, um Variablen bei jedem Schleifendurchlauf hoch oder runter zu zählen. Im nächsten Beispiel wird dies auch demonstriert. Es ist auch möglich, mehrere solcher „hoch- oder runter-Zähl“-Anweisungen durch Komma getrennt anzugeben, das wird im nächsten Beispiel aber nicht gemacht. Der Anweisungsteil wird übrigens direkt nach dem Schleifenrumpf und vor dem nächsten Bedingungstest ausgeführt. Der Initialisierungsteil ist dem Anweisungsteil dahingehend ähnlich, als dass auch hier eine beliebige Anweisung ausgeführt werden kann. Zusätzlich ist es hier aber noch möglich Variablen eines Datentyps anzulegen. Sie können also problemlos 2 `int`-Variablen anlegen, aber nicht eine `int`- und eine `char`-Variable. Der Initialisierungsteil wird nur einmal am Beginn der Schleife ausgeführt.

Thema wird später näher erläutert...

Im Kapitel Lebensdauer und Sichtbarkeit von Objekten^a werden Sie noch etwas genaueres darüber erfahren wo Sie die im Initialisierungsteil der Schleife angelegten Variablen verwenden können. Dort werden Sie auch erfahren wie eine `for`-Schleife mit Hilfe einer `while`-Schleife „nachgebaut“ werden kann.

Jetzt sollten Sie sich jedoch merken das Sie eine solche Variable, nur innerhalb der Schleife, also nicht mehr nach ihrem verlassen, verwenden können.

^a Kapitel 17 auf Seite 99

```
#include <iostream>

using namespace std;

int main() {
    unsigned int benutzer;                // Variablen für Benutzereingabe

    cin >> benutzer;                      // Benutzer gibt Zahl ein

    for(unsigned int i = 1; i <= benutzer; ++i) // for-Schleife
        cout << i << endl;                // Ausgabe von i
}
```

Zusammengefasst:

1. Quelltext vor der Schleife
2. Initialisierungsteil der Schleife
3. Schleifen Bedingung
 - Erfüllt:
 - a) Schleifenrumpf
 - b) Anweisungsteil
 - c) weiter mit 3
 - Nicht Erfüllt:

- a) weiter mit 4
4. Quelltext nach der Schleife

12.4 Die `break`-Anweisung

Jetzt ist es an der Zeit, das Geheimnis um die sinnvolle Verwendung von Endlosschleifen zu enthüllen. Die `break`-Anweisung wird innerhalb von Schleifen verwendet, um die Schleife sofort zu beenden. Der Quelltext wird dann ganz normal nach der Schleife fortgesetzt. `break` können Sie in jeder der drei Schleifen verwenden. Das folgende Beispiel demonstriert den Einsatz von `break`, anhand unseres Lieblingsprogramms in diesem Kapitel.

```
#include <iostream>

using namespace std;

int main() {
    unsigned int benutzer, i=1;    // Variablen für Benutzereingabe

    cin >> benutzer;              // Benutzer gibt Zahl ein

    for(;;){                      // Endlos-for-Schleife
        cout << i << "\n";        // Ausgabe von i
        ++i;                      // Variable erhöhen
        if(i>benutzer)break;      // Abbrechen, wenn Bedingung erfüllt
    }
}
```

Sie können aus jeder Schleife eine Endlosschleife machen, hier wurde die `for`-Schleife gewählt um zu zeigen wie Sie ohne Bedingung aussieht. Bei einer `while`- oder `do-while`-Schleife könnten Sie beispielsweise `true` als Bedingung angeben. Die `for`-Schleife legt jetzt übrigens das gleiche Verhalten an den Tag, wie eine `do-while`-Schleife. Sie können auch problemlos nur einen oder zwei Teile des Schleifenkopfes bei `for`-Schleife übergeben, wichtig ist nur, dass Sie die beiden Semikolons immer angeben, da Sie dem Compiler mitteilen, was welcher Teil des Schleifenkopfes ist.

Nun wissen Sie wieder etwas mehr über `for`, dabei sollte es unter dieser Überschrift doch eigentlich um `break` gehen. Wie Sie aber sehen, hängt letztlich alles mit allem zusammen und so ist es oft schwer eine richtige Abgrenzung zu schaffen.

12.5 Die `continue`-Anweisung

Das zweite wichtige Schlüsselwort für Schleifen ist `continue`. Es wird genau so verwendet wie `break`, bricht die Schleife allerdings nicht völlig ab, sondern setzt die Codeausführung am Ende des Schleifenrumpfes fort. Für `while` und `do-while` bedeutet das beim Bedingungstest, für

die `for`-Schleife beim Anweisungsteil. Mit `continue` können Sie also den Rest des aktuellen Schleifendurchlaufs überspringen. Wir sehen uns das wieder anhand des Beispiels an.

```
#include <iostream>

using namespace std;

int main() {
    unsigned int benutzer;           // Variablen für Benutzereingabe

    cin >> benutzer;                // Benutzer gibt Zahl ein

    for(unsigned int i = 1; i <= benutzer; ++i) { // for-Schleife
        if (i%2 == 0) continue;      // alle geraden Zahlen überspringen
        cout << i << endl;          // Ausgabe von i
    }
}
```

Hier werden nur ungrade Zahlen ausgegeben, da der `%`-Operator (liefert den Rest einer Ganzzahligen Division) bei allen geraden Zahlen, (teilt man durch 2 ist als Rest ja nur 0 oder 1 möglich,) 0 zurückliefert und somit `continue` ausgeführt wird.

Natürlich könnten Sie das auch noch auf eine andere Weise realisieren. Aber es gibt ja beim Programmieren viele Wege, die nach Rom führen, wie in diesem Kapitel anhand der verschiedenen Schleifen schon bewiesen wurde. Leider gibt es aber noch mehr Wege, die auf direktem Wege an Rom vorbeiführen... Aber hier ist für das Beispiel von eben noch ein Pfad, der sicher nach Rom führt:

```
#include <iostream>

using namespace std;

int main() {
    unsigned int benutzer;           // Variablen für Benutzereingabe

    cin >> benutzer;                // Benutzer gibt Zahl ein

    for(unsigned int i = 1; i <= benutzer; i += 2) // for-Schleife mit 2er Schritten
        cout << i << endl;          // Ausgabe von i
}
```

12.6 Kapitelanhang

Aufgaben

Aufgabe 1

Schreiben Sie mithilfe einer Schleife ein kleines Spiel. Der Spielablauf lautet:

- Am Anfang gibt der Spieler einen Zahlenbereich ein. (Zum Beispiel: 1-100)
- Der Spieler muss sich innerhalb dieses Bereiches eine Zahl merken (eingegebene Grenzzahlen sind nicht zulässig).
- Das Programm soll dann die Zahl erraten. Der Benutzer teilt dem Programm mit, ob die Zahl an die er denkt kleiner, größer oder gleich der vom Programm geratenen Zahl ist. Die kann zum Beispiel über die Eingabe von <, > und = erfolgen.

Musterlösung

Es gibt natürlich viele Wege dieses Problem zu lösen und Sie sehen ja ob Ihr Programm funktioniert oder nicht. Hier wird nur eine Musterlösung vorgestellt, falls Sie überhaupt nicht zurechtkommen oder sich einfach dafür interessieren wie der Autor an das Problem herangegangen ist.

```
#include <iostream>

using namespace std;

int main() {
    int min, max; // Variablen für den möglichen Zahlenbereich
    int zahl; // Zahl die der Rechner vermutet

    cout << "Wo fängt die Zahlenreihe an?: "; // Zahlenbereich abfragen
    cin >> min; // Benutzereingabe einlesen

    cout << "Wo hört die Zahlenreihe auf?: "; // Zahlenbereich abfragen
    cin >> max; // Benutzereingabe einlesen

    for (char eingabe = '0'; eingabe != '='; ) { // Abbrechen wenn eingabe '=' ist
        zahl = min + (max - min) / 2; // Mittlere Zahl berechnen
        cout << "Denken Sie an " << zahl << "? "; // Vermutung ausgeben
        cin >> eingabe; // Antwort einlesen

        if (eingabe == '<') // Ist die Zahl kleiner?
            max = zahl; // Setzte max auf den zu großen Wert zahl
        else if (eingabe == '>') // Ist die Zahl größer?
            min = zahl; // Setzte min auf den zu kleinen Wert zahl
        else if (eingabe != '=') // Ist Eingabe auch kein Gleichheitszeichen
            cout << "Sie haben ein unzulässiges Zeichen eingegeben!\n"; // Fehlerhafte Eingabe
        melden

        if (min+1 >= max) { // Keine Zahl mehr im gültigen Bereich
            cout << "Sie sind ein Lügner!\n"; // Das Programm ist äußerst entsetzt
            break; // Schleife wird abgebrochen
        }
    }

    cout << "Die von Ihnen gemerkte Zahl ist " << zahl << "!" << endl; // Ausgabe der erratenen
```

Zahl

}

Ausgabe

Wo fängt die Zahlenreihe an?: 0

Wo hört die Zahlenreihe auf?: 100

Denken Sie an 50? <

Denken Sie an 25? >

Denken Sie an 37? <

Denken Sie an 31? >

Denken Sie an 34? <

Denken Sie an 32? >

Denken Sie an 33? =

Die von Ihnen gemerkte Zahl ist 33!

13 Auswahl

Verzweigungen kennen Sie schon, daher sollte es Ihnen nicht schwer fallen, ein Programm zu schreiben, welches abfragt, welche der verschiedenen, in der Überschrift benannten Leckereien, Sie am liebsten mögen. Das ganze könnte etwa so aussehen:

```
#include <iostream>

using namespace std;

int main(){
    int auswahl;

    cout << "Wählen Sie Ihre Lieblingsleckerei:\n"
         << "1 - Käsesahnetorte\n"
         << "2 - Streuselkuchen\n"
         << "3 - Windbeutel\n";

    cin >> auswahl;

    if(auswahl==1)
        cout << "Sie mögen also Käsesahnetorte!\n";
    else if(auswahl==2)
        cout << "Streuselkuchen ist ja auch lecker...\n";
    else if(auswahl==3)
        cout << "Windbeutel sind so flüchtig wie ihr Name, das können Sie sicher bestätigen?\n";
    else
        cout << "Wollen Sie wirklich behaupten, das Ihnen nichts davon zusagt?\n";

    return 0;
}
```

Ausgabe

```
Wählen Sie Ihre Lieblingsleckerei:
1 - Käsesahnetorte
2 - Streuselkuchen
3 - Windbeutel
<Eingabe>2</Eingabe>
Streuselkuchen ist ja auch lecker...
```

Das funktioniert natürlich, aber finden Sie nicht auch, dass der (Schreib-)Aufwand ein wenig zu groß ist? Außerdem ist diese Schreibweise nicht gerade sofort einleuchtend. Für unseren Fall, wäre

es schöner, wenn wir die Variable `auswahl` als zu untersuchendes Objekt festlegen könnten und dann einfach alle Fälle die uns interessieren durchgehen könnten. Praktischerweise bietet C++ ein Schlüsselwort, das genau dies ermöglicht.

13.1 `switch` und `break`

Eine `switch`-Anweisung hat die folgende Form:

```
switch(«Ganzzahlige Variable»){
    case «Ganzzahl»: «Anweisungsblock»
    case «Ganzzahl»: «Anweisungsblock»
    «...»
    default: «Anweisungsblock»
}
```

Auf unser Beispiel angewandt, würde das dann so aussehen:

```
#include <iostream>

using namespace std;

int main(){
    int auswahl;

    cout << "Wählen Sie Ihre Lieblingsleckerei:\n"
         << "1 - Käsesahnetorte\n"
         << "2 - Streuselkuchen\n"
         << "3 - Windbeutel\n";

    cin >> auswahl;

    switch(auswahl){
        case 1: cout << "Sie mögen also Käsesahnetorte!";
        case 2: cout << "Streuselkuchen ist ja auch lecker...";
        case 3: cout << "Windbeutel sind so flüchtig wie ihr Name, das können Sie sicher
bestätigen?";
        default: cout << "Wollen Sie wirklich behaupten, dass Ihnen nichts davon zusagt?";
    }

    return 0;
}
```

Ausgabe

```
Wählen Sie Ihre Lieblingsleckerei:
1 - Käsesahnetorte
2 - Streuselkuchen
```

```

3 - Windbeutel
<Eingabe>2</Eingabe>
Streuselkuchen ist ja auch lecker...
Windbeutel sind so flüchtig wie ihr Name, das können Sie sicher bestätigen?
Wollen Sie wirklich behaupten, dass Ihnen nichts davon zusagt?

```

Nun ja, Sie haben ein neues Schlüsselwort kennen gelernt, setzen es ein und kommen damit auch prompt zum falschen Ergebnis. Sieht so aus, als würde jetzt jeder der Sätze ab dem Ausgewählten ausgegeben. In der Tat handelt `switch` genau so. Es führt alle Anweisungen, ab dem Punkt aus, an dem das `case`-Argument mit der übergebenen Variable (in unserem Fall `auswahl`) übereinstimmt. Um das Ausführen nachfolgender Anweisungen innerhalb von `switch` zu verhindern, benutzen Sie das Schlüsselwort `break`. Unser Beispiel sieht dann also folgendermaßen aus:

```

#include <iostream>

using namespace std;

int main(){
    int auswahl;

    cout << "Wählen Sie Ihre Lieblingsleckerei:\n"
           "1 - Käsesahnetorte\n"
           "2 - Streuselkuchen\n"
           "3 - Windbeutel\n";

    cin >> auswahl;

    switch(auswahl){
        case 1:
            cout << "Sie mögen also Käsesahnetorte!";
            break;
        case 2:
            cout << "Streuselkuchen ist ja auch lecker...";
            break;
        case 3:
            cout << "Windbeutel sind so flüchtig wie ihr Name, das können Sie sicher
bestätigen?";
            break;
        default:
            cout << "Wollen Sie wirklich behaupten, dass Ihnen nichts davon zusagt?";
    }

    return 0;
}

```

Ausgabe

```

Wählen Sie Ihre Lieblingsleckerei:
1 - Käsesahnetorte

```

```
2 - Streuselkuchen
3 - Windbeutel
<Eingabe>2</Eingabe>
Streuselkuchen ist ja auch lecker...
```

Nun haben Sie wieder das ursprüngliche, gewünschte Verhalten. Was Sie noch nicht haben, ist eine Erklärung, warum extra ein `break` aufgerufen werden muss, um das Ausführen folgender `case`-Zweige zu unterbinden. Aber das werden Sie gleich erfahren.

13.2 Nur Ganzzahlen

Wie bereits aus der oben stehenden Syntaxangabe hervorgeht, kann `switch` nur mit C++-Programmierung/ Einführung in C++/ Variablen, Konstanten und ihre Datentypen#Ganzzahlen¹ benutzt werden. Dies hängt damit zusammen, dass `switch` *ausschließlich* auf Gleichheit mit einem der `case`-Zweige vergleicht. Bei `int`-Werten (mit der üblichen Größe von 4 Byte) ist das noch eine „überschaubare“ Menge von maximal 4.294.967.296 (2^{32}) `case`-Zweigen. Ein `float` kann die gleiche Menge unterschiedlicher Zahlen darstellen, sofern er ebenfalls 4 Byte groß ist. Aber versuchen Sie mal einen genauen `float`-Wert aufzuschreiben. Sie arbeiten also immer mit Näherungen, wenn Sie Gleitkommazahlen in Ihrem Code benutzen. Ihr Compiler regelt das für Sie, er weist der Gleitkommazahl den zu Ihrem Wert am nächsten liegenden, darstellbaren Wert zu. Deshalb ist es so gut wie nie sinnvoll (aber natürlich trotzdem möglich), einen Test auf Gleichheit für Gleitkommazahlen durchzuführen. Bei `switch` wurde das jedoch unterbunden.

13.3 Zeichen und mehrere `case`-Zweige

Eine `if`-Anweisung trifft Ihre Entscheidung hingegen, anhand der Tatsache ob eine Bedingung erfüllt ist, oder nicht. Für die `if`-Anweisung gibt es also nur zwei Möglichkeiten. Allerdings gibt es eine ganze Menge von Möglichkeiten, wie der Zustand `true`, oder `false` erreicht werden kann. Die Rede ist von Vergleichen und Verknüpfungen (wie `&&` (= und) oder `||` (= oder)) und der Umwandlung von anderen Datentypen nach `true`, oder `false`. Für ganzzahlige Datentypen gilt etwa, dass jeder Wert ungleich 0 `true` ergibt, aber 0 hingegen wird als `false` ausgewertet. Für eine genaue Beschreibung dessen, was hier noch mal kurz umrissen wurde, sehen Sie sich das Kapitel über Verzweigungen² an.

Also gut, `switch` kann nur auf Gleichheit testen und erlaubt auch keine Verknüpfungen. Oder sagen wir besser: fast keine. Denn genau deshalb gibt es diese scheinbar aufwendige Regelung mit dem `break`. Das folgende Beispiel demonstriert die Verwendung von Zeichen und die Anwendung von mehreren `case`-Zweigen, die den gleichen Code ausführen (was einer Verknüpfung mit `||` zwischen mehreren Vergleichen mittels `==` gleichkommt):

```
#include <iostream>
```

¹ Kapitel 9.1.3 auf Seite 34

² Kapitel 11 auf Seite 51

```
using namespace std;

int main(){
    char auswahl;

    cout << "Wählen Sie Ihre Lieblingsleckerei:\n"
           "K - Käsesahnetorte\n"
           "S - Streuselkuchen\n"
           "W - Windbeutel\n";

    cin >> auswahl;

    switch(auswahl){
        case 'k':
        case 'K':
            cout << "Sie mögen also Käsesahnetorte!";
            break;
        case 's':
        case 'S':
            cout << "Streuselkuchen ist ja auch lecker...";
            break;
        case 'w':
        case 'W':
            cout << "Windbeutel sind so flüchtig wie ihr Name, das können Sie sicher
bestätigen?";
            break;
        default:
            cout << "Wollen Sie wirklich behaupten, dass Ihnen nichts davon zusagt?";
    }

    return 0;
}
```

Ausgabe

```
Wählen Sie Ihre Lieblingsleckerei:
K - Käsesahnetorte
S - Streuselkuchen
W - Windbeutel
<Eingabe>s</Eingabe>
Streuselkuchen ist ja auch lecker...
```

Dieses Programm benutzt die Anfangsbuchstaben anstatt der bisherigen Durchnummerierung für die Auswahl genutzt. Außerdem ist es möglich sowohl den großen, als auch den kleinen Buchstaben einzugeben. Beides führt den gleichen Code aus.

14 Ein Taschenrechner wird geboren

In diesem Kapitel wollen wir einen kleinen Taschenrechner für die Kommandozeile schreiben. Dieser wird in späteren Kapiteln noch verbessert, aber fürs Erste lernt er nur die vier Grundrechenarten und kann auch nur mit je 2 Zahlen rechnen.

14.1 Die Eingabe

Eine Aufgabe besteht aus zwei Zahlen, die durch ein Rechenzeichen getrennt sind. Für die Zahlen verwenden wir den Typ `double`. Das Rechenzeichen lesen wir als `char` ein. Außerdem brauchen wir noch eine Variable in der wir das Ergebnis speichern. Diese soll ebenfalls vom Typ `double` sein.

```
double zahl1, zahl2, ergebnis;
char rechenzeichen;

cin >> zahl1 >> rechenzeichen >> zahl2;
```

14.2 Die 4 Grundrechenarten

Wie sicher jeder weiß, sind die 4 Grundrechenarten Addition (+), Subtraktion (-), Multiplikation (*) und Division (/). Da die Variable `rechenzeichen` vom Typ `char` und `char` ein ganzzahliger Typ ist, bietet es sich an, die `switch`-Anweisung zu verwenden um die richtige Rechnung zu ermitteln und durchzuführen. Wenn ein ungültiges Rechenzeichen eingegeben wird, geben wir eine Fehlermeldung aus und beenden das Programm.

```
switch(rechenzeichen){
    case '+': ergebnis = zahl1+zahl2; break;
    case '-': ergebnis = zahl1-zahl2; break;
    case '*': ergebnis = zahl1*zahl2; break;
    case '/': ergebnis = zahl1/zahl2; break;
    default: cout << "unbekanntes Rechenzeichen...\n"; return 1;
}
```

Der Rückgabewert `1` - ausgelöst von `return 1;` - beendet das Programm, die `1` (und jeder andere Wert ungleich `0` auch) sagt dabei aus, dass im Programm ein Fehler auftrat.

14.3 Das ganze Programm

Hier ist noch einmal eine Zusammenfassung unseres Taschenrechners:

```
#include <iostream>

using namespace std;

int main(){
    double zahl1, zahl2, ergebnis;           // Variablen für Zahlen
    char rechenzeichen;                     // Variable fürs Rechenzeichen

    cout << "Geben Sie eine Rechenaufgabe ein: "; // Eingabeaufforderung ausgeben
    cin >> zahl1 >> rechenzeichen >> zahl2;    // Aufgabe einlesen

    switch(rechenzeichen){                  // Wert von rechenzeichen ermitteln
        case '+': ergebnis = zahl1+zahl2; break; // entsprechend dem
        case '-': ergebnis = zahl1-zahl2; break; // Rechenzeichen
        case '*': ergebnis = zahl1*zahl2; break; // das Ergebnis
        case '/': ergebnis = zahl1/zahl2; break; // berechnen
        // Fehlerausgabe und Programm beenden, falls falsches Rechenzeichen eingegeben wurde
        default: cout << "unbekanntes Rechenzeichen...\n"; return 1;
    }

    // Aufgabe noch mal komplett ausgeben
    cout << zahl1 << ' ' << rechenzeichen << ' ' << zahl2 << " = " << ergebnis << '\n';
}

```

Ausgabe

```
Geben Sie eine Rechenaufgabe ein: <Eingabe>99 / 3</Eingabe>
99 / 3 = 33

```

15 Zusammenfassung

Ein C++-Programm beginnt mit der Hauptfunktion `main()`.

15.1 Ein- und Ausgabe

Die Ein- und Ausgabe erfolgt in C++ über Streams, mehr dazu erfahren Sie in einem der folgenden Abschnitte. Um die Ein- und Ausgabe zu nutzen müssen Sie die Headerdatei `iostream` einbinden. Die Streamobjekte `cin` und `cout` liegen im Namensraum `std`. Auf Elemente innerhalb eines Namensraums wird mit dem Bereichsoperator `::` zugegriffen.

```
#include <iostream>

int main(){
    std::cout << "Das wird Ausgegeben.\n";

    int wert;
    std::cin >> wert;
    std::cout << "Wert ist: " << wert << std::endl;
}
```

`endl` fügt einen Zeilenumbruch ein und leert anschließend den Ausgabepuffer.

Mit der `using`-Direktive können auch alle Elemente eines Namensraums verfügbar gemacht werden.

```
#include <iostream>

using namespace std;

int main(){
    cout << "Das wird Ausgegeben.\n";

    int wert;
    cin >> wert;
    cout << "Wert ist: " << wert << endl;
}
```

15.2 Kommentare

Es gibt in C++ zwei Arten von Kommentaren

```
// Kommentare über eine Zeile

/* und Kommentare über mehrere
Zeilen*/
```

15.3 Rechnen

C++ beherrscht die 4 Grundrechenarten, die Operatoren lauten +, -, * und /. Es gilt Punkt- vor Strichrechnung und Klammern ändern die Auswertungsreihenfolge. Weiterhin bietet C++ mit dem Zuweisungsoperator kombinierte Rechenoperatoren:

```
lvalue += rvalue;
lvalue -= rvalue;
lvalue *= rvalue;
lvalue /= rvalue;
```

Für Ganzzahlige Datentypen stehen weiterhin der Inkrement- und der Dekrementoperator, jeweils in einer Prä- und einer Postfixvariante zur Verfügung:

```
++lvalue; // Erhöht den Wert und gibt ihn zurück
lvalue++; // Erhöht den Wert und gibt den alten Wert zurück
--lvalue; // Erniedrigt den Wert und gibt ihn zurück
lvalue--; // Erniedrigt den Wert und gibt den alten Wert zurück
```

Die Rückgabe der Präfixvariante ist ein lvalue, die Postfixvariante gibt ein rvalue zurück.

15.4 Variablen

Die Syntax zur Deklaration einer Variable lautet:

```
«Datentyp» «Name»;
```

15.4.1 Datentypen

Die C++-Basisdatentypen sind:

- Typlos
 - void (wird später erläutert)
- Logisch
 - bool
- Zeichen (auch Ganzzahlig)
 - char (signed char **und** unsigned char)
 - wchar_t
- Ganzzahlig
 - short (signed short **und** unsigned short)
 - int (signed int **und** unsigned int)
 - long (signed long **und** unsigned long)
- Gleitkommazahlen
 - float
 - double
 - long double

Die Typmodifizierer `signed` und `unsigned` geben an, ob es sich um einen vorzeichenbehafteten oder vorzeichenlosen Typ handelt. Ohne diese Modifizierer wird ein Vorzeichenbehafteter Typ angenommen, außer bei `char`. Hier ist es Implementierungsabhängig. `signed` und `unsigned` können ohne Datentyp verwendet werden, dann wird `int` als Typ angenommen.

Das Schlüsselwort `const` zeichnet einen Datentyp als konstant aus. Es gehört immer zu dem, was links davon steht, es sei denn, links steht nichts mehr, dann gehört es zu dem Teil auf der rechten Seite. Es ist somit egal ob Sie `const` vor oder hinter einen Basisdatentyp schreiben:

```
const «Datentyp» «Variablenname»;  
«Datentyp» const «Variablenname»;
```

C++ macht keine Angaben über die Größen der Datentypen, es ist lediglich folgendes festgelegt:

- char <= short <= int <= long
- float <= double <= long double

15.4.2 Initialisierung

In C++ gibt es 2 Möglichkeiten für eine Initialisierung:

```
Datentyp «Variablenname» = «Wert»;  
Datentyp «Variablenname» («Wert»);
```

Die erste Variante ist verbreiteter, aber nur möglich wenn zur Initialisierung nur 1 Wert benötigt wird. Alle Basisdatentypen erwarten genau einen Wert.

15.4.3 Typaufwertung

Wird ein Operator auf verschiedene Datentypen angewendet, müssen vor der Operation beide in den gleichen Typ konvertiert werden. Der Zieltyp ist üblicherweise der kleinste, der beide Werte darstellen kann. Für genauere Informationen lesen Sie bitte im Kapitel Rechnen mit unterschiedlichen Datentypen¹.

15.5 Kontrollstrukturen

15.5.1 Anweisungsblöcke

Mit einem Anweisungsblock können mehrere Anweisungen zusammengefasst werden. Er kann überall stehen wo auch eine Anweisung stehen könnte. Die Syntax sieht folgendermaßen aus:

```
{
    «Anweisung»
    «...»
}
```

15.5.2 Verzweigung

Eine Verzweigung hat in C++ folgende Syntax

```
if («Bedingung»)
    «Anweisung»
else
    «Anweisung»
```

Die Bedingung ist immer ein Logischer Wert. Zahlenwerte werden implizit in logische Werte konvertiert. 0 entspricht dabei immer `false`, alle anderen Werte werden zu `true` umgewandelt.

15.5.3 Schleifen

Es gibt 3 Arten von Schleifen. Prinzipiell lässt sich in C++ jede der 3 Arten durch jede der jeweils andere simulieren. Allerdings kann je nach Anwendungsfall, rücksichtlich der Tipparbeit, einer der drei Arten am praktischsten sein.

```
// while-Schleife (Kopfgesteuert)
while («Bedingung»)
```

¹ Kapitel 10 auf Seite 45

```

«Anweisung»

// do-while-Schleife: Alle Anweisungen werden mindestens einmal ausgeführt. (Fußgesteuert)
do
    «Anweisung»
while («Bedingung»);

// for-Schleife: Ermöglicht die Initialisierung, die Bedingung (Kopfgesteuert) und eine Aktion,
// die nach jedem Schleifendurchlauf stattfindet, im Schleifenkopf zu platzieren
for («Deklarationen (z.B. Zählvariablen)»; «Bedingung»; «Aktion nach einem Durchlauf (z.B.
hochzählen)»)
    «Anweisung»

```

15.5.4 Auswahl

Mit der `switch`-Anweisung kann eine Variable auf mehrere (konstante) Werte verglichen werden. Jedem dieser Werte können eine oder mehrere Anweisungen folgen, welcher im Falle einer Übereinstimmung ausgeführt werden, bis das Ende des `switch`-Blocks erreicht wird.

```

switch («Variable») {
    case «Wert»:
        «Anweisung»
        «...»
        »break;«

    «...»

    default: // Wird ausgeführt, falls die Variable keinem der Werte entsprach
        «Anweisung»
        «...»
}

```

Hinweis

Wird das `break;` am Ende eines Anweisungsblockes weggelassen, *werden* auch die darauf folgenden Blöcke ausgeführt, bis ein `break;` gefunden oder das Ende der `switch`-Anweisung erreicht ist.

16 Prozeduren und Funktionen

Unter einer Funktion (*function*, in anderen Programmiersprachen auch Prozedur oder Subroutine genannt) versteht man ein Unterprogramm, das eine bestimmte Aufgabe erfüllt. Funktionen sind unter anderem sinnvoll, um sich oft wiederholende Befehle zu kapseln, so dass diese nicht jedesmal neu geschrieben werden müssen. Zudem verbessert es die Übersichtlichkeit der Quellcode-Struktur erheblich, wenn der Programmtext logisch in Abschnitte unterteilt wird.

16.1 Parameter und Rückgabewert

Die spezielle Funktion `main()` ist uns schon mehrfach begegnet. In C++ lassen sich Funktionen nach folgenden Kriterien unterscheiden:

- Eine Funktion kann Parameter besitzen, oder nicht.
- Eine Funktion kann einen Wert zurückgeben, oder nicht.

Dem Funktionsbegriff der Mathematik entsprechen diejenigen C++-Funktionen, die sowohl Parameter haben als auch einen Wert zurückgeben. Dieser Wert kann im Programm weiter genutzt werden, um ihn z.B. einer Variablen zuzuweisen.

```
int a = f(5); // Aufruf einer Funktion
```

Damit diese Anweisung fehlerfrei kompiliert wird, muss vorher die Funktion `f()` deklariert worden sein. Bei einer Funktion bedeutet *Deklaration* die Angabe des *Funktionsprototyps*. Das heißt, der Typ von Parametern und Rückgabewert muss angegeben werden. Das folgende Beispiel deklariert bspw. eine Funktion, die einen Parameter vom Typ `int` besitzt und einen `int`-Wert zurückgibt.

```
int f (int x); // Funktionsprototyp == Deklaration
```

Soll eine Funktion keinen Wert zurückliefern, lautet der Rückgabety formal `void`.

Nach dem Compilieren ist das Linken der entstanden Objektdateien zu einem ausführbaren Programm nötig. Der Linker benötigt die Definition der aufzurufenden Funktion. Eine Funktionsdefinition umfasst auch die Implementation der Funktion, d.h. den Code, der beim Aufruf der Funktion ausgeführt werden soll. In unserem Fall wäre das:

```
int f(int x); // Funktionsdeklaration
```

```
int main(){
```

```
int a = f(3); // Funktionsaufruf
// a hat jetzt den Wert 9
}

int f(int x){ // Funktionsdefinition
    return x * x;
}
```

Hinweis

Innerhalb eines Programms dürfen Sie eine Funktion beliebig oft (übereinstimmend!) deklarieren, aber nur einmal definieren.

Der Compiler muss die Deklaration kennen, um eventuelle Typ-Unverträglichkeiten abzufangen. Würden Sie die obige Funktion z.B. als `int a = f(2.5);` aufrufen, käme die Warnung, dass `f()` ein ganzzahliges Argument erwartet und keine Fließkommazahl. Eine Definition ist für den Compiler auch immer eine Deklaration, das heißt Sie müssen nicht explizit eine Deklaration einfügen um eine Funktion aufzurufen, die *zuvor* definiert wurde.

Die Trennung von Deklaration und Definition kann zu übersichtlicher Code-Strukturierung bei größeren Projekten genutzt werden. Insbesondere ist es sinnvoll, Deklarationen und Definitionen in verschiedene Dateien zu schreiben. Oft will man, wenn man fremden oder alten Code benutzt, nicht die Details der Implementierung einer Funktion sehen, sondern nur das Format der Parameter o.ä. und kann so in der Deklarationsdatei (*header file*, üblicherweise mit der Endung `.hpp`, z.T. auch `.h` oder `.hh`) nachsehen, ohne durch die Funktionsrümpfe abgelenkt zu werden. (Bei proprietärem Fremdcode bekommt man die Implementation in der Regel gar nicht zu Gesicht!)

Bei einer Deklaration ist es nicht nötig, die Parameternamen mit anzugeben, denn diese sind für den Aufruf der Funktion nicht relevant. Es ist allerdings üblich die Namen dennoch mit anzugeben um zu verdeutlichen was der Parameter darstellt. Der Compiler ignoriert die Namen in diesem Fall einfach, weshalb es auch möglich ist den Parametern in der Deklaration und der Definition unterschiedliche Namen zu geben. Allerdings wird davon aus Gründen der Übersichtlichkeit abgeraten.

Mit der Anweisung `return` gibt die Funktion einen Wert zurück, in unserem Beispiel `x * x`, wobei die Variable `x` als *Parameter* bezeichnet wird. Als Argument bezeichnet man eine Variable oder einen Wert, mit denen eine Funktion aufgerufen wird. Bei Funktionen mit dem Rückgabebetyp `void` schreiben Sie einfach `return;` oder lassen die `return`-Anweisung ganz weg. Nach einem `return` wird die Funktion sofort verlassen, d.h. alle nachfolgenden Anweisungen des Funktionsrümpfs werden ignoriert.

Erwartet eine Funktion mehrere Argumente, so werden die Parameter durch Kommas getrennt. Eine mit

```
int g(int x, double y);
```

deklarierte Funktion könnte z.B. so aufgerufen werden:

```
int a = g(123, -4.44);
```

Für eine leere Parameterliste schreiben Sie hinter dem Funktionsnamen einfach ().

```
int h(){          // Deklaration von h()
    // Quellcode ...
}
```

```
int a = h(); // Aufruf von h()
```

Hinweis

Vergessen Sie beim Aufruf einer Funktion ohne Parameter nicht die leeren Klammern. Andernfalls erhalten Sie nicht den von der Funktion zurückgelieferten Wert, sondern die Adresse der Funktion. Dies ist ein „beliebter“ Anfängerfehler, daher sollten Sie im Falle eines Fehlers erst einmal überprüfen, ob Sie nicht vielleicht irgendwo eine Funktion ohne Parameter falsch aufgerufen haben.

16.2 Übergabe der Argumente

C++ kennt zwei Varianten, wie einer Funktion die Argumente übergeben werden können: *call-by-value* und *call-by-reference*.

16.2.1 call-by-value

Bei *call-by-value* (Wertübergabe) wird der *Wert* des Arguments in einen Speicherbereich kopiert, auf den die Funktion mittels Parametername zugreifen kann. Ein Werteparameter verhält sich wie eine lokale Variable, die „automatisch“ mit dem richtigen Wert initialisiert wird. Der Kopiervorgang kann bei Klassen (Thema eines späteren Kapitels) einen erheblichen Zeit- und Speicheraufwand bedeuten!

```
#include <iostream>

void f1(const int x) {
    x = 3 * x;          // ungültig, weil Konstanten nicht überschrieben werden dürfen
    std::cout << x << std::endl;
}

void f2(int x) {
    x = 3 * x;
    std::cout << x << std::endl;
}

int main() {
    int a = 7;
    f1(a);             // Ausgabe: 21
    f2(5);             // Ausgabe: 15
    std::cout << x;    // Fehler! x ist hier nicht definiert
    std::cout << a;    // a hat immer noch den Wert 7
}
```

Wird der Parameter als `const` deklariert, so darf ihn die Funktion nicht verändern (siehe erstes Beispiel). Im zweiten Beispiel kann die Variable `x` verändert werden. Die Änderungen betreffen aber nur die lokale Kopie und sind für die aufrufende Funktion nicht sichtbar.

16.2.2 call-by-reference

Die Sprache C kennt nur call-by-value. Sollen die von einer Funktion vorgenommen Änderungen auch für das Hauptprogramm sichtbar sein, müssen sogenannte *Zeiger* verwendet werden. C++ stellt ebenfalls Zeiger¹ zur Verfügung. C++ gibt Ihnen aber auch die Möglichkeit, diese Zeiger mittels Referenzen² zu umgehen. Beide sind jedoch noch Thema eines späteren Kapitels.

Im Gegensatz zu call-by-value wird bei *call-by-reference* die Speicheradresse des Arguments übergeben, also der Wert nicht kopiert. Änderungen der (Referenz-)Variable betreffen zwangsläufig auch die übergebene Variable selbst und bleiben nach dem Funktionsaufruf erhalten. Um call-by-reference anzuzeigen, wird der Operator & verwendet, wie Sie gleich im Beispiel sehen werden. Wird keine Änderung des Inhalts gewünscht, sollten Sie den Referenzparameter als `const` deklarieren um so den Speicherbereich vor Änderungen zu schützen. Fehler, die sich aus der ungewollten Änderung des Inhaltes einer übergebenen Referenz ergeben, sind in der Regel schwer zu finden.

Die im folgenden Beispiel definierte Funktion `swap()` vertauscht ihre beiden Argumente. Weil diese als Referenzen übergeben werden, überträgt sich das auf die Variablen mit denen die Funktion aufgerufen wurde:

```
#include <iostream>

void swap(int &a, int &b) {
    int tmp = a; // "temporärer" Variable den Wert von Variable a zuweisen
    a = b;      // a mit dem Wert von Variable b überschreiben
    b = tmp;    // b den Wert der "temporären" Variable zuweisen (Anfangswert von Variable a)
}

int main() {
    int x = 5, y = 10;

    swap(x, y);

    std::cout << "x=" << x << " y=" << y << std::endl;

    return 0;
}
```

Ausgabe

```
x=10 y=5
```

Tip

Ob das kaufmännische Und (&) genau nach `int (int& a, ...)`, oder genau vor der Variablen `a (int &a, ...)`, oder dazwischen `(int & a, ...)` steht, ist für den Compiler nicht von Bedeutung. Auch möglich wäre `(int&a, ...)`.

1 Kapitel 19 auf Seite 111

2 Kapitel 20 auf Seite 121

Nicht-konstante Referenzen können natürlich nur dann übergeben werden, wenn das Argument tatsächlich eine Speicheradresse hat, sprich eine Variable bezeichnet. Ein Literal z.B. 123 oder gar ein Ausdruck $1 + 2$ wäre hier nicht erlaubt. Bei `const`-Referenzen wäre das möglich, der Compiler würde dann eine temporäre Variable anlegen.

Thema wird später näher erläutert...

Wir werden die oben angesprochenen Zeiger in einem späteren Kapitel ausführlich kennen lernen. Sie ermöglichen z.B. das Arbeiten mit Arrays^a. Soll eine Funktion mit einem Array umgehen, werden ihr die Startadresse des Arrays (ein Zeiger) und seine Größe als Parameter übergeben. Wenn das Array nur gelesen werden soll, deklariert man die Werte, auf die der Zeiger zeigt, als `const`. Das Zeichen `*` signalisiert, dass es sich um einen Zeiger handelt.

```
#include <iostream>

void print(int const* array, int const arrayGroesse) {
    std::cout << "Array:" << std::endl;
    for (int i = 0; i < arrayGroesse; ++i) {
        std::cout << array[i] << std::endl;
    }
}

int main() {
    int array[] = { 3, 13, 113 };
    int n = sizeof(array) / sizeof(array[0]);

    print(array, n);
}
```

Ausgabe

```
Array:
3
13
113
```

^a <http://de.wikipedia.org/wiki/Array>

16.3 Default-Parameter

Default-Parameter dienen dazu, beim Aufruf einer Funktion nicht alle Parameter explizit angeben zu müssen. Die nicht angegebenen Parameter werden mit einer Voreinstellung (*default*) belegt. Parameter, die bei einem Aufruf einer Funktion nicht angegeben werden müssen, werden auch als „fakultative Parameter“ bezeichnet.

```
int summe(int a, int b, int c = 0, int d = 0) {
    return a + b + c + d;
}

int main() {
    int x = summe(2, 3, 4, 5); //x == 14
    x = summe(2, 3, 4);       //x == 9, es wird d=0 gesetzt
    x = summe(2, 3);         //x == 5, es wird c=0, d=0 gesetzt
}
```

```
}
```

Standardargumente werden in der Deklaration einer Funktion angegeben, da der Compiler sie beim Aufruf der Funktion kennen muss. Im obigen Beispiel wurde die Deklaration durch die Definition gemacht, daher sind die Parameter hier in der Definition angegeben. Bei einer getrennten Schreibung von Deklaration und Definition könnte das Beispiel so aussehen:

```
int summe(int a, int b, int c = 0, int d = 0); // Deklaration

int main() {
    int x = summe(2, 3, 4, 5); //x == 14
    x = summe(2, 3, 4);        //x == 9, es wird d=0 gesetzt
    x = summe(2, 3);          //x == 5, es wird c=0, d=0 gesetzt
}

int summe(int a, int b, int c, int d) { // Definition
    return a + b + c + d;
}
```

16.4 Funktionen überladen

Überladen (*overloading*) von Funktionen bedeutet, dass verschiedene Funktionen unter dem gleichen Namen angesprochen werden können. Damit der Compiler die Funktionen richtig zuordnen kann, müssen die Funktionen sich in ihrer Funktionssignatur unterscheiden. In C++ besteht die Signatur aus dem Funktionsnamen und ihren Parametern, der Typ des Rückgabewerts gehört nicht dazu. So ist es nicht zulässig, eine Funktion zu überladen, die den gleichen Namen und die gleiche Parameterliste wie eine bereits existierende Funktion besitzt und sich nur im Typ des Rückgabewerts unterscheidet. Das obige Beispiel lässt sich ohne Default-Parameter so formulieren:

```
int summe(int a, int b, int c, int d) {
    return a + b + c + d;
}

int summe(int a, int b, int c) {
    return a + b + c;
}

int summe(int a, int b) {
    return a + b;
}

int main() {
    // ...
}
```

}

16.5 Funktionen mit beliebig vielen Argumenten

Wenn die Zahl der Argumente nicht von vornherein begrenzt ist, wird als Parameterliste die sog. *Ellipse . . .* angegeben. Der Funktion werden die Argumente dann in Form einer Liste übergeben, auf die mit Hilfe der (in der Headerdatei `cstdarg` definierten) `va`-Makros zugegriffen werden kann.

```
#include <cstdarg>
#include <iostream>

int summe(int a, ...) {
    int summe = 0;
    int i = a;

    va_list Parameter;           // Zeiger auf Argumentliste
    va_start(Parameter, a);     // gehe zur ersten Position der Liste

    while (i != 0) {            // Schleife, solange Zahl nicht 0 (0 ist Abbruchbedingung)
        summe += i;             // Zahl zur Summe addieren
        i = va_arg(Parameter, int); // nächstes Argument an i zuweisen, Typ int
    }

    va_end(Parameter);         // Liste löschen

    return summe;
}

int main() {
    std::cout << summe(2, 3, 4, 0) << std::endl; // 9
    std::cout << summe(2, 3, 0) << std::endl;    // 5

    std::cout << summe(1,1,0,1,0) << std::endl; // 2 (da die erste 0 in der while-Schleife für
    Abbruch sorgt)
}
```

Ausgabe

```
9
5
2
```

Obwohl un spezifizierte Argumente manchmal verlockend aussehen, sollten Sie sie nach Möglichkeit vermeiden. Das hat zwei Gründe:

- Die `va`-Befehle können nicht erkennen, wann die Argumentliste zu Ende ist. Es muss immer mit einer expliziten Abbruchbedingung gearbeitet werden. In unserem Beispiel muss als letztes

Argument eine 0 stehen, ansonsten gibt es, je nach Compiler unterschiedliche, „interessante“ Ergebnisse.

- Die `va`-Befehle sind *Makros*, d.h. eine strenge Typüberprüfung findet nicht statt. Fehler werden - wenn überhaupt - erst zur Laufzeit bemerkt.

Tip

Mit dem kommenden C++ Standard C++11 wird mit den „variadic templates“ eine Technik eingeführt, welche diese Art von variabler Parameterliste überflüssig macht. Über Templates wie Sie im aktuellen Standard stehen, werden Sie später mehr erfahren. Sobald der neue Standard endgültig verabschiedet ist, wird es auch zu den „variadic templates“ ein Kapitel geben. Die meisten Compiler beherrschen diese Technik bereits in weiten Teilen, sofern man den neuen Standard explizit aktiviert:

```
#include <iostream>

template < typename First >
First summe(First first){
    return first;
}

template < typename First, typename ... Liste >
First summe(First first, Liste ... rest){
    return first + summe(rest ...);
}

int main() {
    std::cout << summe(2, 3, 4) << std::endl;
    std::cout << summe(2, 3) << std::endl;
    std::cout << summe(1, 1, 0, 1) << std::endl;
}
```

Ausgabe

```
9
5
3
```

Diese Implementierung ist typsicher und benötigt keine Tricks um den letzten Parameter zu kennzeichnen. Hierbei entspricht der Rückgabotyp immer dem Typ des ersten Parameters, was nicht sonderlich sinnvoll ist, aber eine Implementierung die sich auch bezüglich des Rückgabetyps sinnvoll verhält, würde an dieser Stelle endgültig zu weit führen. „Variadic templates“ sind eine extrem nützliche Technik für Fortgeschrittene. Für Sie genügt es zum gegenwärtigen Zeitpunkt zu wissen, dass sie existieren und dass man derartige Funktionen wie jede andere Funktion aufrufen kann.

16.6 Inline-Funktionen

Um den Aufruf einer Funktion zu beschleunigen, kann in die Funktionsdeklaration das Schlüsselwort `inline` eingefügt werden. Dies ist eine Empfehlung (keine Anweisung) an den Compiler, beim Aufruf dieser Funktion keine neue Schicht auf dem Stack anzulegen, sondern den Code direkt auszuführen - den Aufruf sozusagen durch den Funktionsrumpf zu ersetzen.

Da dies – wie eben schon erwähnt – nur eine Empfehlung an den Compiler ist, wird der Compiler eine Funktion nur dann tatsächlich *inline* einbauen, wenn es sich um eine kurze Funktion handelt. Ein typisches Beispiel:

```
inline int max(int a, int b) {  
    return a > b ? a : b;  
}
```

Thema wird später näher erläutert...

Das Schlüsselwort `inline` wird bei der Deklaration angegeben. Allerdings muss der Compiler beim Aufruf den Funktionsrumpf kennen, wenn er den Code direkt einfügen soll. Für den Aufruf einer `inline`-Funktion genügt also wie immer die Deklaration, um dem Compiler jedoch tatsächlich das Ersetzen des Funktionsaufrufs zu ermöglichen, muss auch die Definition bekannt sein. Über diese Eigenheit von `inline`-Funktionen erfahren Sie im Kapitel „Headerdateien“^a mehr. Auch die Bedeutung von Deklaration und Definition wird Ihnen nach diesem Kapitel klarer sein.

^a Kapitel 24 auf Seite 151

17 Lebensdauer und Sichtbarkeit von Objekten

Die Lebensdauer einer Variable beschreibt die Zeit, in der die Variable im Speicher existiert. Die Sichtbarkeit einer Variable ist vor ihrer Lebensdauer zu unterscheiden. Sie beschreibt, wie der Name schon vermuten lässt, wann man auf eine Variable über ihren Namen zugreifen kann, beziehungsweise wann dies nicht möglich ist. Wir werden in diesem Kapitel Variablen nach ihrer Lebensdauer sortiert betrachten und dabei auch immer auf ihren Sichtbarkeitsbereich eingehen.

17.1 Statische Variablen

Als statische Variablen werden alle Variablen bezeichnet, deren Lebensdauer mit der Laufzeit des Programms übereinstimmt. Sie belegen somit während der gesamten Ausführungszeit Speicherplatz, können andererseits aber auch während der gesamten Laufzeit zugegriffen werden.

17.1.1 Lokale statische Variablen

Variablen, die innerhalb einer Funktion als `static` deklariert wurden, werden als lokale statische Variablen bezeichnet, da sie bezüglich der Funktion lokal sind. Solche Variablen sind nur innerhalb der jeweiligen Funktion sichtbar. Da sie jedoch permanent existieren, können Sie über eine entsprechende Referenz oder einen Zeiger auch außerhalb der Funktion auf sie greifen. Die Initialisierung solcher Variablen erfolgt beim ersten Aufruf der Funktion.

17.1.2 Nicht-lokale statische Variablen

Alle statischen Variablen, die nicht in die Kategorie „lokal“ fallen, werden entsprechend als „nicht-lokal“ bezeichnet. Alle Variablen dieser Art sind überall sichtbar, mit Ausnahme von Variablen, die in anonymen Namensräumen deklariert wurden. Diese sind nur innerhalb der aktuellen Übersetzungseinheit sichtbar. Auch diese können natürlich dennoch mittels einer entsprechenden Referenz oder eines Zeigers auch aus anderen Übersetzungseinheiten zugegriffen werden. Als Übersetzungseinheit bezeichnet man das Kompilat einer `cpp`-Datei, welches der Compiler erstellt, bevor es vom Linker zum endgültigen Programm gebunden wird. Aus Quelltextensicht entspricht sie somit einer `cpp`-Datei inklusive aller mit `#include` eingebundenen Header. Um eine Variable in einer anderen Übersetzungseinheit sichtbar zu machen, muss sie dort deklariert werden. Beachten Sie, dass die Variable dort nicht ein zweites Mal definiert werden darf.

Globale Variablen

Variablen, die direkt im globalen Namensraum deklariert wurden, heißen globale Variablen. Im globalen Namensraum heißt, außerhalb von jeder Funktion, Klasse und jedem Namensraum. Ein geeignetes Indiz um zu prüfen, ob eine Variable global deklariert ist, ist den Zugriffsoperator (`::`) ohne Namensraum zu benutzen. Folgendes Beispiel zeigt diesen einfachen Test.

```
extern int globale; // explizite Deklaration
int global;        // Definition

int main(){
    int lokal;

    ::global; // geht -> globale Variable
    ::lokal;  // Fehler -> nicht globale Variable
}
```

Auch hierbei gibt es natürlich wieder eine Ausnahme, denn auch Variablen aus einem anonymen Namensraum lassen sich auf diese Weise zugreifen. Da sich die Nicht-lokalen, statischen Variablen aus interner Sicht jedoch ohnehin alle sehr ähnlich verhalten, spielt ihre Unterscheidung in der Praxis ohnehin eine untergeordnete Rolle. Allgemein sollten Sie globale Variablen komplett vermeiden, da sie in größeren Projekten mit mehreren Programmierern sehr schnell zu Namenskonflikten führen.

Während sich eine globale Variable jedoch über eine explizite Deklaration aus einer anderen Übersetzungseinheit heraus sichtbar machen lässt, ist dies für eine Variable in einem anonymen Namensraum nicht möglich.

Globaler Namensraum

Variablen, die innerhalb eines Namensraumes deklariert werden, vermeiden Namenskonflikte und die Zuordnung zu einzelnen Programmteilen ist schneller ersichtlich. Wenn Sie eine globale Variable benötigen, dann verpacken Sie diese in einen geeigneten Namensraum. Auch hier ist natürlich eine explizite Deklaration in einer anderen Übersetzungseinheit möglich, wobei die Deklaration natürlich auch dort im selben Namensraum erfolgen muss.

Anonymer Namensraum

Der anonyme Namensraum wurde nun schon einige Male angesprochen. Variablen, die in diesem Namensraum definiert wurden, können in der aktuellen Übersetzungseinheit wie globale Variablen behandelt werden. Da ein anonymer Namensraum nur in der aktuellen Übersetzungseinheit sichtbar ist, gibt es keine Möglichkeit seinen Inhalt aus einer anderen Übersetzungseinheit sichtbar zu machen. Der anonyme Namensraum einer anderen Übersetzungseinheit ist entsprechend wieder ein anderer, als der, in der aktuellen Datei.

Statische Klassenvariablen

Statische Klassenvariablen verhalten sich im wesentlichen wie Variablen innerhalb eines benannten Namensraumes. Um sie in einer anderen Übersetzungseinheit zu sehen, ist eine Definition der Klasse nötig, in der sie deklariert ist. Dies erfolgt üblicherweise durch Einbinden der entsprechenden Headerdatei der Klasse.

Probleme mit der Initialisierung

Das große Problem, das sich in Zusammenhang mit statischen, nicht-lokalen Variablen ergibt, ist ihre Initialisierungsreihenfolge bezüglich unterschiedlicher Übersetzungseinheiten. Diese ist nicht definiert, da es für den Compiler beziehungsweise Linker im allgemeinen unmöglich ist, herauszufinden, welche Reihenfolge die richtige ist. Betrachten Sie zur Veranschaulichung das folgende Beispiel mit zwei Übersetzungseinheiten.

```
//*****//
//***** Datei "B.hpp" *****//
// Definition von B
struct B{
    B(int value):value_(value){}
    int value_;
};

//*****//
//***** Datei "file1.cpp" *****//
#include "B.hpp"

B b(5); // Definition und Initialisierung der globalen Variable b

//*****//
//***** Datei "file2.cpp" *****//
#include <iostream>
#include "B.hpp"

extern B b; // Deklaration der Variable b

struct A{ // Definition von A
    A(B const& value):b_(value){}
    B b_;
};

A a(b); // Globale Variable a, die mit b initialisiert wird

int main(){
    std::cout << a.b_.value_ << std::endl;
```

```
}
```

Falls die Variable B zuerst initialisiert wird, liefert das Programm wie gewünscht die Ausgabe 5. Sie können ja mal versuchen Ihrem Compiler die beiden cpp-Dateien auf der Kommandozeile in unterschiedlicher Reihenfolge zu übergeben. Wahrscheinlich wird das Programm je nachdem welche Datei zuerst gebunden wird, ein anderes Ergebnis liefern. Sie können sich jedoch keinesfalls darauf verlassen, dass Sie Ihre Wunschinitialisierungsreihenfolge immer mit dieser Try-and-Error-Methode herstellen können. Es ist nicht immer möglich, weshalb es vom C++-Standard auch nicht definiert wurde.

Glücklicherweise gibt es einen einfachen Ausweg aus diesem Dilemma. Eine kleine Designänderung wird Ihr Problem sofort beheben. Wie Ihnen inzwischen bekannt ist, werden lokale, statische Variablen erst beim ersten Funktionsaufruf initialisiert. Wenn Sie also derartige Initialisierungsprobleme befürchten müssen, dann können Sie statt einer nicht-lokalen, statischen Variable eine Funktion verwenden, die eine Referenz auf eine lokale statische Variable zurückgibt. Die Initialisierungsreihenfolge ergibt sich dann automatisch und ohne Zutun von Compiler und Linker zur Laufzeit des Programms.

```
//*****//
//***** Datei "B.hpp" *****//
// Definition von B
struct B{
    B(int value):value_(value){}
    int value_;
};

//*****//
//***** Datei "file1.cpp" *****//
#include "B.hpp"

// Definition und Initialisierung einer statischen
// lokalen Variable die durch einen Funktionsaufruf
// zugegriffen werden kann
B& b(){
    static B _b(5); // Initialisierung beim ersten Aufruf
    return _b;
}

//*****//
//***** Datei "file2.cpp" *****//
#include <iostream>
#include "B.hpp"

B& b(); // Deklaration der Funktion b

struct A{ // Definition von A
    A(B const& value):b_(value){}
```

```

    B b_;
};

// Kapselungsfunktion
A& a() {
    static A _a(b()); // Initialisierung beim ersten Aufruf
    return _a;
}

int main() {
    std::cout << a().b_.value_ << std::endl;
}

```

Ausgabe

5

Diese Variante erfordert zwar einen minimal höheren Schreibaufwand, im Gegenzug erhalten Sie aber ein klar definiertes und sinnvolles Verhalten und als kleinen Bonus bekommen Sie noch einen möglicherweise hilfreichen Effekt dazu. Da die Initialisierung der Variablen nun erst dann stattfindet, wenn Sie diese tatsächlich verwenden, startet ihr Programm etwas schneller und Variablen, die während der Laufzeit nicht genutzt werden, verschwenden keine Zeit mehr mit einer unnötigen Initialisierung. Natürlich belegen sie trotzdem immer noch Speicherplatz, also gehen Sie trotz der gewonnen Laufzeitvorteile mit bedacht vor, wenn Sie statische Variablen verwenden.

17.2 Alter Text

Der Rest des Kapitels muss noch einmal überarbeitet werden, die aktuelle Version bietet jedoch schon einen grundlegenden Überblick. Ich möchte die Überarbeitung selbst vornehmen, da ich eine etwas andere Struktur vorgesehen habe. Der aktuelle Text wird natürlich in meine Version mit einfließen. ;) --Prog¹ 16:19, 6. Dez. 2010 (CET)

17.2.1 Lokale Variablen

Im Gegensatz zu globalen, werden lokale Variablen in einem bestimmten Anweisungsblock (z.B. Schleifen, `if`-Abfragen oder Funktionen) deklariert. Ihre Existenz endet, wenn dieser Block wieder verlassen wird.

```

void foo(int a) {
    int lok1 = 0; // lok1 ist eine Variable im Anweisungsblock von void foo(int a)

    if (a < 0) {
        int lok2 = 1; // lok2 ist eine Variable im if-Block
    } // hier wird lok2 aus dem Speicher gelöscht...
}

```

¹ <http://de.wikibooks.org/wiki/Benutzer%3AProg>

```
} // ...und hier lokl
```

Hinweis

Bei Schleifen ist zu beachten, dass Variablen, deren Deklaration im Schleifenrumpf steht, bei jedem Durchlauf neu initialisiert werden.

```
while (1) {
    int var = 0;
    std::cout << var << std::endl; // die Ausgabe ist hier immer 0
    ++var;
}
```

17.2.2 Statische Variablen

Statische Variablen (auch statische Klassenmember) werden wie globale zu Beginn des Programms im Speicher angelegt und bei seinem Ende wieder daraus entfernt. Der Unterschied zu einer globalen Variable wird weiter unten auf dieser Seite im Teil über Sichtbarkeit geklärt.

17.2.3 Dynamisch erzeugte Variablen

Eine Variable, die mittels dem `new` Operator angefordert wird, ist dynamisch. Sie existiert so lange bis sie durch einen Aufruf von `delete` wieder gelöscht wird.

Hinweis

Der Aufruf von `delete` ist die einzige Möglichkeit den von einer dynamisch erzeugten Variable belegten Speicher wieder frei zu geben. Geschieht dies nicht, so kann es leicht zu einem Speicherleck kommen.

Lesen Sie bitte auch das Kapitel über den `new` und `delete`^a um Informationen über deren Verwendung zu erhalten.

^a <http://de.wikibooks.org/wiki/C%2B%2B-Programmierung%2F%20Speicherverwaltung%2F%20new%20und%20delete>

17.2.4 Objekte und Membervariablen

Objekte werden wie normale Variablen gehandhabt, d. h. sie können global, lokal, statisch oder dynamisch erzeugt sein. Ihre Member haben die gleiche Lebensdauer wie sie selbst. Eine Ausnahme bilden statische Klassenvariablen, die von Anfang bis Ende des Programmablaufes im Speicher vorhanden sind.

17.3 Sichtbarkeit

17.3.1 Allgemein

Um überhaupt die Chance zu haben mit einer Variablen zu arbeiten, muss diese im Quelltext bereits deklariert worden sein. Folgendes Codestück ist also falsch und führt zu einem Fehler.

```
// ...
var = 34; // Fehler z.B. "symbol var not found"
int var;
// ...
```

Thema wird später näher erläutert...

Das Prinzip der Datenkapselung in der objektorientierten Programmierung finden Sie im Abschnitt über Klassen^a.

^a <http://de.wikibooks.org/wiki/C%2B%2B-Programmierung%2F%20Inhaltsverzeichnis%23Anker%3AEigene%20Datentypen%20definieren>

17.3.2 Gültigkeitsbereiche und deren Schachtelung

Jede Variable gehört zu einem bestimmten Gültigkeitsbereich (engl. *scope*). Dieser legt fest, wann eine Variable von uns „gesehen“ und damit benutzt werden kann. Vereinfacht gesagt bildet jedes Paar aus geschweiften Klammern (`{ }`) einen eigenen Definitionsbereich. Dazu gehören beispielsweise `if`, `else`, Schleifen und Funktionen. Diese unterschiedlichen Bereiche sind nun ineinander geschachtelt ähnlich wie die berühmten Matrjoschka-Puppen² mit dem Unterschied, dass die Definitionsbereiche nicht „kleiner“ werden und dass es mehrere „nebeneinander“ geben kann.

```
// das hier gehört zum globalen Bereich

int func() { // hier beginnt der Bereich der Funktion func...
    return 0;
    // ... und ist hier auch schon wieder beendet
}

int bar(int val) { // val gehört zum Definitionsbereich "bar"
    if (val == 7) { // diese if-Anweisung hat auch ihren eigenen Gültigkeitsbereich...
        int ich_gehoer_zum_if;
    } // ... der hier zu Ende ist
} // bar ende
```

² <http://de.wikipedia.org/wiki/Matrjoschka>

17.3.3 Welche Variablen sind sichtbar?

Jetzt ist es leicht zu bestimmen mit welchen Variablen wir an einer bestimmten Stelle im Programmcode arbeiten können: Es sind diejenigen, die entweder dem derzeitigen oder einem Gültigkeitsbereich auf einer höheren Ebene angehören. Wenn wir uns erneut das Beispiel der Puppen vor Augen halten, so wird klar was hiermit gemeint ist.

Beispiel:

```
int out;
{
    int inner_1;
}

{
    int inner_2;
    // an dieser Stelle könnten wir sowohl auf out als auch auf inner_
2 zugreifen, nicht jedoch
auf inner_1;
}
```

Zusätzlich gilt noch, dass von zwei Variablen gleichen Namens nur auf die weiter innere liegende zugegriffen werden kann. Außerdem wird die gleichnamige innere Variable im selben Speicherbereich agieren.

```
int var=9;

{
    int var=3;
    std::cout << var << std::endl; // die Ausgabe ist 3
}
```

18 Rekursion

Jede Funktion kann sowohl andere Funktionen als auch sich selbst aufrufen. Ein solcher Selbstaufruf wird auch rekursiver Aufruf genannt. Das dahinter stehende Konzept bezeichnet man entsprechend als Rekursion.

Eine Ausnahme von dieser Regel bildet wiederum die Funktion `main()`. Sie darf ausschließlich vom Betriebssystem aufgerufen werden, also weder von einer anderen Funktion, noch aus sich selbst heraus.

Eine rekursive Problemlösung ist etwas langsamer und speicheraufwendiger als eine iterative Variante (also mit Schleifen). Dafür ist der Code allerdings auch kompakter und ein „intelligenter“ Compiler ist meist in der Lage, eine Rekursion in eine Iteration umzuwandeln um somit die Nachteile aufzuheben. Sie sollten also keine Scheu haben ein Problem mit Rekursion zu lösen, wenn Sie so schneller ein richtiges Ergebnis erhalten als beim Schreiben einer iterativen Variante. Sollten dadurch im Laufe der Entwicklung eines Programms Geschwindigkeits- oder Speichernachteile auftreten, so können Sie die Funktion immer noch durch eine iterativ arbeitende ersetzen.

18.1 Fakultät

Als erstes einfaches Beispiel einer rekursiven Problemlösung nehmen wir die Berechnung der Fakultät¹. Da die Fakultät für negative und nicht ganze Zahlen nicht definiert ist, benutzen wir als Datentyp `unsigned int`:

```
#include <iostream>          // Für std::cin und std::cout

unsigned int fakultaet(unsigned int zahl) {
    if (zahl <= 1) {
        return 1; // Die Fakultät von 0 und 1 ist als 1 definiert.
    }

    return fakultaet(zahl - 1) * zahl;
}

int main() {
    unsigned int zahl;

    std::cout << "Bitte Zahl eingeben: ";
    std::cin >> zahl;          // Zahl einlesen
}
```

¹ <http://de.wikipedia.org/wiki/Fakultät%20%28Mathematik%29>

```
std::cout << "Die Fakultät von " << zahl << // Antwort ausgeben
    " ist " << fakultaet(zahl) << "!" << std::endl;
}
```

Ausgabe

```
Bitte Zahl eingeben: <eingabe>4</eingabe>
Die Fakultät von 4 ist 24!
```

Genau wie bei einer Schleife, ist auch bei einer Rekursion eine Abbruchbedingung definiert (also erforderlich) und genau wie bei einer Schleife würde ohne Abbruchbedingung eine Endlosrekursion auftreten, analog zur Endlosschleife. So eine Endlosschleife bezeichnet man auch als infiniten Regress. Wenn der Wert der Variablen `zahl` kleiner, oder gleich eins ist, so wird eins zurückgegeben, andernfalls wird weiter rekursiv aufgerufen. Eine iterative Variante für das gleiche Problem könnte folgendermaßen aussehen:

```
unsigned int fakultaet(unsigned int zahl) {
    unsigned int wert = 1;

    for (unsigned int i = 2; i <= zahl; ++i) {
        wert *= i;
    }

    return wert;
}
```

18.2 Fibonacci-Zahlen

Als zweites Beispiel wollen wir Fibonacci-Zahlen² ausrechnen.

```
#include <iostream>

unsigned int fibonacci(unsigned int zahl) {
    if (zahl == 0) { // Die Fibonacci-Zahl von null ist null
        return 0;
    }

    if (zahl == 1) { // Die Fibonacci-Zahl von eins ist eins
        return 1;
    }

    // Ansonsten wird die Summe der zwei vorherigen Fibonacci-Zahlen zurückgegeben
    return fibonacci(zahl - 1) + fibonacci(zahl - 2);
}
```

² <http://de.wikipedia.org/wiki/Fibonacci-Folge>

```
int main() {
    unsigned int zahl;

    std::cout << "Bitte Zahl eingeben: ";
    std::cin >> zahl;                // Zahl einlesen
    std::cout << "Die Fibonacci-Zahl von " << zahl << // Antwort ausgeben
        " ist " << fibonacci(zahl) << "!" << std::endl;
}

```

Ausgabe

```
Bitte Zahl eingeben: <eingabe>12</eingabe>
Die Fibonacci-Zahl von 12 ist 144!
```

Die iterative Entsprechung sieht folgendermaßen aus:

```
unsigned int fibonacci(unsigned int zahl) {
    if (zahl == 0) { // Die Fibonacci-Zahl von null ist null
        return 0;
    }

    if (zahl == 1) { // Die Fibonacci-Zahl von eins ist eins
        return 1;
    }

    unsigned int ret;
    unsigned int h1 = 0;
    unsigned int h2 = 1;

    for (unsigned int i = 1; i < zahl; ++i) {
        ret = h1 + h2; // Ergebnis ist die Summe der zwei vorhergehenden Fibonacci-Zahlen
        h1 = h2;      // und den beiden Hilfsvariablen die
        h2 = ret;     // neuen vorhergehenden Fibonacci-Zahlen
    }

    return ret;
}

```

Bei vielen komplexen Problemen eignet sich Rekursion oft besser zur Beschreibung, als eine iterative Entsprechung. Aus diesem Grund trifft man das Konzept der Rekursion in der Programmierung recht häufig an.

18.3 Merge sort

Merge sort ist ein Beispiel für eine Funktion, bei der Rekursion sinnvoll eingesetzt wird. Die Idee ist: Um ein Array zu sortieren, sortiere erst die erste Hälfte, dann die zweite Hälfte, und dann füge die beiden Teile zusammen (merge). Der folgende Code implementiert Merge sort für `int`-Arrays.

Sie erwartet ein Array, den ersten Index des zu sortierenden Bereichs, und den Index auf das erste Element nach dem zu sortierenden Bereich. Da die genaue Implementierung des Merge-Schritts hier nicht von Interesse ist, wird einfach angenommen, dass dafür bereits eine Funktion `merge` existiert.

```
void mergesort(int array[], int begin, int end)
{
    int mid = begin + (end-begin)/2;    // Mitte des Feldes bestimmen

    mergesort(array, begin, mid);      // Linke Hälfte
    mergesort(array, mid, end);        // Rechte Hälfte

    merge(array, begin, mid, end);
}
```

19 Zeiger

19.1 Grundlagen zu Zeigern

Zeiger (engl. *pointers*) sind Variablen, die als Wert die Speicheradresse einer anderen Variable enthalten.

Jede Variable wird in C++ an einer bestimmten Position im Hauptspeicher abgelegt. Diese Position nennt man *Speicheradresse* (engl. *memory address*). C++ bietet die Möglichkeit, die Adresse jeder Variable zu ermitteln. Solange eine Variable gültig ist, bleibt sie an ein und derselben Stelle im Speicher.

Am einfachsten vergegenwärtigt man sich dieses Konzept anhand der globalen Variablen. Diese werden außerhalb aller Funktionen und Klassen deklariert und sind überall gültig. Auf sie kann man von jeder Klasse und jeder Funktion aus zugreifen. Über globale Variablen ist bereits zur Compilierzeit bekannt, wo sie sich innerhalb des Speichers befinden (also kennt das Programm ihre Adresse).

Zeiger sind nichts anderes als normale Variablen. Sie werden deklariert (und definiert), besitzen einen Gültigkeitsbereich, eine Adresse und einen Wert. Dieser Wert, der *Inhalt* der Zeigervariable, ist aber nicht wie in unseren bisherigen Beispielen eine Zahl, sondern die Adresse einer anderen Variable oder eines Speicherbereichs. Bei der Deklaration einer Zeigervariable wird der Typ der Variable festgelegt, auf den sie verweisen soll.

```
#include <iostream>

int main() {
    int Wert;           // eine int-Variable
    int *pWert;        // eine Zeigervariable, zeigt auf einen int
    int *pZahl;        // ein weiterer "Zeiger auf int"

    Wert = 10;         // Zuweisung eines Wertes an eine int-Variable

    pWert = &Wert;     // Adressoperator '&' liefert die Adresse einer Variable
    pZahl = pWert;     // pZahl und pWert zeigen jetzt auf dieselbe Variable
}
```

Der *Adressoperator* & kann auf jede Variable angewandt werden und liefert deren Adresse, die man einer (dem Variablentyp entsprechenden) Zeigervariablen zuweisen kann. Wie im Beispiel gezeigt, können Zeiger gleichen Typs einander zugewiesen werden. Zeiger verschiedenen Typs bedürfen einer Typumwandlung. Die Zeigervariablen `pWert` und `pZahl` sind an verschiedenen Stellen im Speicher abgelegt, nur die Inhalte sind gleich.

Wollen Sie auf den Wert zugreifen, der sich hinter der im Zeiger gespeicherten Adresse verbirgt, so verwenden Sie den *Dereferenzierungsoperator* `*`.

```
{
    *pWert += 5;
    *pZahl += 8;

    std::cout << "Wert = " << Wert << std::endl;
}
```

Ausgabe

```
Wert = 23
```

Man nennt das *den Zeiger dereferenzieren*. Im Beispiel erhalten Sie die Ausgabe `Wert = 23`, denn `pWert` und `pZahl` verweisen ja beide auf die Variable `Wert`.

Um es noch einmal hervorzuheben: Zeiger auf Integer (`int`) sind selbst keine Integer. Den Versuch, einer Zeigervariablen eine Zahl zuzuweisen, beantwortet der Compiler mit einer Fehlermeldung oder mindestens einer Warnung. Hier gibt es nur eine Ausnahme: die Zahl 0 darf jedem beliebigen Zeiger zugewiesen werden. Ein solcher *Nullzeiger* zeigt nirgendwohin. Der Versuch, ihn zu dereferenzieren, führt zu einem Laufzeitfehler.

Thema wird später näher erläutert...

Der Sinn von Zeigern erschließt sich vor allem Anfängern nicht unmittelbar. Das ändert sich allerdings schnell, sobald der dynamische Speicher^a besprochen wird.

^a <http://de.wikibooks.org/wiki/C%2B%2B-Programmierung%2F%20Inhaltsverzeichnis%23Immer%20n%FCtzlich%20%20%13%20Speicher>

19.2 Zeiger und `const`

Das Schlüsselwort `const` kann auf zweierlei Arten in Verbindung mit Zeigern genutzt werden:

- Um den Wert, auf den der Zeiger zeigt, konstant zu machen,
- Um den Zeiger selbst konstant zu machen.

Im ersteren Fall kann der Zeiger im Laufe seines Lebens auf verschiedene Objekte zeigen, diese Werte können dann allerdings (über diesen Zeiger) nicht geändert werden. Im zweiten Fall kann der Zeiger nicht auf eine andere Adresse "umgebogen" werden. Der Wert an dieser Stelle kann allerdings verändert werden. Natürlich sind auch beide Varianten in Kombination möglich.

```
int          Wert1;          // eine int-Variable
int          Wert2;          // noch eine int-Variable
int const *  p1Wert = &Wert1; // Zeiger auf konstanten int
int * const  p2Wert = &Wert1; // konstanter Zeiger auf int
int const * const p3Wert = &Wert1; // konstanter Zeiger auf konstanten int

p1Wert = &Wert2; // geht
```

```

*p1Wert = Wert2; // geht nicht, int konstant

p2Wert = &Wert2; // geht nicht, Zeiger konstant
*p2Wert = Wert2; // geht

p3Wert = &Wert2; // geht nicht, Zeiger konstant
*p3Wert = Wert2; // geht nicht, int konstant

```

Wie Sie sich sicher noch erinnern, gehört `const` immer zu dem was links von ihm steht. Es sei denn links steht nichts mehr, dann gehört es zu dem was rechts davon steht.

19.3 Zeiger und Funktionen

Wenn Sie einen Zeiger als Parameter an eine Funktion übergeben, können Sie den Wert an der übergebenen Adresse ändern. Eine Funktion, welche die Werte zweier Variablen vertauscht, könnte folgendermaßen implementiert werden:

```

#include <iostream>

void swap(int *wert1, int *wert2) {
    int tmp;
    tmp    = *wert1;
    *wert1 = *wert2;
    *wert2 = tmp;
}

int main() {
    int a = 7, b = 9;

    std::cout << "a: " << a << ", b: " << b << std::endl;
    swap(&a, &b);
    std::cout << "a: " << a << ", b: " << b << std::endl;
}

```

Ausgabe

```

a: 7, b: 9
a: 9, b: 7

```

Diese Funktion hat natürlich einige Schwachstellen. Beispielsweise stürzt sie ab, wenn ihr ein Nullzeiger übergeben wird. Aber sie zeigt, dass es mit Zeigern möglich ist, den Wert einer Variable außerhalb der Funktion zu verändern. In Kürze werden Sie sehen, dass sich dieses Beispiel besser mit Referenzen lösen lässt.

Funktionen, die einen Zeiger auf einen konstanten Datentyp erwarten, können auch mit einem Zeiger auf einen nicht-konstanten Datentyp aufgerufen werden. Das folgende Minimalbeispiel soll dies zeigen:

```
// Funktion, die einen Zeiger auf einen konstanten int erwartet
void function(int const* parameter){}

int main() {
    int* zeiger;      // Zeiger auf nicht-konstanten int
    function(zeiger); // Funktioniert
}
```

19.3.1 Probleme mit Zeigern auf Zeiger auf konstante Daten

Ein Problem, über das die meisten Programmierer irgendwann stolpern, ist die Übergabe eines Zeigers auf einen Zeiger auf nicht-konstante Daten an eine Funktion, die einen Zeiger auf einen Zeiger auf konstante Daten erwartet. Da sich das `const` hier nicht direkt auf die Daten des Zeigers bezieht, sondern erst auf die Daten des Zeigers, auf den der Zeiger zeigt, erlaubt der Compiler die Übergabe nicht. Die Lösung des Problems ist relativ einfach: teilen Sie dem Compiler mittels `const_cast` explizit mit, dass Sie die Umwandlung vornehmen möchten.

```
// Funktion, die einen Zeiger auf einen Zeiger einen konstanten int erwartet
void function(int const** parameter){}

int main() {
    int** zeiger;      // Zeiger auf Zeiger auf nicht-konstanten int
    function(zeiger); // Fehler: Typen sind inkompatibel

    // Lösung: explizites hinzucastern der Konstantheit
    function(const_cast< int const** >(zeiger));
}
```

In den meisten Fällen werden Sie einen Parametertyp haben, der etwa die Form `Typ const*const*` hat und Daten übergeben will, deren Typ `Typ**` lautet. Die Umwandlung der Konstantheit der Daten unmittelbar auf die der Zeiger zeigt, dass der Compiler dies automatisch vornehmen kann. Alle anderen Umwandlungen müssen Sie explizit mittels `const_cast` erledigen. Es ist somit egal, ob Sie `const_cast< int const*const* >` oder nur `const_cast< int const** >` für die explizite Umwandlung angeben.

19.4 Zeigerarithmetik

Zeiger sind keine Zahlen. Deshalb sind einige arithmetischen Operationen auf Zeiger nicht anwendbar, und für die übrigen gelten andere Rechenregeln als in der Zahlenarithmetik. C++ kennt die Größe des Speicherbereichs, auf den ein Zeiger verweist. Inkrementieren (oder Dekrementieren) verändert die referenzierte Adresse unter Berücksichtigung dieser Speichergröße. Das folgende Beispiel soll den Unterschied zwischen Zahlen und Zeigerarithmetik verdeutlichen:

```

#include <iostream>

int main() {
    std::cout << "Zahlenarithmetik" << std::endl;

    int a = 1;    // a wird 1
    std::cout << "a: " << a << std::endl;

    a++;        // a wird 2
    std::cout << "a: " << a << std::endl;

    std::cout << "Zeigerarithmetik" << std::endl;

    int *p = &a; // Adresse von a an p zuweisen

    std::cout << "p verweist auf: " << p << std::endl;
    std::cout << " Größe von int: " << sizeof(int) << std::endl;

    p++;
    std::cout << "p verweist auf: " << p << std::endl;
}

```

Ausgabe

```

Zahlenarithmetik
a: 1
a: 2
Zeigerarithmetik
p verweist auf: 0x7fff3aa60090
Größe von int: 4
p verweist auf: 0x7fff3aa60094

```

Wie Sie sehen, erhöht sich der Wert des Zeigers nicht um eins, sondern um vier, was genau der Größe des Typs entspricht, auf den er zeigt: `int`. Auf einer Plattform, auf der `int` eine andere Größe hat, würde natürlich entsprechend dieser Größe gezählt werden. Im nächsten Beispiel sehen Sie, wie ein Zeiger auf eine weitere Zeigervariable verweist, welche ihrerseits auf einen `int`-Wert zeigt.

```

#include <iostream>

int main() {
    int a = 1;
    int *p = &a; // Adresse von a an p zuweisen
    int **pp = &p; // Adresse von p an pp zuweisen
    std::cout << "pp verweist auf: " << pp << std::endl;
    std::cout << " Größe von int*: " << sizeof(int*) << std::endl;
    ++pp;
    std::cout << "pp verweist auf: " << pp << std::endl;
}

```

Ausgabe

```
pp verweist auf: 0x7fff940cb6f0
Größe von int*: 8
pp verweist auf: 0x7fff940cb6f8
```

Wie Sie sehen hat ein Zeiger auf `int` im Beispiel eine Größe von 8 Byte. Die Größe von Datentypen ist allerdings architektur-, compiler- und systembedingt. `pp` ist vom Typ „Zeiger auf Zeiger auf `int`“, was sich dann in C++ als `int**` schreibt. Um also auf die Variable "hinter" diesen beiden Zeigern zuzugreifen, muss man `**pp` schreiben.

Es spielt keine Rolle, ob man in Deklarationen `int* p;` oder `int *p;` schreibt. Einige Programmierer schreiben den Stern direkt hinter den Datentyp (`int* p`), andere schreiben ihn direkt vor Variablennamen (`int *p`), und wieder andere lassen zu beidem ein Leerzeichen (`int * p`). In diesem Buch wird die Konvention verfolgt, den Stern direkt vor Variablennamen zu schreiben, wenn einer vorhanden ist (`int *p`), andernfalls wird er direkt nach dem Datentyp geschrieben (`int*`).

19.5 Negativbeispiele

Zur Verdeutlichung zwei Beispiele, die *nicht funktionieren*, weil sie vom Compiler nicht akzeptiert werden:

```
int * pWert;
int Wert;
pWert = Wert; // dem Zeiger kann kein int zugewiesen werden
Wert = pWert; // umgekehrt natürlich auch nicht
```

Zeigervariablen erlauben als Wert nur Adressen auf Variablen. Daher kann einer Zeigervariable wie in diesem Beispiel kein Integer-Wert zugewiesen werden.

Im Folgenden wollen wir den Wert, auf den `pWert` zeigt, inkrementieren. Einige der Beispiele bearbeiten die Adresse, die `pWert` enthält. Erst das letzte Beispiel verändert tatsächlich den Wert, auf den `pWert` zeigt. Beachten Sie, dass jede Codezeile ein Einzelbeispiel ist.

```
int Wert = 0;
int *pWert = &Wert; // pWert zeigt auf Wert

pWert += 5; // ohne Dereferenzierung (*pWert) verändert man die Adresse, auf die
           // der Zeiger verweist, und nicht deren Inhalt
std::cout << pWert; // es wird nicht die Zahl ausgegeben, auf die pWert
                  // zeigt, sondern deren (veränderte) Adresse
printf("Wert enthält: %d", pWert); // gleiche Ausgabe

pWert++; // Diese Operation verändert wiederum die Adresse, da nicht dereferenziert
         // wird.
*pWert++; // Auf diese Idee kommt man als Nächstes. Doch auch das hat nicht den
```

```

// gewünschten Effekt. Da der (Post-)Inkrement-Operator vor dem
Dereferenzierungs-
// operator ausgewertet wird, verändert sich wieder die Adresse.
(*pWert)++; // Da der Ausdruck in der Klammer zuerst ausgewertet wird, erreichen wir
// diesmal den gewünschten Effekt: Eine Änderung des Wertes.

```

19.6 void-Zeiger (anonyme Zeiger)

Eine besondere Rolle spielen die „Zeiger auf void“, die so genannten *generischen Zeiger*. Einer Zeigervariable vom Typ `void*` kann jeder beliebige Zeiger zugewiesen werden. `void`-Zeiger werden in der Sprache C beispielsweise bei der dynamischen Speicherverwaltung verwendet. In C++ kommt man weitgehend ohne sie aus. Vermeiden Sie Zeiger auf `void`, wenn Sie eine andere Möglichkeit haben. Ein Objekt kann nicht vom Typ `void` sein. Entsprechend ist es auch nicht möglich, einen `void`-Zeiger zu dereferenzieren. Das folgende kleine Codebeispiel ist praxisfern. Aber wie schon erwähnt gibt es in C++ nur noch sehr selten eine Notwendigkeit für `void`-Zeiger, und keine dieser seltenen Situationen könnte an dieser Stelle im Buch bereits erklärt werden. Die meisten C++-Programmierer werden in Ihrem gesamten Leben nicht in die Verlegenheit kommen, einen `void`-Zeiger wirklich verwenden zu müssen.

```

#include <iostream>

int main() {
    int value = 1;
    int* pointer = &value; // zeigt auf value

    void* void_pointer;
    void_pointer = pointer; // void_pointer zeigt auf pointer

```

Sie können jetzt nicht ohne Weiteres auf `*void_pointer` zugreifen, um an den Wert von `value` zu kommen. Da es sich um einen Zeiger vom Typ `void` handelt, muss man diesen erst umwandeln (engl. *to cast*). In diesem Fall nach `int*`.

```

    std::cout << *reinterpret_cast< int* >(void_pointer) << std::endl;
}

```

Der Ablauf dieser letzten Zeile noch einmal im Detail:

1. Zeiger `void_pointer` ist vom Typ `void*` und zeigt auf `value`
2. `reinterpret_cast< int* >`, Zeiger ist vom Typ `int*`
3. Zeiger dereferenzieren (`*`), um Wert (vom Typ `int`) zu erhalten
4. Wert ausgeben

19.7 Funktionszeiger

Zeiger können nicht nur auf Variablen, sondern auch auf Funktionen verweisen. Die Deklaration eines solchen Funktionszeigers sieht im ersten Augenblick etwas schwierig aus, aber sie ist dennoch leicht zu merken. Sie schreiben einfach den Prototypen der Funktion, auf die ihr Zeiger verweisen soll und geben statt des Funktionsnamens den Variablennamen an. Selbigen stellen Sie einen Stern voran, um klar zu machen, dass es sich um einen Zeiger handelt, und um dem Compiler zu vermitteln, dass der Stern nicht zum Rückgabebetyp gehört, fassen Sie ihn und den Variablennamen in Klammern ein. Das folgende kleine Beispiel zeigt die Verwendung:

```
#include <iostream>

int multiplication(int a, int b){
    return a*b;
}

int division(int a, int b){
    return a/b;
}

int main(){
    int (*rechenoperation)(int, int) = 0; // Anlegen eines Funktionszeigers, Initialisierung mit 0

    rechenoperation = &multiplication;
    std::cout << (*rechenoperation)(40, 8) << std::endl;
    rechenoperation = &division;
    std::cout << (*rechenoperation)(40, 8) << std::endl;
}
```

Ausgabe

```
320
5
```

Man liest: `rechenoperation` ist ein Zeiger auf eine Funktion, die zwei `ints` übernimmt und einen `int` zurückgibt. Im Kapitel über Felder wird eine allgemeingültige Regel für das Lesen komplexer Datentypen vorgestellt. Wie Sie sehen wird der Zeigervariable nacheinander die Adresse zweier Funktionen zugewiesen, die dem Typ der Zeigervariable entsprechen. Eine für den Zeiger gültige Funktion muss also zwei `ints` übernehmen und einen `int` zurückgeben.

Um die Adresse der Funktion zu erhalten, müssen Sie den Adressoperator auf den Funktionsnamen anwenden. Beachten Sie, dass die Klammern, die Sie zum Aufruf einer Funktion *immer* setzen müssen, hier keinesfalls gesetzt werden dürfen. `&multiplication()` würde Ihnen die Adresse des von `multiplication()` zurückgelieferten Objekts beschaffen. Es sei auch darauf hingewiesen, dass der Adressoperator nicht zwingend zum Ermitteln der Funktionsadresse notwendig ist. Sie sollten ihn aus Gründen der Übersicht allerdings immer mitschreiben.

Gleiches gilt bei der Dereferenzierung: ein expliziter Stern vor dem Funktionszeiger macht deutlich, dass es sich um eine Zeigervariable handelt. Während der Fehlersuche kann dies beim Lesen des Codes erheblich helfen. Die Klammern sind, wie schon bei der Deklaration, nötig, um dem Compiler mitzuteilen, dass sich der Stern nicht auf den Rückgabewert, sondern auf die Funktion bezieht.

```
int x;
x = (*multiplication)(40, 8); // ruft multiplication() auf und weist x den Rückgabewert zu
x = multiplication(40, 8);    // alternative (nicht empfohlene) Syntax
```

In C werden Funktionszeiger oft für generische Funktionen verwendet, wofür es in C++ mit den Templates (auf deutsch etwa „Vorlagen“) eine bessere Lösung gibt. Insbesondere werden in C++ statt Funktionszeigern auch oft *Funktoren* verwendet, welche aber erst später vorgestellt werden. Vorweggenommen sei an dieser Stelle bereits, dass ein Funktor etwas mehr Schreibaufwand benötigt als ein Funktionszeiger, dafür aber auch einiges kann, was mit einfachen Funktionszeigern nicht möglich ist.

Tip

Verwenden Sie `typedef` um komplizierte Typen zu vereinfachen. Für unseren Zeigertyp könnte man etwa auch schreiben:

```
int (*Zeigertyp)(int, int); // Zeigertyp als Alias für den Funktionszeigertyp
Zeigertyp rechenoperation = 0; // Anlegen eines Funktionszeigers, Initialisierung mit 0
```

Thema wird später näher erläutert...

Im Zusammenhang mit Klassen^a werden uns weitere Arten von Zeigern begegnen:

- Zeiger auf statische Datenelemente
- Zeiger auf Elementfunktionen.

Beide werden Sie zu einem späteren Zeitpunkt noch kennenlernen. Auch im Abschnitt „Speicherverwaltung“^b werden Ihnen Zeiger noch einmal begegnen.

^a <http://de.wikibooks.org/wiki/C%2B%2B-Programmierung%2F%20Inhaltsverzeichnis%23Anker%3AEigene%20Datentypen%20definieren>

^b <http://de.wikibooks.org/wiki/C%2B%2B-Programmierung%2F%20Inhaltsverzeichnis%23Anker%3ASpeicherverwaltung>

20 Referenzen

20.1 Grundlagen zu Referenzen

Referenzen sind interne Zeiger auf Variablen. Sie werden also genau so verwendet wie gewöhnliche Variablen, verweisen jedoch auf das Objekt, mit dem sie initialisiert wurden. Die Zeigerverwendung wird vor dem Programmierer verborgen.

```
int a = 1; // eine Variable
int &r = a; // Referenz auf die Variable a

std::cout << "a: " << a << " r: " << r << std::endl;
++a;
std::cout << "a: " << a << " r: " << r << std::endl;
++r;
std::cout << "a: " << a << " r: " << r << std::endl;
```

Ausgabe

```
a: 1 r: 1
a: 2 r: 2
a: 3 r: 3
```

Wie Sie im Beispiel sehen, sind `a` und `r` identisch. Gleiches können Sie natürlich auch mit einem Zeiger erreichen, auch wenn bei einem Zeiger die Syntax etwas anders ist als bei einer Referenz.

Im Beispiel wurde die Referenz auf `int r`, mit dem `int a` *initialisiert*. Beachten Sie, dass die Initialisierung einer Referenzvariablen nur beim Anlegen erfolgen kann, danach kann ihr Wert nur noch durch eine Zuweisung geändert werden. Daraus folgt, dass eine Referenz immer initialisiert werden muss und es nicht möglich ist, eine Referenzvariable auf ein neues Objekt verweisen zu lassen:

```
int a = 10; // eine Variable
int b = 20; // noch eine Variable
int &r = a; // Referenz auf die Variable a

std::cout << "a: " << a << " b: " << b << " r: " << r << std::endl;
++a;
std::cout << "a: " << a << " b: " << b << " r: " << r << std::endl;
r = b; // r zeigt weiterhin auf a, r (und somit a) wird 20 zugewiesen
std::cout << "a: " << a << " b: " << b << " r: " << r << std::endl;
```

Ausgabe

```
a: 10 b: 20 r: 10
a: 11 b: 20 r: 11
a: 20 b: 20 r: 20
```

Wie Sie sehen, ist es nicht möglich, `r` als Alias für `b` zu definieren, nachdem es einmal mit `a` initialisiert wurde. Die Zuweisung bewirkt genau das, was auch eine Zuweisung von `b` an `a` bewirkt hätte. Dass eine Referenz wirklich nichts weiter ist als ein Aliasname wird umso deutlicher, wenn man sich die Adressen der Variablen aus dem ersten Beispiel ansieht:

```
int a = 1; // eine Variable
int &r = a; // Referenz auf die Variable a

std::cout << "a: " << &a << " r: " << &r << std::endl;
```

Ausgabe

```
a: 0x7fffbf623c54 r: 0x7fffbf623c54
```

Wie Sie sehen, sind die Adressen identisch.

20.2 Anwendung von Referenzen

Vielleicht haben Sie sich bereits gefragt, wofür Referenzen nun eigentlich gut sind, schließlich könnte man ja auch einfach die Originalvariable benutzen.

20.2.1 Selbstdefinierte Referenzen

Referenzen bieten in einigen Anwendungsfällen eine Beschleunigung und bessere Lesbarkeit der Quelltexte. Sie müssen initialisiert werden.

```
#include <iostream>

int main(){
    unsigned int const x = 2, y = 3, z = 4;
    unsigned int zahlen_array[x][y][z] = {
        { { 0, 1, 2, 3}, { 4, 5, 6, 7}, { 8, 9, 10, 11} },
        { {12, 13, 14, 15}, {16, 17, 18, 19}, {20, 21, 22, 23} }
    };
    for(unsigned int a = 0; a < x; ++a){
        for(unsigned int b = 0; b < y; ++b){
            for(unsigned int c = 0; c < z; ++c){
                // ref wird als Referenz auf zahlen_array[a][b][c] initialisiert
                unsigned int& ref = zahlen_array[a][b][c];
                // entspricht 'zahlen_array[a][b][c] *= 2;'
                ref *= 2;
            }
        }
    }
}
```

```

        // entspricht 'zahlen_array[a][b][c] += a * b * c;'
        ref += a * b * c;
        // entspricht 'std::cout << zahlen_array[a][b][c] << std::endl;'
        std::cout << ref << ", ";
    }
    std::cout << std::endl;
}
std::cout << std::endl;
}
}

```

Ausgabe

```

0, 2, 4, 6,
8, 10, 12, 14,
16, 18, 20, 22,

24, 26, 28, 30,
32, 34, 36, 38,
40, 42, 44, 46,

```

Bei mehrfacher Verwendung, kann eine Referenz Ihnen viel Tipparbeit ersparen und vor allem erhöht sie die Übersichtlichkeit des Quellcodes. Außerdem kann diese Vorgehensweise die Performance verbessern, da der Zugriff auf Daten in Klassen oder Feldern, durch eine Referenzdefinition vereinfacht wird. Bei obigem Beispiel wird zur Laufzeit, im Arbeitsspeicher, bei jeder Verwendung von `zahlen_array[a][b][c]` zuerst der Speicherort der einzelnen Zahl berechnet, dabei müssen die Inhalts-, Feldgrößen und Offsets innerhalb des Felds berücksichtigt werden. An anderen Stellen mag dies alles mit STL-Containerklassen und verschachtelten Methoden- und Operatoraufrufen erfolgen.

Dies alles können Sie dem Prozessor nicht ersparen. Sie können aber dafür sorgen, dass es, pro Schritt, nur einmal vorkommt. Die Verwendung einer Referenz ergibt daher Sinn, sobald Sie `zahlen_array[a][b][c]` mehr als einmal verwenden. Eine Referenz ist intern meist mit einem Zeiger implementiert. Es sei allerdings darauf hingewiesen, dass der Compiler in vielen Fällen diese Optimierung auch selbst vornehmen kann, daher ist ein Performancegewinn nicht zwingend.

20.2.2 Call-By-Reference

Möglicherweise erinnern Sie sich aber auch noch, dass im Kapitel „Prozeduren und Funktionen“ die Wertübergabe als Referenz (*call-by-reference*) vorgestellt wurde. Darauf wird nun genauer eingegangen.

Referenzen bieten genau wie Zeiger die Möglichkeit, den Wert einer Variable außerhalb der Funktion zu ändern. Im Folgenden sehen Sie die im Kapitel über Zeiger vorgestellte Funktion `swap()` mit Referenzen:

```

#include <iostream>

void swap(int &wert1, int &wert2) {

```

```
int tmp;
tmp = wert1;
wert1 = wert2;
wert2 = tmp;
}

int main() {
    int a = 7, b = 9;

    std::cout << "a: " << a << ", b: " << b << "\n";

    swap(a, b);

    std::cout << "a: " << a << ", b: " << b << "\n";
    return (0);
}
```

Ausgabe

```
a: 7, b: 9
a: 9, b: 7
```

Diese Funktion bietet gegenüber der Zeigervariante zwei Vorteile. Die Syntax ist einfacher und es ist nicht möglich, so etwas wie einen Nullzeiger zu übergeben. Um diese Funktion zum Absturz zu bewegen, ist schon einige Mühe nötig.

20.2.3 const-Referenzen

Referenzen auf konstante Variablen spielen in C++ eine besondere Rolle. Eine Funktion, die eine Variable übernimmt, kann genauso gut auch eine Referenz auf eine konstante Variable übernehmen. Folgendes Beispiel soll dies demonstrieren:

```
#include <iostream>

void ausgabe1(int wert) {
    std::cout << "wert: " << wert << "\n";
}

void ausgabe2(int const &wert) {
    std::cout << "wert: " << wert << "\n";
}

int main() {
    ausgabe1(5);
    ausgabe2(5);
    return (0);
}
```

Ausgabe

```
ausgabe1 wert: 5
ausgabe2 wert: 5
```

Die beiden Ausgabefunktionen sind an sich identisch, lediglich die Art der Parameterübergabe unterscheidet sich. `ausgabe1()` übernimmt einen `int`, `ausgabe2()` eine Referenz auf einen konstanten `int`. Beide Funktionen lassen sich auch vollkommen identisch aufrufen. Würde `ausgabe2()` eine Referenz auf einen nicht-konstanten `int` übernehmen, wäre ein Aufruf mit einer Konstanten, wie dem `int`-Literal `5` nicht möglich.

In Verbindung mit Klassenobjekten ist die Übergabe als Referenz auf ein konstantes Objekt sehr viel schneller, dazu erfahren Sie aber zu gegebener Zeit mehr. Für die Ihnen bereits bekannten Basisdatentypen ist tatsächlich die Übergabe als Wert effizienter.

20.2.4 Referenzen als Rückgabetyt

Referenzen haben als Rückgabewert die gleichen Vorteile wie bei der Wertübergabe. Allerdings sind sie in diesem Zusammenhang wesentlich gefährlicher. Es kann schnell passieren, dass Sie versehentlich eine Referenz auf eine lokale Variable zurückgeben. Diese Variable ist außerhalb der Funktion allerdings nicht mehr gültig, daher ist das Resultat, wenn Sie außerhalb der Funktion darauf zugreifen, undefiniert. Aus diesem Grund sollten Sie Referenzen als Rückgabewert nur verwenden wenn Sie wirklich wissen, was Sie tun.

```
#include <iostream>

// gibt die Referenz des Parameters x zurück
int &zahl(int &x) {
    return x;
}

int main() {
    int y = 3;
    zahl(y) = 5;
    std::cout << y; // Ausgabe: 5
    return y;
}
```


21 Felder

In C++ lassen sich mehrere Variablen desselben Typs zu einem *Array* (im Deutschen bisweilen auch *Datenfeld* oder *Vektor* genannt) zusammenfassen. Auf die *Elemente* des Arrays wird über einen Index zugegriffen. Bei der Definition sind der Typ der Elemente und die Größe des Arrays anzugeben. Folgende Möglichkeiten stehen zum Anlegen eines Array zur Verfügung:

```
int feld[10]; // Anlegen ohne Initialisierung
int feld[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; // Mit Initialisierung (automatisch 10
Elemente)
int feld[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; // 10 Elemente, mit Initialisierung
```

Soll das Array initialisiert werden, verwenden Sie eine Aufzählung in geschweiften Klammern, wobei der Compiler die Größe des Arrays selbst ermitteln *kann*. Es wird empfohlen, diese automatische Größenerkennung nicht zu nutzen. Wenn die Größenangabe explizit gemacht wurde, gibt der Compiler einen Fehler aus, falls die Anzahl der Initialisierungselemente nicht mit der Größenangabe übereinstimmt.

Wie bereits erwähnt, kann auf die einzelnen Elemente eines Arrays mit dem Indexoperator `[]` zugegriffen werden. Beim Zugriff auf Arrayelemente beginnt die Zählung bei 0. Das heißt, ein Array mit 10 Elementen enthält die Elemente 0 bis 9. Ein Arrayindex ist immer ganzzahlig.

```
#include <iostream>

int main() {
    int feld[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; // 10 Elemente, mit Initialisierung

    for(int i = 0; i < 10; ++i) {
        std::cout << feld[i] << " "; // Elemente 0 bis 9 ausgeben
    }
}
```

Ausgabe

```
1 2 3 4 5 6 7 8 9 10
```

Mit Arrayelementen können alle Operationen wie gewohnt ausgeführt werden.

```
feld[4] = 88;
feld[3] = 2;
feld[2] = feld[3] - 5 * feld[7 + feld[3]];
```

```
if(feld[0] < 1){
    ++feld[9];
}

for(int n = 0; n < 10; ++n){
    std::cout << feld[n] << std::endl;
}
```

Hinweis

Beachten Sie, dass der Compiler keine Indexprüfung durchführt. Wenn Sie ein nicht vorhandenes Element, z.B. `feld[297]` im Programm verwenden, kann Ihr Programm bei einigen Durchläufen unerwartet abstürzen. Zugriffe über Arraygrenzen erzeugen undefiniertes Verhalten. In modernen Desktop-Betriebssystemen kann das Betriebssystem einige dieser Bereichsüberschreitungen abfangen und das Programm abbrechen („segmentation fault“). Manchmal überschreibt man auch einfach nur den eigenen Speicherbereich in anderen Variablen, was zu schwer zu findenden Bugs führt.

21.1 Zeiger und Arrays

Der Name eines Arrays wird vom Compiler (ähnlich wie bei Funktionen) als Adresse des Arrays interpretiert. In Folge dessen haben Sie neben der Möglichkeit über den Indexoperator auch die Möglichkeit, mittels Zeigerarithmetik auf die einzelnen Arrayelemente zuzugreifen.

```
#include <iostream>

int main() {
    int feld[10] = {1,2,3,4,5,6,7,8,9,10};

    std::cout << feld[3] << "\n"; // Indexoperator
    std::cout << *(feld + 3) << "\n"; // Zeigerarithmetik
}
```

Ausgabe

```
4
4
```

Im Normalfall werden Sie mit der ersten Syntax arbeiten, es ist jedoch nützlich zu wissen, wie Sie ein Array mittels Zeigern manipulieren können. Es ist übrigens nicht möglich, die Adresse von `feld` zu ändern. `feld` verhält sich also in vielerlei Hinsicht wie ein konstanter Zeiger auf das erste Element des Arrays. Einen deutlichen Unterschied werden Sie allerdings bemerken, wenn Sie den Operator `sizeof()` auf die Arrayvariable anwenden. Als Rückgabe bekommen Sie die Größe des gesamten Arrays. Teilen Sie diesen Wert durch die Größe eines beliebigen Arrayelements, erhalten Sie die Anzahl der Elemente im Array.

```
#include <iostream>
```

```
int main() {
    int feld[10];

    std::cout << "   Array Größe: " << sizeof(feld) << "\n";
    std::cout << " Element Größe: " << sizeof(feld[0]) << "\n";
    std::cout << "Element Anzahl: " << sizeof(feld) / sizeof(feld[0]) << "\n";
}
```

Ausgabe

```
   Array Größe: 40
 Element Größe: 4
Element Anzahl: 10
```

21.2 Mehrere Dimensionen

Auch mehrdimensionale Arrays sind möglich. Hierfür werden einfach Größenangaben für die benötigte Anzahl von Dimensionen gemacht. Das folgende Beispiel legt ein zweidimensionales Array der Größe 4×8 an, das 32 Elemente enthält. Theoretisch ist die Anzahl der Dimensionen unbegrenzt.

```
int feld[4][8];    // Ohne Initialisierung

int feld[4][8] = { // Mit Initialisierung
    { 1, 2, 3, 4, 5, 6, 7, 8 },
    { 9, 10, 11, 12, 13, 14, 15, 16 },
    { 17, 18, 19, 20, 21, 22, 23, 24 },
    { 25, 26, 27, 28, 29, 30, 31, 32 }
};
```

Wie Sie sehen, können auch mehrdimensionale Arrays initialisiert werden. Die äußeren geschweiften Klammern beschreiben die erste Dimension mit 4 Elementen. Die inneren geschweiften Klammern beschreiben dementsprechend die zweite Dimension mit 8 Elementen. Beachten Sie, dass die inneren geschweiften Klammern lediglich der Übersicht dienen, sie sind nicht zwingend erforderlich. Dementsprechend ist es für mehrdimensionale Arrays immer nötig, die Größe aller Dimensionen anzugeben.

Genaugenommen wird eigentlich ein Array von Arrays erzeugt. In unserem Beispiel ist `feld` ein Array mit 4 Elementen vom Typ „Array mit 8 Elementen vom Typ `int`“. Dementsprechend sieht auch der Aufruf mittels Zeigerarithmetik auf einzelne Elemente aus.

```
#include <iostream>

int main(){
    // Anlegen des Arrays mit Initialisierung von eben
```

```
std::cout << feld[2][5]          << "\n"; // Indexoperator
std::cout << *((feld + 2) + 5) << "\n"; // Zeigerarithmetik
}
```

Ausgabe

```
22
22
```

Es sind ebenso viele Dereferenzierungen wie Dimensionen nötig. Um sich vor Augen zu führen, wie die Zeigerarithmetik für diese mehrdimensionalen Arrays funktioniert, ist es nützlich, sich einfach die Adressen bei Berechnungen anzusehen:

```
#include <iostream>

int main(){
    int feld[4][8];

    std::cout << "Größen\n";
    std::cout << "int[4][8]: " << sizeof(feld)    << "\n";
    std::cout << "    int[8]: " << sizeof(*feld)  << "\n";
    std::cout << "        int: " << sizeof(**feld) << "\n";

    std::cout << "Adressen\n";
    std::cout << "        feld: " << feld        << "\n";
    std::cout << "    feld + 1: " << feld + 1    << "\n";
    std::cout << "(*feld) + 1: " << (*feld) + 1 << "\n";
}
```

Ausgabe

```
Größen
int[4][8]: 128
    int[8]: 32
        int: 4
Adressen
        feld: 0x7fff2be5d400
    feld + 1: 0x7fff2be5d420
(*feld) + 1: 0x7fff2be5d404
```

Wie Sie sehen, erhöht `feld + 1` die Adresse um den Wert 32 (Hexadezimal 20), was `sizeof(int[8])` entspricht. Also der Größe aller verbleibenden Dimensionen. Die erste Dereferenzierung liefert wiederum eine Adresse zurück. Wird diese um 1 erhöht, so steigt der Wert lediglich um 4 (`sizeof(int)`).

Hinweis

Beachten Sie, dass auch für mehrdimensionale Arrays keine Indexprüfung erfolgt. Greifen Sie nicht auf ein Element zu, dessen Grenzen außerhalb des Arrays liegen. Beim Array `int [12] [7] [9]` können Sie auf die Elemente `[0..11][0..6][0..8]` zugreifen. Die Zählung beginnt also auch hier immer bei 0 und endet dementsprechend 1 unterhalb der Dimensionsgröße.

21.3 Arrays und Funktionen

Arrays und Funktionen arbeiten in C++ nicht besonders gut zusammen. Sie können keine Arrays als Parameter übergeben und auch keine zurückgeben lassen. Da ein Array allerdings eine Adresse hat (und der Arrayname diese zurückliefert), kann man einfach einen Zeiger übergeben. C++ bietet (ob nun zum besseren oder schlechteren) eine alternative Syntax für Zeiger bei Funktionsparametern an.

```
void funktion(int *parameter);
void funktion(int parameter[]);
void funktion(int parameter[5]);
void funktion(int parameter[76]);
```

Jeder dieser Prototypen ist gleichwertig. Die Größenangaben beim dritten und vierten Beispiel werden vom Compiler ignoriert. Innerhalb der Funktion können Sie wie gewohnt mit dem Indexoperator auf die Elemente zugreifen. Beachten Sie, dass Sie die Arraygröße innerhalb der Funktion *nicht* mit `sizeof(arrayname)` feststellen können. Bei diesem Versuch würden Sie stattdessen die Größe eines Zeigers auf ein Array-Element erhalten.

Aufgrund dieses Verhaltens könnte man die Schreibweise des ersten Prototypen auswählen. Andere Programmierer argumentieren, dass bei der zweiten Schreibweise deutlich wird, dass der Parameter ein Array repräsentiert. Eine Größenangabe bei Arrayparametern ist manchmal anzutreffen, wenn die Funktion nur Arrays dieser Größe bearbeiten kann. Um es noch einmal zu betonen: Diese Größenangaben sind nur ein Hinweis für den Programmierer; der Compiler wird ohne Fehler und Warnung Ihren Array mit allen Elementen übernehmen, auch wenn die beim Funktionsparameter angegebene Größe nicht mit Ihrem Array übereinstimmt.

Bei mehrdimensionalen Arrays sehen die Regeln ein wenig anders aus, da diese Arrays vom Typ *Array* sind. Wie Sie wissen ist es zulässig, Zeiger als Parameter zu übergeben. Entsprechend ist es natürlich auch möglich, einen Zeiger auf einen Array zulässig. Die folgenden Prototypen zeigen, wie die Syntax bei mehrdimensionalen Arrays aussieht.

```
void funktion(int (*parameter)[8]);
void funktion(int parameter[][8]);
void funktion(int parameter[4][8]);
```

Alle diese Prototypen haben einen Parameter vom Typ „Zeiger auf Array mit acht Elementen vom Typ `int`“. Ab der zweiten Dimension geben Sie also tatsächlich Arrays an, somit müssen Sie natürlich auch die Anzahl der Elemente zwingend angeben. Daher können Sie `sizeof()` in

der Funktion verwenden, um die Größe zu ermitteln. Dies ist allerdings nicht notwendig, da Sie bereits im Vorfeld wissen, wie groß der Array ist und vom welchem Typ er ist. Die Größe berechnet sich wie folgt: `sizeof(Typ)*Anzahl` der Elemente. In unserem Beispiel entspricht dies $4 \cdot 8 = 32$. Auch ein zweidimensionales Array können Sie innerhalb der Funktion mit dem normalen Indexoperator zugreifen.

Beachten Sie, dass beim ersten Prototypen die Klammern zwingend notwendig sind, andernfalls hätten die eckigen Klammern auf der rechten Seite des Parameternamen Vorrang. Somit würde der Compiler dies wie oben gezeigt als einen Zeiger behandeln, natürlich unabhängig von der Anzahl der angegebenen Elemente. Ohne diese Klammern würden Sie also einen Zeiger auf einen Zeiger auf `int` deklarieren.

21.4 Lesen komplexer Datentypen

Sie kennen nun Zeiger, Referenzen und Arrays, sowie natürlich die grundlegenden Datentypen. Es kann Ihnen passieren, dass Sie auf Datentypen treffen, die all das in Kombination nutzen. Im Folgenden werden Sie lernen, solche komplexen Datentypen zu lesen und zu verstehen, wie man Sie schreibt.

Tip

Als einfache Regel zum Lesen von solchen komplexeren Datentypen können Sie sich merken:

- Es wird ausgehend vom Namen gelesen.
- Steht etwas rechts vom Namen, wird es ausgewertet.
- Steht rechts nichts mehr, wird der Teil auf der linken Seite ausgewertet.
- Mit Klammern kann die Reihenfolge geändert werden.

Die folgenden Beispiele werden zeigen, dass diese Regeln immer gelten:

```
int i;           // i ist ein int
int *j;         // j ist ein Zeiger auf int
int k[6];       // k ist ein Array von sechs Elementen des Typs int
int *l[6];      // l ist ein Array von sechs Elementen des Typs Zeiger auf int
int (*m)[6];    // m ist ein Zeiger auf ein Array von sechs Elementen des Typs int
int **(&n)[6];   // n ist eine Referenz auf einen Zeiger auf ein Array von
                // sechs Elementen des Typs Zeiger auf int
int *(*o[6])[5]; // o ist ein Array von sechs Elementen des Typs Zeiger auf ein
                // Array von fünf Elementen des Typs Zeiger auf int
int **(*p[6])[5]; // p ist Array von sechs Elementen des Typs Zeiger auf ein Array
                // von fünf Elementen des Typs Zeiger auf Zeiger auf int
```

Nehmen Sie sich die Zeit, die Beispiele nachzuvollziehen. Wenn Sie keine Probleme damit haben, sehen Sie sich das nächste sehr komplexe Beispiel an. Es soll die allgemeine Gültigkeit dieser Regel noch einmal demonstrieren:

```
int ((*(*pFunc[5])(int*, double&))[6])();
```

`pFunc` ist ein Array mit fünf Elementen, das Zeiger auf Zeiger auf Funktionen enthält, die einen Zeiger auf `int` und eine Referenz auf `double` übernehmen und einen Zeiger auf Arrays mit sechs Elementen vom Typ Zeiger auf Funktionen, ohne Parameter, mit einem `int` als Rückgabewert zurückgeben. Einem solchen Monstrum werden Sie beim Programmieren wahrscheinlich selten bis nie begegnen, aber falls doch, können Sie es mit den obengenannten Regeln ganz einfach entschlüsseln. Wenn Sie nicht in der Lage sind, dem Beispiel noch zu folgen, brauchen Sie sich keine Gedanken zu machen: Nur wenige Menschen sind in der Lage, sich ein solches Konstrukt überhaupt noch vorzustellen. Wenn Sie es nachvollziehen können, kommen Sie sehr wahrscheinlich mit jeder Datentypdeklaration klar.

22 Zeichenketten

22.1 Einleitung

In C gibt es keinen eingebauten Datentyp für Zeichenketten, lediglich einen für einzelne Zeichen. Da es in C noch keine Klassen gab, bediente man sich dort der einfachsten Möglichkeit, aus Zeichen Zeichenketten zu bilden: Man legte einfach einen Array von Zeichen an. C++ bietet eine komfortablere Lösung an: Die C++-Standardbibliothek enthält eine Klasse namens `string`. Um diese Klasse nutzen zu können, müssen Sie die gleichnamige Headerdatei `string` einbinden.

```
#include <iostream>
#include <string>

int main() {
    std::string zeichenkette;
    zeichenkette = "Hallo Welt!";
    std::cout << zeichenkette << std::endl;
}
```

Ausgabe

```
Hallo Welt!
```

Wir werden uns in diesem Kapitel mit der C++-Klasse `string` auseinandersetzen. Am Ende des Kapitels beleuchten wir den Umgang mit C-Strings (also `char`-Arrays) etwas genauer. Natürlich liegt auch `string`, wie alle Teile der Standardbibliothek, im Namensraum `std`.

22.2 Wie entsteht ein `string`-Objekt?

Zunächst sind einige Worte zur Notation von Zeichenketten in doppelten Anführungszeichen nötig. Wie Ihnen bereits bekannt ist, werden einzelne Zeichen in einfachen Anführungszeichen geschrieben. Dieser Zeichenliteral ist dann vom Typ `char`. Die doppelten Anführungszeichen erzeugen hingegen eine Instanz eines `char`-Arrays. `"Hallo Welt!"` ist zum Beispiel vom Typ `char[12]`.

Es handelt sich also um eine Kurzschreibweise, zum Erstellen von `char`-Arrays, damit Sie nicht `{'H', 'a', 'l', 'l', 'o', ' ', 'w', 'e', 'l', 't', '!', '\0'}` schreiben müssen, um eine einfache Zeichenkette zu erstellen. Was ist das `'\0'` und warum ist das Array 12 `chars` lang, obwohl es nur 11 Zeichen enthält? Wie bereits erwähnt, ist ein C-String ein Array von Zeichen. Da ein solcher C-String natürlich im Programmablauf Zeichenketten unterschiedlicher Längen enthalten konnte, beendete man die Zeichenkette durch ein Endzeichen: `'\0'`

(Zahlenwert 0). Somit musste ein Array von Zeichen in C immer ein Zeichen länger sein, als die längste Zeichenkette, die im Programmverlauf darin gespeichert wurde.

Diese Kurzschreibweise kann aber noch mehr, als man auf den ersten Blick vermuten würde. Die eben genannte lange Notation zur Initialisierung eines Arrays funktioniert im Quelltext nur, wenn der Compiler auch weiß, von welchem Datentyp die Elemente des Arrays sein sollen. Da Zeichenlitterale jedoch implizit in größere integrale Typen umgewandelt werden können, kann er den Datentyp nicht vom Typ der Elemente, die für die Initialisierung genutzt wurden ableiten:

```
#include <string>

int main() {
    // char-Array mit 12 Elementen
    char a[] = {'H', 'a', 'l', 'l', 'o', ' ', 'W', 'e', 'l', 't', '!', '\0'};

    // int-Array mit 12 Elementen
    int b[] = {'H', 'a', 'l', 'l', 'o', ' ', 'W', 'e', 'l', 't', '!', '\0'};

    // char-Array mit 12 Elementen
    std::string z = {'H', 'a', 'l', 'l', 'o', ' ', 'W', 'e', 'l', 't', '!', '\0'};
}
```

Bei der Notation mit Anführungszeichen ist dagegen immer bekannt, dass es sich um ein char-Array handelt. Entsprechend ist die Initialisierung eines int-Arrays damit nicht möglich. Folgendes dagegen schon:

```
#include <string>

int main() {
    // char-Array mit 12 Elementen
    char a[] = "Hallo Welt!";

    // char-Array mit 12 Elementen
    std::string z = "Hallo Welt!";
}
```

Bei der Erzeugung eines string-Objekts wird eine Funktion aufgerufen, die sich Konstruktor nennt. Was genau ein Konstruktor ist, erfahren Sie im Kapitel über Klassen. In unserem Fall wird also der Konstruktor für das string-Objekt z aufgerufen. Als Parameter erhält er das char-Array "Hallo Welt!". Wie Ihnen bereits bekannt ist, können an Funktionen keine Arrays übergeben werden. Stattdessen wird natürlich ein Zeiger vom Arrayelementtyp (also char) übergeben. Dabei geht aber die Information verloren, wie viele Elemente dieses Array enthält und an dieser Stelle kommt das '\0'-Zeichen (Nullzeichen) ins Spiel. Anhand dieses Zeichens kann auch innerhalb der Zeichenkette erkannt werden, wie lang die übergebene Zeichenkette ist.

Damit wissen Sie nun, wie aus dem einfachen char-Array das fertige string-Objekt wird. Jetzt ist es an der Zeit zu erfahren, was Sie mit diesem Objekt alles machen können

22.3 string und andere Datentypen

Wie Sie bereits im Beispiel von eben gesehen haben, lässt sich die `string`-Klasse problemlos mit anderen Datentypen und Klassen kombinieren. Im ersten Beispiel dieses Kapitels wurde zunächst eine Zuweisung eines `char`-Arrays vorgenommen. Anschließend wurde das `string`-Objekt über `cout` ausgegeben. Auch die Eingabe einer Zeichenkette über `cin` ist mit einem `string`-Objekt problemlos möglich:

```
#include <iostream>
#include <string>

int main() {
    std::string zeichenkette;

    std::cin >> zeichenkette;
    std::cout << zeichenkette;
}
```

Diese Art der Eingabe erlaubt es lediglich, bis zum nächsten Whitespace einzulesen. Es kommt jedoch häufig vor, dass man eine Zeichenkette bis zum Zeilenende oder einem bestimmten Endzeichen einlesen möchte. In diesem Fall ist die Funktion `getline` hilfreich. Sie erwartet als ersten Parameter einen Eingabestream und als zweiten ein `string`-Objekt.

```
#include <iostream>
#include <string>

int main() {
    std::string zeichenkette;

    // Liest bis zum Zeilenende
    std::getline(std::cin, zeichenkette);
    std::cout << zeichenkette;
}
```

Als optionalen dritten Parameter kann man das Zeichen angeben, bis zu dem man einlesen möchte. Im Fall von eben wurde als der Default-Parameter `'\n'` (Newline-Zeichen) benutzt. Im folgenden Beispiel wird stattdessen bis zum ersten kleinen `y` eingelesen.

```
#include <iostream>
#include <string>

int main() {
    std::string zeichenkette;

    // Liest bis zum nächsten y
```

```
std::getline(std::cin, zeichenkette, 'y');
std::cout << zeichenkette;
}
```

22.4 Zuweisen und Verketteten

Genau wie die Basisdatentypen, lassen sich auch `strings` einander zuweisen. Für die Verkettung von `strings` wird der `+`-Operator benutzt und das Anhängen einer Zeichenkette ist mit `+=` möglich.

```
#include <iostream>
#include <string>

int main() {
    std::string string1, string2, string3;

    string1 = "ich bin ";
    string2 = "doof";
    string3 = string1 + string2;
    std::cout << string3 << std::endl;

    string3 += " - " + string1 + "schön";
    std::cout << string3 << std::endl;

    std::cout << string1 + "schön " + string2 << std::endl;
}
```

Ausgabe

```
ich bin doof
ich bin doof - ich bin schön
ich bin schön doof
```

Spielen Sie einfach ein wenig mit den Operatoren, um den Umgang mit ihnen zu lernen.

22.5 Nützliche Methoden

Die `string`-Klasse stellt einige nützliche Methoden bereit. Etwa um den String mit etwas zu füllen, ihn zu leeren oder über verschiedene Eigenschaften Auskunft zu bekommen. Eine Methode wird mit folgender Syntax aufgerufen:

```
«stringname».«methodenname»(«parameter...»);
```

Die Methoden `size()` und `length()` erwarten keine Parameter und geben beide die aktuelle Länge der gespeicherten Zeichenkette zurück. Diese Doppelung in der Funktionalität existiert,

da `string` in Analogie zu den anderen Containerklassen der C++-Standardbibliothek `size()` anbieten muss, der Name `length()` für die Bestimmung der Länge eines Strings aber natürlicher und auch allgemein üblich ist. `empty()` gibt `true` zurück falls der String leer ist, andernfalls `false`.

Mit `clear()` lässt sich der String leeren. Die `resize()`-Methode erwartet ein oder zwei Parameter. Der erste ist die neue Größe des Strings, der zweite das Zeichen, mit dem der String aufgefüllt wird, falls die angegebene Länge größer ist, als die aktuelle. Wird der zweite Parameter nicht angegeben, wird der String mit `'\0'` (Nullzeichen) aufgefüllt. In der Regel werden Sie dieses Verhalten nicht wollen, geben Sie also ein Füllzeichen an, falls Sie sich nicht sicher sind, was Sie tun. Ist die angegebene Länge geringer, als die des aktuellen Strings, wird am Ende abgeschnitten.

Um den Inhalt zweier Strings auszutauschen existiert die `swap()`-Methode. Sie erwartet als Parameter den String mit dem ihr Inhalt getauscht werden soll. Dies ist effizienter, als das Vertauschen über eine dritte, temporäre `string`-Variable.

```
#include <iostream>
#include <string>

int main() {
    std::string zeichenkette1 = "Ich bin ganz lang!";
    std::string zeichenkette2 = "Ich kurz!";

    std::cout << zeichenkette1 << std::endl;
    std::cout << zeichenkette2 << std::endl;

    zeichenkette1.swap(zeichenkette2);

    std::cout << zeichenkette1 << std::endl;
    std::cout << zeichenkette2 << std::endl;
}
```

Ausgabe

```
Ich bin ganz lang!
Ich kurz!
Ich kurz!
Ich bin ganz lang!
```

22.6 Zeichenzugriff

Genau wie bei einem Array können Sie den `[]`-Operator (Zugriffsoperator) verwenden, um auf einzelne Zeichen im String zuzugreifen. Allerdings wird, ebenfalls genau wie beim Array, nicht überprüft, ob der angegebene Wert noch innerhalb der enthaltenen Zeichenkette liegen.

Alternativ existiert die Methode `at()`, die den Index als Parameter erwartet und eine Grenzprüfung ausführt. Im Fehlerfall löst sie eine `out_of_range`-Exception aus. Da Sie den Umgang mit Exceptions wahrscheinlich noch nicht beherrschen, sollten Sie diese Methode vorerst nicht einsetzen

und stattdessen genau darauf achten, dass Sie nicht versehentlich über die Stringlänge hinaus zugreifen.

```
#include <iostream>
#include <string>

int main() {
    std::string zeichenkette = "Ich bin ganz lang!";

    std::cout << zeichenkette[4] << std::endl;
    std::cout << zeichenkette.at(4) << std::endl;

    std::cout << zeichenkette[20] << std::endl;    // Ausgabe von Datenmüll
    std::cout << zeichenkette.at(20) << std::endl; // Laufzeitfehler
}
```

Ausgabe

```
b
b
```

```
terminate called after throwing an instance of 'std::out_of_range'
```

```
what(): basic_string::at
```

```
Abgebrochen
```

Hinweis

Beachten Sie beim Zugriff, dass das erste Zeichen den Index 0 hat. Das letzte Zeichen hat demzufolge den Index `zeichenkette.length() - 1`.

22.7 Manipulation

22.7.1 Suchen

Die Methode `find()` sucht das erste Vorkommen eines Strings und gibt die Startposition (Index) zurück. Der zweite Parameter gibt an, ab welcher Position des Strings gesucht werden soll.

```
#include <iostream>
#include <string>

int main() {
    std::string str = "Zeichenkette";
    std::string find = "k";
    std::cout << str.find(find, 0);
}
```

Ausgabe

Wird ein Substring nicht gefunden, gibt `find()` den Wert `std::string::npos` zurück.

Das Gegenstück zu `find()` ist `rfind()`. Es ermittelt das letzte Vorkommen eines Strings. Die Parameter sind die gleichen wie bei `find()`.

22.7.2 Löschen

Mit der Methode `erase()` können Zeichen im String gelöscht werden. Der erste Parameter gibt den Startwert an. Zusätzlich kann man mit dem zweiten Parameter die Anzahl der Zeichen festlegen. Wird die Methode nur mit dem Startwert aufgerufen, löscht sie alle Zeichen ab dieser Position.

```
#include <iostream>
#include <string>

int main() {
    std::string str = "Hallo Welt.";
    str.erase(5, 1);
    std::cout << str << std::endl;
}
```

Ausgabe

```
HalloWelt.
```

22.7.3 Ersetzen

Sie können `replace()` verwenden, um Strings zu ersetzen. Dafür benötigen Sie die Anfangsposition und die Anzahl der Zeichen, die anschließend ersetzt werden sollen.

```
#include <iostream>
#include <string>

int main() {
    std::string str = "Zeichenkette";
    str.replace(str.find("k"), std::string("kette").length(), "test");
    std::cout << str << std::endl;
}
```

Ausgabe

```
Zeichentest
```

Wie Sie sehen, verwenden wir `find()`, um die Startposition zu ermitteln. Der zweite Parameter gibt die Länge an. Hier soll die alternative Schreibweise verdeutlicht werden; Sie müssen nicht eine zusätzliche Variable deklarieren, sondern können die `std::string`-Klasse wie eine Funktion verwenden und über Rückgabewert auf die Methode `length()` zugreifen. Im dritten Parameter spezifizieren

Sie den String, welcher den ursprünglichen String zwischen der angegebenen Startposition und Startposition + Laenge ersetzt.

22.7.4 Einfügen

Die Methode `insert()` erlaubt es Ihnen, einen String an einer bestimmten Stelle einzufügen.

```
#include <iostream>
#include <string>

int main() {
    std::string str = "Hallo Welt.";
    str.insert(5, " schöne");
    std::cout << str << std::endl;
}
```

Ausgabe

```
Hallo schöne Welt
```

22.7.5 Kopieren

Mit der Methode `substr()` kann man sich einen Substring zurückgeben lassen. Der erste Parameter gibt den Startwert an. Zusätzlich kann man mit dem zweiten Parameter noch die Anzahl der Zeichen festlegen. Wird die Methode nur mit dem Startwert aufgerufen, gibt sie alle Zeichen ab dieser Position zurück.

```
#include <iostream>
#include <string>

int main() {
    std::string str = "Hallo Welt.";
    std::cout << str.substr(0, str.find(' ') - 0) << std::endl;
}
```

Ausgabe

```
Hallo
```

Das `-0` soll verdeutlichen, das der Startwert abgezogen werden muss, an dieser Stelle ist es natürlich überflüssig.

22.8 Vergleiche

C++-Strings können Sie, genau wie Zahlen, miteinander vergleichen. Was Gleichheit und Ungleichheit bei einem String bedeutet, wird Ihnen sofort klar sein. Sind alle Zeichen zweier Strings identisch, so sind beide gleich, andernfalls nicht. Die Operatoren `<`, `>`, `<=` und `>=` geben da schon einige Rätsel mehr auf.

Im Grunde kennen Sie die Antwort bereits. Zeichen sind in C++ eigentlich Zahlen. Sie werden zu Zeichen, indem den Zahlen entsprechende Symbole zugeordnet werden. Der Vergleich erfolgt also einfach mit den Zahlen, welche die Zeichen kodieren. Das erste Zeichen der Strings, das sich unterscheidet, entscheidet darüber, welcher der Strings größer bzw. kleiner ist.

Die meisten Zeichenkodierungen beinhalten in den ersten 7 Bit den ASCII-Code, welchen die nachfolgende Tabelle zeigt.

ASCII-Codetabelle, Nummerierung im Hexadezimalsystem (Teil 1)								
Code¹	...0	...1	...2	...3	...4	...5	...6	...7
0...	<i>NUL</i>	<i>SOH</i>	<i>STX</i>	<i>ETX</i>	<i>EOT</i>	<i>ENQ</i>	<i>ACK</i>	<i>BEL</i>
1...	<i>DLE</i>	<i>DC1</i>	<i>DC2</i>	<i>DC3</i>	<i>DC4</i>	<i>NAK</i>	<i>SYN</i>	<i>ETB</i>
2...	<i>SP</i>	!	"	#	\$	%	&	'
3...	0	1	2	3	4	5	6	7
4...	@	A	B	C	D	E	F	G
5...	P	Q	R	S	T	U	V	W
6...	'	a	b	c	d	e	f	g
7...	p	q	r	s	t	u	v	w

ASCII-Codetabelle, Nummerierung im Hexadezimalsystem (Teil 2)								
Code ²	...8	...9	...A	...B	...C	...D	...E	...F
0...	<i>BS</i>	<i>HT</i>	<i>LF</i>	<i>VT</i>	<i>FF</i>	<i>CR</i>	<i>SO</i>	<i>SI</i>
1...	<i>CAN</i>	<i>EM</i>	<i>SUB</i>	<i>ESC</i>	<i>FS</i>	<i>GS</i>	<i>RS</i>	<i>US</i>
2...	()	*	+	,	-	.	/
3...	8	9	:	;	<	=	>	?
4...	H	I	J	K	L	M	N	O
5...	X	Y	Z	[\]	^	_
6...	h	i	j	k	l	m	n	o
7...	x	y	z					

```
#include <string>
```

```
int main(){
    std::string gross = "Ich bin ganz groß!";
    std::string klein = "Ich bin ganz klein!";

    gross == klein; // ergibt false ('g' != 'k')
    gross != klein; // ergibt true ('g' != 'k')
    gross < klein; // ergibt true ('g' < 'k')
    gross > klein; // ergibt false ('g' < 'k')
    gross <= klein; // ergibt true ('g' < 'k')
    gross >= klein; // ergibt false ('g' < 'k')
}
```

22.9 Zahl zu string und umgekehrt

In C++ gibt es, im Gegensatz zu vielen anderen Programmiersprachen, keine Funktion, um direkt Zahlen in Strings oder umgekehrt umzuwandeln. Es ist allerdings nicht besonders schwierig, eine solche Funktion zu schreiben. Wir haben für die Umwandlung zwei Möglichkeiten:

- Die C-Funktionen `atof()`, `atoi()`, `atol()` und `sprintf()`
- C++-String-Streams

Die C-Variante wird in Kürze im Zusammenhang mit C-Strings besprochen. Für den Moment wollen wir uns der C++-Variante widmen. Stringstreams funktionieren im Grunde genau wie die Ihnen bereits bekannten Ein-/Ausgabestreams `cin` und `cout` mit dem Unterschied, dass sie ein `string`-Objekt als Ziel benutzen.

```
#include <iostream> // Standard-Ein-/Ausgabe
#include <sstream> // String-Ein-/Ausgabe
```

```
int main() {
    std::ostringstream strout; // Unser Ausgabe-Stream
    std::string str;           // Ein String-Objekt
    int var = 10;              // Eine ganzzahlige Variable

    strout << var;              // ganzzahlige Variable auf Ausgabe-Stream ausgeben
    str = strout.str();        // Streaminhalt an String-Variablen zuweisen

    std::cout << str << std::endl; // String ausgeben
}
```

Der vorliegende Code wandelt eine Ganzzahl in einen String um, indem die Ganzzahl auf dem Ausgabe-Stringstream ausgegeben und dann der Inhalt des Streams an den String zugewiesen wird. Die umgekehrte Umwandlung funktioniert ähnlich. Natürlich verwenden wir hierfür einen Eingabe-Stringstream (`istringstream` statt `ostringstream`) und übergeben den Inhalt des Strings an den Stream, bevor wir ihn von diesem auslesen.

```
#include <iostream> // Standard-Ein-/Ausgabe
#include <sstream> // String-Ein-/Ausgabe
```

```
int main() {
    std::istringstream strin; // Unser Eingabe-Stream
    std::string str = "17";   // Ein String-Objekt
    int var;                  // Eine ganzzahlige Variable

    strin.str(str);          // Streaminhalt mit String-Variablen füllen
    strin >> var;             // ganzzahlige Variable von Eingabe-Stream einlesen

    std::cout << var << std::endl; // Zahl ausgeben
}
```

Statt `istringstream` und `ostringstream` können Sie übrigens auch ein `stringstream`-Objekt verwenden, welches sowohl Ein- als auch Ausgabe erlaubt, al-

Bevor der Compiler eine C++-Datei zu sehen kriegt, läuft noch der Präprozessor durch. Er überarbeitet den Quellcode, sodass der Compiler daraus eine Objektdatei erstellen kann. Diese werden dann wiederum vom Linker zu einem Programm gebunden. In diesem Kapitel soll es um den Präprozessor gehen, wenn Sie allgemeine Informationen über Präprozessor, Compiler und Linker brauchen, dann lesen Sie das Kapitel „Compiler¹“.

23.1 #include

Die Präprozessordirektive `#include` haben Sie schon häufig benutzt. Sie fügt den Inhalt der angegebenen Datei ein. Dies ist nötig, da der Compiler immer nur eine Datei übersetzen kann. Viele Funktionen werden aber in verschiedenen Dateien benutzt. Daher definiert man die Prototypen der Funktionen (und einige andere Dinge, die Sie noch kennenlernen werden) in so genannten Headerdateien. Diese Headerdateien werden dann über `#include` eingebunden, wodurch Sie die Funktionen usw. aufrufen können.

Thema wird später näher erläutert...

Ausführlich Informationen über Headerdateien erhalten Sie im gleichnamigen Kapitel^a.

^a Kapitel 24 auf Seite 151

`#include` bietet zwei Möglichkeiten, Headerdateien einzubinden:

```
#include "name" // Sucht im aktuellen Verzeichnis und dann in den Standardpfaden des Compilers
#include <name> // Sucht gleich in den Standardpfaden des Compilers
```

„Aktuelles Verzeichnis“ bezieht sich immer auf das Verzeichnis, in welchem die Datei liegt.

Die erste Syntax funktioniert immer, hat aber den Nachteil, dass dem Compiler nicht mitgeteilt wird, dass es sich um eine Standardheaderdatei handelt. Wenn sich im aktuellen Verzeichnis beispielsweise eine Datei namens `iostream` befände und Sie versuchten über die erste Syntax, die Standardheaderdatei `iostream` einzubinden, bänden Sie stattdessen die Datei im aktuellen Verzeichnis ein, was sehr unwahrscheinlich sein sollte, da Sie hoffentlich immer Dateiendungen wie `.hpp` oder `.h` für Ihre eigenen Header verwenden. Außerdem verlängert das Einbinden von Standardheadern in Anführungszeichen den Präprozessordurchlauf, je nach Anzahl der Dateien im aktuellen Quellpfad.

Aus diesem Grund ist es wichtig zu wissen, ob der eingebundene Header für eine Bibliotheksfunktionalität in den vordefinierten Pfaden des verwendeten Compilers steht, oder ob es eigener Inhalt ist, oder es sich um Zusatzbibliotheken handelt, deren Include-Verzeichnisse an anderen Stellen zu finden sind. Es sollte die Variante gewählt werden, bei der sich die Headerdatei vom Präprozessor am schnellsten finden lässt.

¹ Kapitel 3 auf Seite 9

Tip

Verwenden Sie für Verweise auf Ihre eigenen *Includes* immer eine relative Pfadangabe mit normalen Slashes '/' als Verzeichnisseparator, damit Sie Ihre Quellcodeverzeichnisse auch an anderen Stellen und in anderen Entwicklungsumgebungen schneller kompiliert bekommen. Eine Verzeichnisangabe wie `"/home/ichuser/code/cpp/projekt/zusatzlib/bla.h"` oder `"c:\users\manni\Eigene Dateien\code\cpp\projekt\zusatzlib\bla.h"` ist sehr unangenehm und sorgt für große Verwirrung. Schreiben Sie es nun in `"/./zusatzlib/bla.h"` um, können Sie später Ihr gesamtes Projekt leichter in anderen Pfaden kompilieren und sparen sich selbst und Ihrem Präprozessor einiges an Ärger und Verwirrung.

23.2 #define und #undef

`#define` belegt eine Textsubstitution mit dem angegebenen Wert, z.B.:

```
#define BEGRUESSUNG "Hallo Welt!\n"
cout << BEGRUESSUNG;
```

Makros sind auch möglich, z.B.:

```
#define ADD_DREI(x, y, z) x + y + z + 3
int d(1), e(20), f(4), p(0);
p = ADD_DREI(d, e, f);           // p = d + e + f + 3;
```

Diese sind allerdings mit Vorsicht zu genießen, da durch die strikte Textsubstitution des Präprozessors ohne jegliche Logikprüfungen des Compilers fatale Fehler einfließen können, und sollten eher durch (`inline`)-Funktionen, Konstanten oder sonstige Konzepte realisiert werden. Manchmal lässt es sich allerdings nicht umgehen, ein Makro (weiter) zu verwenden. Beachten Sie bitte immer, dass es sich hierbei um Anweisungen an eine Rohquelltextvorbearbeitungsstufe handelt und nicht um Programmcode.

Da anstelle von einfachen Werten auch komplexere Terme als Parameter für das Makro verwendet werden können und das Makro selbst auch in einen Term eingebettet werden kann, sollten immer Klammern verwendet werden. Bsp.:

```
#define MUL_NOK(x, y) x*y
#define MUL_OK(x, y) ((x)*(y))

resultNok = a*MUL_NOK(b, c + d); // a*b*c + d      = a*b*c + d      <= falsch
resultOk  = a*MUL_OK(b, c + d);  // a*((b)*(c + d)) = a*b*c + a*b*d <= richtig
```

`#undef` löscht die Belegung einer Textsubstitution/Makro, z.B.:

```
#undef BEGRUESSUNG
```

Buchempfehlung

Bitte denken Sie bei allen Ideen, auf die Sie nun kommen, dass Sie fast immer eine Alternative zur Makroprogrammierung haben. Verwenden Sie diese Makros vor allem als Zustandsspeicher für den Präprozessordurchlauf an sich und nicht um Funktionalität Ihres Programms zu erweitern. Einer statischen, konstanten Definition oder Templatefunktionen ist immer der Vorzug zu geben.

23.3

Der #-Ausdruck erlaubt es, den einem Makro übergebenen Parameter als Zeichenkette zu interpretieren:

```
#define STRING(s) #s
cout << STRING(Test) << endl;
```

Das obige Beispiel gibt also den Text "Test" auf die Standard-Ausgabe aus.

23.4

Der ##-Ausdruck erlaubt es, in Makros definierte Strings innerhalb des Präprozessorlaufs miteinander zu kombinieren, z.B.:

```
#define A "Hallo"
#define B "Welt"
#define A_UND_B A##B
```

Als Resultat beinhaltet die Konstante A_UND_B die Zeichenkette "HalloWelt". Der ##-Ausdruck verbindet die Namen der symbolischen Konstanten, nicht deren Werte. Ein weiteres Anwendungsbeispiel:

```
#define MAKE_CLASS( NAME )      \
class NAME                      \
{                                \
    public:                      \
        static void Init##NAME() {};\
};
...
MAKE_CLASS( MyClass )
...
```

```
MyClass::InitMyClass();
```

23.5 #if, #ifdef, #ifndef, #else, #elif und #endif

Direktiven zur *bedingten Übersetzung*, d.h. Programmteile werden entweder übersetzt oder ignoriert.

```
#if 'Ausdruck1'  
// Programmteil 1  
#elif 'Ausdruck2'  
// Programmteil 2  
/*  
    ...  
*/  
#else  
// Programmteil sonst  
#endif
```

Die Ausdrücke hinter #if bzw. #elif werden der Reihe nach bewertet, bis einer von ihnen einen von 0 verschiedenen Wert liefert. Dann wird der zugehörige Programmteil wie üblich verarbeitet, die restlichen werden ignoriert. Ergibt alle Ausdrücke 0, wird der Programmteil nach #else verarbeitet, sofern vorhanden.

Als Bedingungen sind nur *konstante Ausdrücke* erlaubt, d.h. solche, die der Präprozessor tatsächlich auswerten kann. Definierte Makros werden dabei expandiert, verbleibende Namen durch 0L ersetzt. Insbesondere dürfen keine Zuweisungen, Funktionsaufrufe, Inkrement- und Dekrementoperatoren vorkommen. Das spezielle Konstrukt

```
#defined 'Name'
```

wird durch 1L bzw. 0L ersetzt, je nachdem, ob das Makro *Name* definiert ist oder nicht.

#ifdef ist eine Abkürzung für #if defined.

#ifndef ist eine Abkürzung für #if ! defined.

Beispiel: Nehmen wir an, dass Sie Daten in zusammengesetzten Strukturen immer an 4-Byte Grenzen ausrichten wollen. Verschiedene Compiler bieten hierfür verschiedene compilerspezifische Pragma-Direktiven. Da diese Konfiguration nicht im C++-Standard definiert ist, kann nicht sichergestellt werden, dass es eine allgemeingültige Anweisung hierfür gibt. Sie definieren daher mit #define in einer Headerdatei, die überall eingebunden ist, welchen Compiler Sie verwenden. Z.B. mit #define FLAG_XY_COMPILER_SUITE in einer Datei namens "compiler-config-all.hpp". Das gibt Ihnen die Möglichkeit, an Stellen, an denen Sie compilerspezifisches Verhalten verwenden wollen, dieses auch auszuwählen.

```
#include "compiler-config-all.hpp"
```

```
#ifdef WIN32
#pragma pack(4)
#elif USING_GCC
#pragma align=4
#elif FLAG_XY_COMPILER_SUITE
#pragma ausrichtung(byte4)
#endif
```

23.6 #error und #warning

`#error` gibt eine Fehlermeldung während des Compilerlaufs aus und bricht den Übersetzungsvorgang ab, z.B.:

```
#error Dieser Quellcode-Abschnitt sollte nicht mit compiliert werden!
```

`#warning` ist ähnlich wie `#error`, mit dem Unterschied, dass das Kompilieren nicht abgebrochen wird. Es ist allerdings nicht Teil von ISO-C++, auch wenn die meisten Compiler es unterstützen. Meistens wird es zum Debuggen eingesetzt:

```
#warning Dieser Quellcode-Abschnitt muss noch überarbeitet werden.
```

23.7 #line

Setzt den Compiler-internen Zeilenzähler auf den angegebenen Wert, z.B.:

```
#line 100
```

23.8 #pragma

Das `#pragma`-Kommando ist vorgesehen, um eine Reihe von Compiler-spezifischen Anweisungen zu implementieren, z.B.:

```
#pragma comment(lib, "advapi32.lib")
```

Kennt ein bestimmter Compiler eine Pragma-Anweisung nicht, so gibt er üblicherweise eine Warnung aus, ignoriert diese nicht für ihn vorgesehene Anweisung aber ansonsten.

Um z.B. bei MS VisualC++ eine "störende" Warnung zu unterdrücken, gibt man folgendes an:

```
#pragma warning( disable: 4010 ) // 4010 ist Nummer der Warnung
```

23.9 Vordefinierte Präprozessor-Variablen

- `__LINE__`: Zeilennummer
- `__FILE__`: Dateiname
- `__DATE__`: Datum des Präprozessoraufrufs im Format Monat/Tag/Jahr
- `__TIME__`: Zeit des Präprozessoraufrufs im Format Stunden:Minuten:Sekunden
- `__cplusplus`: Ist nur definiert, wenn ein C++-Programm verarbeitet wird

24 Headerdateien

Sie haben bereits mit 2 Headerdateien Bekanntschaft gemacht: `iostream` und `string`. Beide sind so genannte Standardheader, das heißt, sie sind in der Standardbibliothek jedes Compilers enthalten.

24.1 Was ist eine Headerdatei?

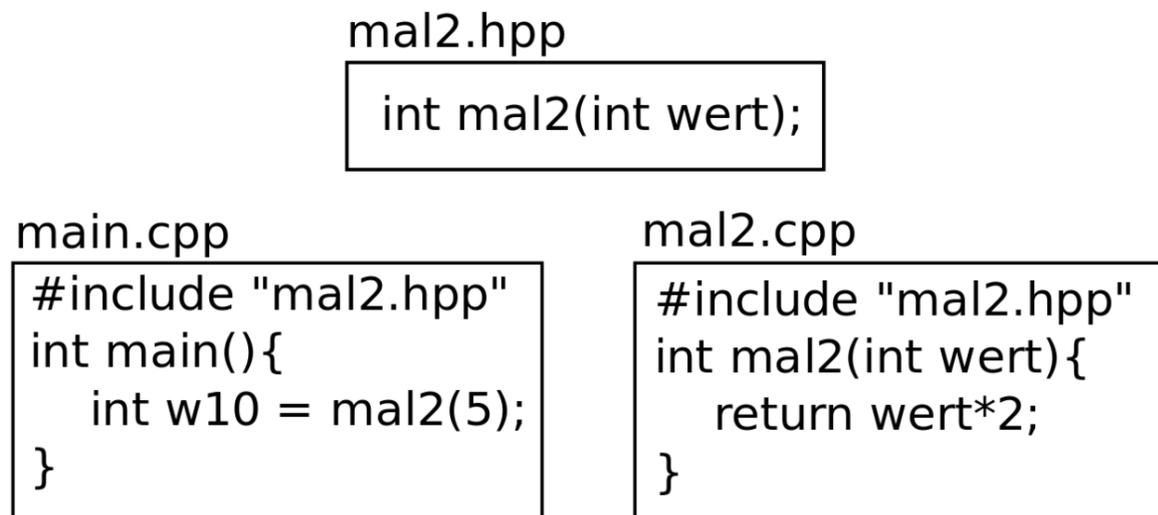


Abb. 2

Headerdateien sind gewöhnliche C++-Dateien, die im Normalfall Funktionsdeklarationen und ähnliches enthalten. Sicher werden Sie sich erinnern: Deklarationen machen dem Compiler bekannt, wie etwas benutzt wird. Wenn Sie also eine Headerdatei einbinden und darin eine Funktionsdeklaration steht, dann weiß der Compiler wie die Funktion aufgerufen wird. Der Compiler weiß zu diesem Zeitpunkt nicht, was die Funktion tut, aber das ist auch nicht nötig um sie aufrufen zu können.

Das Einbinden einer Headerdatei erfolgt über die Präprozessordirektive `#include`. Der Code in der Datei die mit `include` referenziert wird, wird vom Präprozessor einfach an der Stelle eingefügt, an der das `include` stand.

In der nebenstehenden Darstellung wird die Headerdatei `mal2.hpp` von den Quelldateien `main.cpp` und `mal2.cpp` eingebunden. Um dieses Beispiel zu übersetzen, müssen die Dateien alle im gleichen Verzeichnis liegen. Sie übergeben die Quelldateien an einen Compiler und rufen anschließend den Linker auf, um die entstanden Objektdateien zu einem Programm zu binden.

Hinweis

Falls Sie mit der GCC arbeiten, beachten Sie bitte, dass g++ sowohl Compiler als auch Linker ist. Wenn Sie nur die beiden Quelldateien ohne weitere Parameter angeben, wird nach dem Kompilieren automatisch gelinkt. Der zweite Aufruf entfällt somit.

Einige Informationen zu Compilern und Linkern finden Sie übrigens im Kapitel *Compiler^a*.

^a Kapitel 3 auf Seite 9

24.2 Namenskonventionen

Übliche Dateiendungen für C++-Quelltexte sind „.cpp“ oder „.cc“. Headerdateien haben oft die Endungen „.hpp“, „.hh“ und „.h“. Letztere ist allerdings auch die gebräuchlichste Dateiendung für C-Header, weshalb zugunsten besserer Differenzierung empfohlen wird, diese nicht zu benutzen.

Die Standardheader von C++ haben überhaupt keine Dateiendung, wie Sie vielleicht schon anhand der beiden Headerdateien `iostream` und `string` erraten haben. In der Anfangszeit von C++ endeten Sie noch auf „.h“, inzwischen ist diese Notation aber nicht mehr gültig, obwohl sie immer noch von vielen Compilern unterstützt wird. Der Unterschied zwischen „.iostream“ und „.iostream.h“ besteht darin, dass in ersterer Headerdatei alle Deklarationen im Standardnamespace `std` vorgenommen werden.

Prinzipiell kommt es nicht darauf an, welche Dateiendungen Sie ihren Dateien geben, dennoch ist es sinnvoll, eine übliche Endung zu verwenden, wenn auch nur, um einer möglichen Verwechslungsgefahr vorzubeugen. Wenn Sie sich einmal für eine Dateiendung entschieden haben, ist es vorteilhaft, darin eine gewisse Konstanz zu bewahren, nicht nur vielleicht aus Gemütlichkeit und Zeiteinsparnis, sondern auch im Sinne des schnellen Wiederfindens. Einige Beispiele für Headerdateiendungen finden Sie, wenn Sie sich einfach mal einige weitverbreitete C++-Bibliotheken ansehen. Boost verwendet „.hpp“, wxWidgets nutzt „.h“ und Qt orientiert sich an der Standardbibliothek, hat also gar keine Dateiendung.

24.3 Schutz vor Mehrfacheinbindung

Da Headerdateien oft andere Headerdateien einbinden, kann es leicht passieren, dass eine Headerdatei mehrfach eingebunden wird. Da viele Header nicht nur Deklarationen, sondern auch Definitionen enthalten, führt dies zu Compiler-Fehlermeldungen, da innerhalb einer Übersetzungseinheit ein Name stets nur genau einmal definiert werden darf (mehrfache Deklarationen, die keine Definitionen sind, sind jedoch erlaubt). Um dies zu vermeiden, wird der Präprozessor verwendet. Am Anfang der Headerdatei wird ein Präprozessor-`#ifndef` ausgeführt, das prüft, ob ein Symbol nicht definiert wurde, ist dies der Fall, wird das Symbol definiert. Am Ende der Headerdatei wird die Abfrage mit einem Präprozessor-`#endif` wieder beendet.

```
#ifndef _mal2_hpp_
#define _mal2_hpp_

int mal2(int wert);
```

```
#endif
```

Als Symbol wird üblicherweise aus dem Dateinamen des Headers abgeleitet. In diesem Fall wurde der Punkt durch einen Unterstrich ersetzt und ein weiterer Unterstrich vor und nach dem Dateinamen eingefügt. Wenn der Präprozessor nun die Anweisung bekommt, diese Datei in eine andere einzubinden, wird er das anstandslos tun. Findet er eine zweite Anweisung, die Datei einzubinden, wird er alles von `#ifndef` bis `#endif` überspringen, da das Symbol `_ma12_hpp_` ja nun bereits definiert wurde.

Sie können sich eine beliebige Methode zum Generieren von Symbolen ausdenken. Da es schnell passieren kann, dass Sie eine Bibliothek verwenden, in der Headerdateien enthalten sind, welche Sie auch in ihrem eigenen Projekt bereits verwenden, ist es jedoch empfehlenswert an dieser Stelle nicht mit Zeichen zu geizen. Eine mögliche Variante ist beispielsweise neben dem Dateinamen auch noch den Projektnamen zu verwenden und eventuell auch noch eine feste Zeichenfolge, um das Symbol unmissverständlich als Mehrfacheinbindungsschutz zu kennzeichnen. Das könnte dann etwa so aussehen:

```
_Projektname_Dateiname_Dateiendung_INCLUDED_
```

Tip

Mehr Informationen über Präprozessor-Anweisungen finden Sie im Kapitel Vorarbeiter des Compilers^a.

^a Kapitel 22.8 auf Seite 145

24.4 Inline-Funktionen

Im Kapitel „Prozeduren und Funktionen“¹ haben Sie bereits erfahren was eine Inline-Funktion ist. Das Schlüsselwort `inline` empfiehlt dem Compiler, beim Aufruf einer Funktion den Funktionsrumpf direkt durch den Funktionsaufruf zu ersetzen, wodurch bei kurzen Funktionen die Ausführungsgeschwindigkeit gesteigert werden kann. Es bewirkt aber noch mehr. Normalerweise darf eine Definition immer nur einmal gemacht werden. Da für das Inlinen einer Funktion aber die Definition bekannt sein muss, gibt es für Inline-Funktionen eine Ausnahme: Sie dürfen beliebig oft definiert werden, solange alle Definitionen identisch sind. Deshalb dürfen (und sollten) Inline-Funktionen in den Headerdateien definiert werden, ohne dass sich der Linker später über eine mehrfache Definition in verschiedenen Objektdateien beschweren wird.

Hinweis

Inline-Funktionen dürfen auch in `cpp`-Dateien definiert werden. Allerdings können sie dann auch nur innerhalb der Objektdatei, die aus der `cpp`-Datei erzeugt wird, geinlined werden und das ist in aller Regel nicht beabsichtigt.

¹ Kapitel 16 auf Seite 87

25 Das Klassenkonzept

In einem späteren Abschnitt werden Sie etwas über objektorientierte Programmierung erfahren. Die notwendige Basis dafür werden wir in diesem Abschnitt erarbeiten. Sie haben inzwischen einiges über die verschiedenen Basisdatentypen gelernt und auch erfahren, wie Sie diese Daten mithilfe von Funktionen manipulieren können. Im Prinzip könnten Sie mit diesen Mitteln bereits jedes beliebige Programm schreiben. Allerdings werden Programme, die mit diesen Mitteln geschrieben wurden oft unübersichtlich, sobald eine gewisse Größe überschritten wird. Aus diesem Grund wurde ein neues Konzept erdacht, das für mehr Übersicht sorgt. Anstatt einfach Daten (Variablen) zu haben, die von Funktionen manipuliert werden, fasst man die Daten und die darauf arbeitenden Funktionen zu einem neuen Konstrukt zusammen. Dieses Konstrukt nennt sich „Klasse“ und das Zusammenfassen wird als Kapselung bezeichnet.

Eine Klasse hat üblicherweise einen Namen und besteht aus Variablen und Funktionen, welche als Klassenmember (zu deutsch etwa Mitglieder) bezeichnet werden. Ein Funktionsmember bezeichnet man auch als Methode der Klasse. Der wesentliche Vorteil dieser Technik besteht darin, dass ein Nutzer der Klasse nicht wissen muss, wie die Klasse intern arbeitet. Andererseits können Sie die konkrete Implementierung der Klasse jederzeit ändern, ohne dass der Code, in dem die Klasse verwendet wird, geändert werden muss. Lediglich bei Schnittstellenänderungen muss auch Code außerhalb der Klasse angepasst werden. Als Schnittstelle bezeichnet man die Member der Klasse, die von außen sichtbar sind. Dies werden meistens eine Reihe von Methoden sein, während Variablen nur sehr selten direkt sichtbar sein sollten.

Eine Klasse ist ein benutzerdefinierter Datentyp. Somit können Sie also, genau wie bei einem Basisdatentyp, Variablen vom Typ der Klasse (= deren Name) erstellen. Eine Variable vom Typ einer Klasse beinhaltet alle in der Klasse deklarierten Variablen. Somit verbraucht eine Klassenvariable so viel Speicherplatz, wie die Variablen, die in ihr deklariert wurden. Natürlich hat der Übersetzer hier einige Freiräume, sodass Sie nicht einfach davon ausgehen können, dass eine Klassenvariable genau so viel Speicherplatz belegt, wie die Summe ihrer Variablenmember. Wenn der Übersetzer es aus Optimierungsgründen für sinnvoll hält, etwas mehr Speicherplatz zu nutzen, um etwa die internen Variablen so im Speicher auszurichten, dass auf sie schnell zugegriffen werden kann, darf er dies tun. Wenn Sie die genaue Größe wissen möchten, die eine Variable im Speicher belegt, können Sie den `sizeof`-Operator auf die Klasse oder eine Variable vom Typ der Klasse anwenden.

Um etwas besser zwischen einer Variable innerhalb einer Klasse und einer Variable vom Typ der Klasse unterscheiden zu können, werden Variablen vom Typ der Klasse oft als Objekte der Klasse bezeichnet. Der Begriff Objekt ist in C++ etwas weiter gefasst, als der Begriff Variable. Somit sind auch alle Variablen vom Typ eines Basisdatentyps Objekte, aber nicht alle umgekehrt. Beispielsweise kann eine Funktion ein Objekt zurückgeben. Da dieses jedoch keinen Namen hat, handelt es sich dabei nicht um eine Variable.

Klassen haben zwei spezielle Methoden, die beim Erstellen bzw. Zerstören eines Objektes vom Typ der Klasse aufgerufen werden. Erstere bezeichnet man als Konstruktor und die zweite als Destruktor der Klasse. Es ist möglich, verschiedene Operatoren für eine Klasse zu überladen, sodass diese

dann auf Klassenobjekte angewendet werden können. Innerhalb einer Klasse kann es verschiedene Sichtbarkeitsbereiche geben, die darüber entscheiden, ob ein Member nur innerhalb von Methoden der Klasse oder auch von außerhalb aufgerufen werden kann. Diese Sichtbarkeitsbereiche entscheiden entsprechend darüber, welche Member zur Schnittstelle gehören.

Das waren jetzt wieder sehr viele Informationen auf wenig Raum, aber machen Sie sich keine Sorgen, wenn Sie sich nicht alles merken konnten: In den nächsten Kapiteln wird auf die verschiedenen Eigenschaften von Klassen noch näher eingegangen, sodass Sie die Gedanken dahinter nachvollziehen können.

25.1 Ein eigener Datentyp

Nun wird es aber Zeit, dass wir auch mal eine eigene Klasse schreiben. Das folgende Beispiel soll die prinzipielle Arbeitsweise einer Klasse demonstrieren.

```
#include <iostream>

class Auto{
public:
    Auto(int tankgroesse, float tankinhalt, float verbrauch);

    void info()const;

    void fahren(int km);
    void tanken(float liter);

private:
    int    tankgroesse_;
    float tankinhalt_;
    float verbrauch_;
};

Auto::Auto(int tankgroesse, float tankinhalt, float verbrauch):
    tankgroesse_(tankgroesse),
    tankinhalt_(tankinhalt),
    verbrauch_(verbrauch)
    {}

void Auto::info()const{
    std::cout << "In den Tank passen " << tankgroesse_ << " Liter Treibstoff.\n";
    std::cout << "Aktuell sind noch " << tankinhalt_ << " Liter im Tank.\n";
    std::cout << "Der Wagen verbraucht " << verbrauch_ << " Liter pro 100 km.\n";
    std::cout << std::endl;
}

void Auto::fahren(int km){
```

```

std::cout << "Fahre " << km << " km.\n";
tankinhalt_ -= verbrauch_*km/100;

if(tankinhalt_ < 0.0f){
    tankinhalt_ = 0.0f;

    std::cout << "Mit dem aktuellen Tankinhalt schaffen Sie die Fahrt leider nicht.\n";
    std::cout << "Der Wagen ist unterwegs liegengeblieben, Zeit zu tanken!\n";
}

std::cout << std::endl;
}

void Auto::tanken( float liter ) {
    std::cout << "Tanke " << liter << " Liter.\n";
    tankinhalt_ += liter;

    if( tankinhalt_ > tankgroesse_ ) {
        tankinhalt_ = tankgroesse_;

        std::cout << "Nicht so übereifrig! Ihr Tank ist jetzt wieder voll.\n";
        std::cout << "Sie haben aber einiges daneben gegossen!\n";
    }

    std::cout << std::endl;
}
}

```

Diese Klasse nutzt vieles, was Sie im Laufe dieses Abschnittes noch kennenlernen werden. Für den Moment sollten Sie wissen, dass diese Klasse drei verschiedene Daten beinhaltet. Diese Daten sind die drei Variablen, deren Namen auf einen Unterstrich () enden. Vier Funktionen arbeiten auf diesen Daten.

Sie haben nun gesehen, wie die Klasse aufgebaut ist, und in den folgenden Kapiteln wird dieser Aufbau genauer erläutert. Jetzt sollen Sie jedoch erst einmal den Vorteil einer Klasse verstehen, denn um eine Klasse zu benutzen, müssen Sie keine Ahnung haben, wie diese Klasse intern funktioniert.

```

int main(){
    Auto wagen(80, 60.0f, 5.7f);

    wagen.info();

    wagen.tanken(12.4f);
    wagen.info();
}

```

```
wagen.fahren(230);  
wagen.info();  
  
wagen.fahren(12200);  
wagen.info();  
  
wagen.tanken(99.0f);  
wagen.info();  
}
```

Ausgabe

In den Tank passen 80 Liter Treibstoff.
Aktuell sind noch 60 Liter im Tank.
Der Wagen verbraucht 5.7 Liter pro 100 km.

Tanke 12.4 Liter.

In den Tank passen 80 Liter Treibstoff.
Aktuell sind noch 72.4 Liter im Tank.
Der Wagen verbraucht 5.7 Liter pro 100 km.

Fahre 230 km.

In den Tank passen 80 Liter Treibstoff.
Aktuell sind noch 59.29 Liter im Tank.
Der Wagen verbraucht 5.7 Liter pro 100 km.

Fahre 12200 km.
Mit dem aktuellen Tankinhalt schaffen Sie die Fahrt leider nicht.
Der Wagen ist unterwegs liegengeblieben, Zeit zu tanken!

In den Tank passen 80 Liter Treibstoff.
Aktuell sind noch 0 Liter im Tank.
Der Wagen verbraucht 5.7 Liter pro 100 km.

Tanke 99 Liter.
Nicht so übereifrig! Ihr Tank ist jetzt wieder voll.
Sie haben aber einiges daneben gegossen!

In den Tank passen 80 Liter Treibstoff.
Aktuell sind noch 80 Liter im Tank.
Der Wagen verbraucht 5.7 Liter pro 100 km.

In der ersten Zeile von `main()` wird ein `Auto`-Objekt mit dem Namen `wagen` erstellt. Anschließend werden Methoden dieses Objekts aufgerufen, um die Daten darin zu verwalten. Von den Daten innerhalb des Objekts kriegen Sie beim Arbeiten mit dem Objekt überhaupt nichts mit. Lediglich die Ausgabe verrät, dass die drei Methoden untereinander über diese Daten „kommunizieren“.

Die vierte Methode (jene, die mit dem Klassennamen identisch ist) wird übrigens auch aufgerufen. Gleich in der ersten Zeile von `main()` wird diese Methode genutzt, um das Objekt zu erstellen. Es handelt sich also um den Konstruktor der Klasse.

26 Erstellen und Zerstören

In C++-Klassen gibt es zwei besondere Arten von Methoden: Konstruktoren und den Destruktor. Ein Konstruktor wird beim Anlegen eines Objektes ausgeführt, der Destruktor vor der „Zerstörung“ desselben. Der Name des Konstruktors ist immer gleich dem Klassennamen, der Destruktor entspricht ebenfalls dem Klassennamen, jedoch mit einer führendem Tilde (~).

Konstruktoren und Destrukturen haben keinen Rückgabety, auch nicht `void`. Der Konstruktor kann nicht als Methode aufgerufen werden, beim Destruktor ist dies hingegen möglich (aber nur selten nötig, dazu später mehr).

26.1 Konstruktor

Jede Klasse hat einen oder mehrere Konstruktoren. Ein solcher Konstruktor dient zur Initialisierung eines Objektes. Im Folgenden wird eine Klasse `Bruch` angelegt, die über den Konstruktor die Membervariablen `zaehler_` und `nenner_` initialisiert.

```
class Bruch{
public:
    Bruch(int z, int n):
        zaehler_(z), // Initialisierung von zaehler_
        nenner_(n)   // Initialisierung von nenner_
    {}

private:
    int zaehler_;
    int nenner_;
};

int main(){
    Bruch objekt(7, 10); // Initialisierung von objekt
}
```

Wie Sie sehen, ist der Methodenrumpf von `Bruch::Bruch` leer. Die Initialisierung findet in den beiden Zeilen über dem Rumpf statt. Nach dem Prototyp wird ein Doppelpunkt geschrieben, darauf folgt eine Liste der Werte, die initialisiert werden sollen. Vielleicht erinnern Sie sich noch, dass es zur Initialisierung zwei syntaktische Varianten gibt:

```
int wert(8);
```

```
int wert = 8;
```

Innerhalb der Initialisierungsliste ist nur die erste Variante zulässig. Auch in der Hauptfunktion `main()` findet eine Initialisierung statt. Hier könnte die zweite Variante verwendet werden, jedoch besitzt der Konstruktor zwei Parameter, daher ist die Variante mit dem Gleichheitszeichen nicht sofort offensichtlich. Dieses Thema wird am Ende des Kapitels genauer besprochen, da zum Verständnis dessen ein wenig Hintergrundwissen nötig ist, das im Verlauf des Kapitels vermittelt wird.

Hinweis

Beachten Sie, dass die Initialisierung der Variablen einer Klasse in der Reihenfolge erfolgt, in der sie in der Klasse deklariert wurden. Die Initialisierungsreihenfolge ist *unabhängig* von der Reihenfolge, in der sie in der Initialisierungsliste angegeben wurden. Viele Compiler warnen, wenn man so etwas macht, da derartige Code möglicherweise nicht das tut, was man erwartet.

Natürlich ist es wie bei Funktionen möglich und in der Regel zu empfehlen, die Methodendeklaration von der Methodendefinition zu trennen. Die Definition sieht genau so aus, wie bei einer normalen Funktion. Der einzige auffällige Unterschied besteht darin, dass dem Methodennamen der Klassenname vorangestellt wird, getrennt durch den Bereichsoperator (`:`).

```
class Bruch{
public:
    Bruch(int z, int n);    // Deklaration

private:
    int zaehler_;
    int nenner_;
};

Bruch::Bruch(int z, int n): // Definition
    zaehler_(z),          // Initialisierung von zaehler_
    nenner_(n)            // Initialisierung von nenner_
{}

int main(){
    Bruch objekt(7, 10);   // Der Bruch 7/10 oder auch 0.7
}
```

Wie bereits erwähnt, hat der Konstruktor keinen Rückgabotyp, daher wird auch in der Definition keiner angegeben. Bei den Basisdatentypen ist es bezüglich der Performance übrigens egal, ob Sie diese initialisieren oder zuweisen. Sie könnten also den gleichen Effekt erzielen, wenn Sie statt der Initialisierungsliste eine Zuweisung im Funktionsrumpf benutzen. Allerdings gilt dies wirklich nur für die Basisdatentypen, bei komplexen Datentypen ist die Initialisierung oft deutlich schneller. Außerdem können konstante Variablen ausschließlich über die Initialisierungsliste einen Wert erhalten. Nun aber noch mal ein einfaches Beispiel, in dem der Funktionsrumpf des Konstruktors zur Anfangswertzuweisung dient:

```

class Bruch{
public:
    Bruch(int z, int n = 1); // Deklaration

private:
    int zaehler_;
    int nenner_;
};

Bruch::Bruch(int z, int n){ // Definition
    zaehler_ = z; // Zuweisung
    nenner_ = n; // Zuweisung
}

```

Wie Sie sehen entfällt der Doppelpunkt, wenn Sie die Initialisierungsliste nicht nutzen. Natürlich können Sie auch Initialisierungsliste und Funktionsrumpf parallel benutzen.

Irgendwann werden Sie sicher einmal in die Verlegenheit kommen ein Array als Membervariable innerhalb Ihrer Klasse zu deklarieren. Leider gibt es keine Möglichkeit Arrays zu initialisieren, sie müssen immer im Konstruktorrumpf mittels Zuweisung ihren Anfangswert erhalten. Entsprechend ist es auch nicht möglich ein Memberarray mit konstanten Daten zu erstellen. Wenn Sie also zwingend eine Initialisierung benötigen, müssen Sie wohl oder übel Einzelvariablen erstellen oder auf eine arrayähnliche Klasse zurückgreifen. `std::vector` aus der Standardheaderdatei `vector` ist meist eine gute Alternative. Um Performanceeinbußen zu verhindern, müssten Sie das `std::vector`-Objekt zunächst mittels der Methode `reserve(n)` anweisen für `n` Elemente Speicher zu allozieren und anschließend Ihre Elemente mittels der Methode `push_back(element)` hinzufügen. Beides passiert natürlich im Funktionsrumpf Ihres Konstruktors, in der Initialisierungsliste müssten Sie keine Angaben machen.

26.1.1 Defaultparameter

Sie können einem Konstruktor ebenso Defaultparameter vorgeben, wie einer gewöhnlichen Funktion. Die syntaktischen Regeln sind ebenfalls identisch.

```

class Bruch{
public:
    Bruch(int z, int n = 1); // Deklaration mit Defaultwert

private:
    int zaehler_;
    int nenner_;
};

Bruch::Bruch(int z, int n): // Definition
    zaehler_(z),

```

```
nenner_(n)
{}
```

26.1.2 Mehrere Konstruktoren

Auch das Überladen des Konstruktors funktioniert wie das Überladen einer Funktion. Deklarieren Sie mehrere Prototypen innerhalb der Klasse und schreiben für jeden eine Definition:

```
class Bruch{
public:
    Bruch(int z);          // Deklaration
    Bruch(int z, int n); // Deklaration

private:
    int zaehler_;
    int nenner_;
};

Bruch::Bruch(int z):      // Definition
    zaehler_(z),
    nenner_(1)
{}

Bruch::Bruch(int z, int n): // Definition
    zaehler_(z),
    nenner_(n)
{}
```

Wenn mehrere Konstruktoren das gleiche tun, ist es oft sinnvoll, diese gleichen Teile in eine eigene Methode (üblicherweise mit dem Namen `init()`) zu schreiben. Das ist kürzer, übersichtlicher, meist schneller und hilft auch noch bei der Vermeidung von Fehlern. Denn wenn Sie den Code nachträglich ändern, dann müssten Sie diese Änderungen sonst für jeden Konstruktor vornehmen. Nutzen Sie eine `init()`-Methode, müssen Sie den Code nur innerhalb von dieser einmal ändern.

```
class A{
public:
    A(double x);
    A(int x);

private:
    void init();

    int x_;
    int y_;
```

```

};

A::A(double x):
    x_(x) {
    init();
}

A::A(int x):
    x_(x) {
    init();
}

void A::init(){
    y_=7000;
}

```

Dieses Beispiel ist zugegebenermaßen ziemlich sinnfrei, aber das Prinzip der `init()`-Funktion wird deutlich. Wenn Sie sich nun irgendwann entscheiden, `y_` als Anfangswert 3333 zuzuweisen, dann müssen Sie dies nur in der `init()`-Funktion ändern und nicht in jedem einzelnen Konstruktor.

26.1.3 Standardkonstruktor

Als Standardkonstruktor bezeichnet man einen Konstruktor, der keine Parameter erwartet. Ein Standardkonstruktor für unsere `Bruch`-Klasse könnte zum Beispiel folgendermaßen aussehen:

```

class Bruch{
public:
    Bruch();           // Deklaration Standardkonstruktor

private:
    int zaehler_;
    int nenner_;
};

Bruch::Bruch():      // Definition Standardkonstruktor
    zaehler_(0),
    nenner_(1)
{}

```

Natürlich könnten wir in diesem Beispiel den Konstruktor überladen, um neben dem Standardkonstruktor auch die Möglichkeiten zur Initialisierung mit beliebigen Werten zu haben. Es bietet es sich hier jedoch an, dafür Defaultparameter zu benutzen. Das folgende kleine Beispiel erlaubt die Initialisierung mit einer ganzen Zahl und mit einer gebrochenen Zahl (also zwei Ganzzahlen: Zähler und Nenner). Außerdem wird auch gleich noch der Standardkonstruktor bereitgestellt, welcher den `Bruch` mit 0 initialisiert, also den Zähler auf Null und den Nenner auf Eins setzt.

```
class Bruch{
public:
    // Erlaubt 3 verschiedene Aufrufmöglichkeiten, darunter die des Standardkonstruktors
    Bruch(int z = 0, int n = 1);

private:
    int zaehler_;
    int nenner_;
};

Bruch::Bruch(int z, int n): // Definition des Konstruktors
    zaehler_(z),
    nenner_(n)
    {}
```

Im Kapitel „Leere Klassen?“¹ werden Sie noch einiges mehr über den Standardkonstruktor erfahren.

26.1.4 Kopierkonstruktor

Neben dem Standardkonstruktor hat der Kopierkonstruktor noch eine besondere Bedeutung. Er erstellt, wie der Name bereits andeutet, ein Objekt einer Klasse, anhand eines bereits vorhandenen Objektes. Der Parameter des Kopierkonstruktors ist also immer eine Referenz auf ein konstantes Objekt derselben Klasse. Der Kopierkonstruktor unserer `Bruch`-Klasse hat folgende Deklaration.

```
Bruch::Bruch(Bruch const& bruch_objekt);
```

Beachten Sie, dass ein Kopierkonstruktor immer eine Referenz auf ein Objekt (meist ein konstantes Objekt) übernimmt. Würde an dieser Stelle eine „call-by-value“-Übergabe stehen, müsste für die Objektübergabe an den Kopierkonstruktor ja selbst das Objekt zunächst (mittels Kopierkonstruktor) kopiert werden. Dies führt zu einer Endlosrekursion beim Kompilieren und ist somit verboten.

Wenn wir keinen eigenen Kopierkonstruktor schreiben, erstellt der Compiler einen für uns. Dieser implizite Kopierkonstruktor initialisiert alle Membervariablen mit den entsprechenden Werten der Membervariablen im übergebenen Objekt. Für den Moment ist diese vom Compiler erzeugte Variante ausreichend. Später werden Sie Gründe kennenlernen, die es notwendig machen, einen eigenen Kopierkonstruktor zu schreiben. Wenn Sie die Nutzung des Kopierkonstruktors verbieten wollen, dann schreiben Sie seine Deklaration in den `private`-Bereich der Klassendeklaration. Das bewirkt einen Kompilierfehler, wenn jemand versucht den Kopierkonstruktor aufzurufen.

Neben dem Kopierkonstruktor erzeugt der Compiler übrigens auch noch eine Kopierzuweisung, auch zu dieser werden Sie später noch mehr erfahren. Auch diese können Sie verbieten, indem Sie sie im `private` deklarieren. Wie dies geht, erfahren Sie im Kapitel zu Operatorüberladung².

1 <http://de.wikibooks.org/wiki/C%2B%2B-Programmierung%2F%20Eigene%20Datentypen%20definieren%2F%20Leere%20Klassen%3F>

2 Kapitel 28 auf Seite 173

26.2 Destruktor

Im Gegensatz zum Konstruktor, gibt es beim Destruktor immer nur einen pro Klasse. Das liegt daran, dass ein Destruktor keine Parameter übergeben bekommt. Sein Aufruf erfolgt in der Regel implizit durch den Compiler bei der Zerstörung eines Objektes. Für den Anfang werden Sie mit dem Destruktor wahrscheinlich nicht viel anfangen können, da es mit den Mitteln, die Sie bis jetzt kennen, kaum nötig werden kann, dass bei der Zerstörung eines Objektes Aufräumarbeiten ausgeführt werden. Das folgende kleine Beispiel enthält einen Destruktor, der einfach gar nichts tut:

```
class A{
public:
    ~A(); // Deklaration Destruktor
};

A::~A(){ // Definition Destruktor
    // Hier könnten "Aufräumarbeiten" ausgeführt werden
}
```

Wenn Sie den Abschnitt über Speicherverwaltung³ gelesen haben, werden Sie wissen, wie nützlich der Destruktor ist. Im Moment reicht es, wenn Sie mal von ihm gehört haben. Auch über den Destruktor werden Sie im Kapitel „Leere Klassen?“⁴ weitere Informationen erhalten.

26.3 Beispiel mit Ausgabe

Um noch einmal deutlich zu machen, an welchen Stellen Konstruktor und Destruktor aufgerufen werden, geben wir einfach innerhalb der Methoden eine Nachricht aus:

```
#include <iostream>

class A{
public:
    A();          // Deklaration Konstruktor
    ~A();        // Deklaration Destruktor

    void print(); // Deklaration einer Methode
};

A::A() {        // Definition Konstruktor
    std::cout << "Innerhalb des Konstruktors" << std::endl;
}
```

3 <http://de.wikibooks.org/wiki/C%2B%2B-Programmierung%2F%20Inhaltsverzeichnis%23Anker%3ASpeicherverwaltung>

4 <http://de.wikibooks.org/wiki/C%2B%2B-Programmierung%2F%20Eigene%20Datentypen%20definieren%2F%20Leere%20Klassen%3F>

```
A::~A() { // Definition Destruktor
    std::cout << "Innerhalb des Destruktors" << std::endl;
}

void A::print() { // Definition der print()-Methode
    std::cout << "Innerhalb der Methode print()" << std::endl;
}

int main() {
    A objekt; // Anlegen des Objekts == Konstruktoraufruf
    objekt.print(); // Aufruf der Methode print()
    return 0;
} // Ende von main(), objekt wird zerstört == Destruktoraufruf
```

Ausgabe

```
Innerhalb des Konstruktors
Innerhalb der Methode print()
Innerhalb des Destruktors
```

26.4 Explizite Konstruktoren

Bevor wir den Begriff des expliziten Konstruktors klären, soll kurz auf die oben angesprochene zweite syntaktische Möglichkeit zum Konstruktoraufruf eingegangen werden.

```
class Bruch{
public:
    Bruch(int z = 0, int n = 1);
// ...
};

int main() {
    Bruch objekt1 = Bruch(7, 10); // Bruch objekt1(7, 10);
    Bruch objekt2 = Bruch(7); // Bruch objekt1(7);
    Bruch objekt3 = Bruch(); // Bruch objekt1;
}
```

Dieser Variante einen Konstruktor aufzurufen ist absolut gleichwertig zu der, die im Kommentar steht. Beachten Sie an dieser Stelle, dass hier keineswegs ein Kopierkonstruktor aufgerufen wird, es handelt sich tatsächlich nur um eine andere syntaktische Möglichkeit. Im folgenden Beispiel wird eine Klasse String definiert, welche über 2 Konstruktoren verfügt.

```
#include <iostream>

class String{
```

```

public:
    String(char const* zeichenkette);
    String(int anzahl, char zeichen = '*');

    char feld[80];
};

String::String(char const* zeichenkette){
    char* i = feld;
    while(*zeichenkette != '\0'){
        *i++ = *zeichenkette++;
    }
    *i = '\0';
}

String::String(int anzahl, char zeichen){
    for(int i = 0; i < anzahl ;++i)
        feld[i] = zeichen;
    feld[anzahl] = '\0';
}

int main(){
    String str1 = "Ein Text";
    String str2 = String(5, 'Z');
    String str3 = 'A';

    std::cout << str1.feld << std::endl;
    std::cout << str2.feld << std::endl;
    std::cout << str3.feld << std::endl;
}

```

Ausgabe

Ein Text

ZZZZZ

Auf die Implementierung der Konstruktoren soll an dieser Stelle nicht eingegangen werden und natürlich wäre `feld` in einer echten Klasse privat. Der erste Konstruktor erzeugt ein Objekt anhand einer Zeichenkette. Der zweite Konstruktor kann mit einem oder mit zwei Parametern aufgerufen werden. Er erzeugt einen String der `anzahl` viele `zeichen` enthält.

Sicher wird Ihnen auffallen, dass für die Initialisierung von `str1` und `str2` zwar das Gleichheitszeichen verwendet wird, dahinter jedoch kein Objekt der Klasse `String` folgt. Für diese syntaktische Möglichkeit muss der Compiler eine sogenannte implizite Konvertierung durchführen. Falls die Klasse über einen Konstruktor mit einem Parameter verfügt, der zu dem des übergebenen Objekts kompatibel ist, wird dieser Konstruktor verwendet. Kompatibel bedeutet, das sich das übergebene Objekt in maximal einem Schritt in ein Objekt mit dem Typ des Parameters umwandeln lässt. Bei `str3` wird ein Objekt vom Typ `char const` übergeben. Der Compiler findet in der Klasse einen Konstruktor der mit einem `int` aufgerufen werden kann und ein `char` lässt sich direkt in einen

`int` umwandeln. Entsprechend wählt der Compiler diesen Konstruktor aus. Analog wird für `str1` ein Objekt vom Typ `char const[9]` übergeben und dieses ist zu `char const*` kompatibel.

Es ist im Falle von `str3` jedoch sehr wahrscheinlich, das eigentlich ein String definiert werden sollte, der aus dem einzelnen Zeichen 'A' besteht und der Programmierer nur übersehen hat, das die Klasse gar nicht über einen entsprechenden Konstruktor verfügt. Das Problem wäre gelöst, wenn die implizite Konvertierung nicht stattfinden würde und genau das ist bei expliziten Konstruktoren der Fall. Das Schlüsselwort für die Deklaration lautet `explicit`.

```
#include <iostream>

class String{
public:
    String(char const* zeichenkette);
    explicit String(int anzahl, char zeichen = '*');

    char feld[80];
};

int main(){
    String str1 = "Ein Text"; // implizite Konvertierung
    String str3 = 'A';       // Fehler, keine implizite Konvertierung möglich
}
```

In den vielen Fällen ist, wie bei `str1`, eine implizite Konvertierung gewünscht, da sie zur besseren Lesbarkeit des Quellcodes beitragen kann. Beachten Sie auch das die folgenden beiden Zeilen explizite Konstruktoraufrufe sind und entsprechend durchgeführt werden:

```
String str4('A');           // ist explizit
String str5 = String('A'); // ist explizit
```

Die Möglichkeit eine Konstruktor mittels impliziter Konvertierung aufzurufen, sollte immer dann genutzt werden, wenn das Objekt, das Sie zu Erstellen beabsichtigen, den übergeben Wert repräsentiert. In unserem String-Beispiel ist dies bei `str1` der Fall. Das erzeugte Objekt repräsentiert den übergebenen String. Im Falle von `str2` wird hingegen ein Objekt nach den Vorgaben der übergeben Parameter erstellt. Der Unterschied ist rein sprachlicher Natur, im ersten Fall wird eben bereits ein String (in Form eines C-Strings) übergeben. Technisch gesehen wird natürlich in beiden Fällen ein Objekt anhand der Parameter konstruiert.

Wenn Sie nun beabsichtigen einen String zu erstellen, der aus dem Zeichen 'A' besteht, würde das erstellte Objekt des Zeichen 'A' repräsentieren. Wenn Sie sich infolgedessen für die Anwendung einer impliziten Konvertierung wie bei `str3` entscheiden, wird der Compiler dies dank explizitem Konstruktor mit einer Fehlermeldung quittieren. Wenn Sie beabsichtigen einen String mit 'A' vielen Zeichen zu konstruieren, dann wählen Sie die explizite Syntax wie bei `str4` oder `str5`. Dies wird Ihr Compiler anstandslos übersetzen.

27 Privat und öffentlich

Mit den Schlüsselwörtern `public` und `private` wird festgelegt, von wo aus auf ein Mitglied einer Klasse zugegriffen werden kann. Auf Member, die als `public` deklariert werden, kann von überall aus zugegriffen werden, sie sind also *öffentlich* verfügbar. `private`-deklarierte Member lassen sich nur innerhalb der Klasse ansprechen, also nur innerhalb von Methoden derselben Klasse. Typischerweise werden Variablen `private` deklariert, während Methoden `public` sind. Eine Ausnahme bilden Hilfsmethoden, die gewöhnlich im `private`-Bereich deklariert sind, wie etwa die `init()`-Methode, die von verschiedenen Konstruktoren aufgerufen wird, um Codeverdopplung zu vermeiden.

Standardmäßig sind die Member einer Klasse `private`. Geändert wird der Sichtbarkeitsbereich durch die beiden Schlüsselwörter gefolgt von einem Doppelpunkt. Alle Member, die darauffolgend deklariert werden, fallen in den neuen Sichtbarkeitsbereich.

```
class A{
    // private Member

public:
    // öffentliche Member

private:
    // private Member
};
```

Zwischen den Sichtbarkeitsbereichen kann somit beliebig oft gewechselt werden. Der implizit `private` Bereich sollte aus Gründen der Übersichtlichkeit nicht genutzt werden. In diesem Buch folgen wir der Regel, als erstes die öffentlichen Member zu deklarieren und erst danach die privaten. Die umgekehrte Variante findet jedoch ebenfalls eine gewisse Verbreitung. Entscheiden Sie sich für eine Variante und wenden Sie diese konsequent an.

In Zusammenhang mit Vererbung werden Sie noch einen weiteren Sichtbarkeitsbereich kennen lernen. Dieser ist für unsere Zwecke aber noch nicht nötig und wird daher auch erst später erläutert.

27.1 Strukturen

Neben den Klassen gibt es in C++ auch noch die aus C stammenden Strukturen. Eine Struktur, auch Datenverbund genannt, fasst mehrere Variablen zu einem Typ zusammen. Im Gegensatz zu den C-Strukturen sind C++-Strukturen allerdings vollständig kompatibel zu Klassen. Sie können in

einer Struktur also problemlos Methoden deklarieren oder die Zugriffsrechte über die Schlüsselworte `public`, `protected` (wird später erläutert) und `private` festlegen.

Der einzige wirkliche Unterschied zwischen Strukturen und Klassen ist in C++, dass Klassen implizit `private`, während Strukturen implizit `public` sind. Da dieser implizite Zugriffsrechtebereich aber für gewöhnlich nicht genutzt werden sollte, ist dieser Unterschied fast unbedeutend. Das folgende kleine Beispiel zeigt die Definition einer Struktur im Vergleich zu der einer Klasse.

```
struct A{                               | class B{
    // öffentliche Member                | // private Member
                                         |
public:                                  | public:
    // öffentliche Member                | // öffentliche Member
                                         |
private:                                 | private:
    // private Member                    | // private Member
};                                       | };
```

Thema wird später näher erläutert...

Im Kapitel über Vererbung^a werden Sie noch sehen, dass Sie eine Klasse problemlos von einer Struktur ableiten können und umgekehrt.

^a Kapitel 39 auf Seite 229

28 Operatoren überladen

Im Kapitel über Zeichenketten haben Sie gelernt, dass es sich bei `std::string` um eine Klasse handelt. Dennoch war es Ihnen möglich, mehrere `std::string`-Objekte über den `+`-Operator zu verknüpfen oder einen String mittels `=` bzw. `+=` an einen `std::string` zuzuweisen bzw. anzuhängen. Der Grund hierfür ist, dass diese Operatoren für die `std::string`-Klasse überladen wurden. Das heißt, ihnen wurde in Zusammenhang mit der Klasse `std::string` eine neue Bedeutung zugewiesen.

Vorweg sollten Sie auf jeden Fall eines beherzigen. Überladen Sie Operatoren nur, wenn beim Operatoraufruf intuitiv klar ist, was passiert. Ist dies nicht der Fall, dann schreiben Sie besser eine einfache Methode mit einem aussagekräftigen Namen. Der Sinn der Operatorüberladung ist die Erhöhung der Übersichtlichkeit des Quellcodes. Wenn man erst in der Dokumentation nachschlagen muss, um herauszufinden, was ein überladener Operator tut, dann haben Sie dieses Ziel verfehlt. Es muss natürlich nicht im Detail klar sein, welchen Effekt der Operator hat, aber man sollte intuitiv eine ungefähre Vorstellung haben, wenn man Code liest, in dem ein überladener Operator verwendet wurde. Beispielweise kann für eine Klasse, die eine komplexe Zahl repräsentiert, nicht sofort eindeutig klar sein, wie diese als Text ausgegeben wird, aber man sollte immer am Methodennamen oder eben am Operator erkennen, dass hier eine Ausgabe stattfindet.

Um Operatoren überladen zu können, müssen Sie zunächst einmal wissen, dass diese in C++ im Prinzip nur eine spezielle Schreibweise für Funktionen sind. Das folgende kleine Beispiel demonstriert den Aufruf der überladenen `std::string` in ihrer Funktionsform:

```
#include <iostream>
#include <string>

int main(){
    std::string a = "7", b = "3", c;

    c = a + b;
    std::cout << c << std::endl;

    c.operator=(operator+(a,b));
    std::cout << c << std::endl;
}
```

Ausgabe

73

73

Wie Ihnen beim Lesen des Beispiels vielleicht schon aufgefallen ist, wurde der Zuweisungsoperator als Methode von `std::string` überladen. Der Verkettungsoperator ist hingegen als gewöhnliche

Funktion überladen. Dieses Vorgehen ist üblich, aber nicht zwingend. Die Deklarationen der beiden Operatoren sehen folgendermaßen aus:

```
class std::string{
    // Zuweisungsoperator als Member von std::string
    std::string& operator=(std::string const& assign_string);
};

// Verkettungsoperator als globale Funktion
std::string operator+(std::string const& lhs, std::string const& rhs);
```

Da es sich bei `std::string` um ein `typedef` von einer Templateklasse handelt, sehen die Deklarationen in der C++-Standardbibliothek noch etwas anders aus, das ist für unser Beispiel aber nicht relevant. Auf die Rückgabetypen der verschiedenen Operatoren wird später in diesem Kapitel noch näher eingegangen.

Wenn Sie den Zuweisungsoperator als globale Funktion überladen wollten, müssten Sie ihm zusätzlich als ersten Parameter eine Referenz auf ein `std::string`-Objekt übergeben. Würden Sie den Verkettungsoperator als Methode der Klasse deklarieren wollen, entfielen der erste Parameter. Er würde durch das Objekt ersetzt, für das der Operator aufgerufen wird. Da die Operatorfunktion nichts an dem Objekt ändert, für das sie aufgerufen wird, ist sie außerdem konstant. Allerdings hätten Sie an dieser Stelle ein Problem, sobald Sie einen `std::string` zum Beispiel mit einem C-String verknüpfen wollen.

Bei einer Operatorüberladung als Methode ist der erste Operand immer ein Objekt der Klasse. Für den Zuweisungsoperator ist das in Ordnung, da ja immer etwas an einen `std::string` zugewiesen werden soll. Wenn Sie also folgende Operatorüberladungen in der Klasse vornahmen, werden Sie bei der Nutzung der Klasse schnell auf Probleme stoßen.

```
class std::string{
    // Zuweisung eines anderen std::string's
    std::string& operator=(std::string const& assign_string);
    // Zuweisung eines C-Strings
    std::string& operator=(char const assign_string[]);

    // Verkettung mit anderen std::string's
    std::string operator+(std::string const& rhs)const;
    // Verkettung mit C-Strings
    std::string operator+(char const rhs[])const;
};

int main(){
    std::string std_str_1 = "std_1";
    std::string std_str_2 = "std_2";
    char        c_str_1[80] = "c_str";

    std_str_1 = std_str_2; // Geht
```

```

    std_str_1 = c_str_1;    // Geht
// c_str_1    = std_str_1; // Geht nicht (soll es auch nicht!)
    std_str_1 + std_str_2; // Geht
    std_str_1 + c_str_1;   // Geht
// c_str_1    + std_str_2; // Geht nicht (sollte es aber)
}

```

Die in der Klasse deklarierten Formen lassen sich problemlos nutzen. Wenn wir aber wollen, dass man auch „C-String + std::string“ schreiben kann, müssen wir noch eine weitere Operatorüberladung global deklarieren. Da dies jedoch uneinheitlich aussieht und obendrein die Klassendeklaration unnötig aufbläht, deklariert man alle Überladungen des Verkettungsoperators global. Das Ganze sieht dann folgendermaßen aus.

```

std::string operator+(std::string const& lhs, std::string const& rhs);
std::string operator+(std::string const& lhs, char const rhs[]);
std::string operator+(char const lhs[],      std::string const& rhs);

```

28.1 Definition der überladenen Operatoren

Im Folgenden werden Sie einige beispielhafte Implementierungen für verschiedene Operatoren kennenlernen, wobei wir uns weiterhin an der `std::string`-Klasse orientieren möchten. Vergewärtigen wir uns zunächst, welche Operatoren für diese Klasse überladen sind. Bereits erwähnt wurden der Zuweisungsoperator (=) und der Verknüpfungsoperator (+), infolge dessen ist auch die verknüpfende Zuweisung (+=) überladen. Weiterhin stehen die Vergleichsoperatoren (==, !=, <, <=, > und >=) und ebenso wie der Ein- (>>) und Ausgabeoperator (<<) bereit. Außerdem steht der Zugriffoperator in einer konstanten und einer nicht-konstanten Version zur Verfügung.

Für die folgenden Beispiele nehmen wir an, dass unser String in einem Array von chars mit 1024 Elementen verwaltet wird. Dementsprechend werden wir als Klassenname ab sofort nicht mehr `std::string`, sondern `MyString` verwenden. Nachdem Sie die dynamische Speicherverwaltung kennengelernt haben, können Sie auch mal versuchen, eine eigene Stringklasse zu schreiben, welche nicht auf eine festgelegte Zeichenmenge begrenzt ist. Der Einfachheit halber werden wir im Folgenden keine Grenzprüfung für unser `char`-Array vornehmen. Wenn Sie also vorhaben Ihre Objekte mit langen Strings zu füllen, dann achten Sie darauf, dass Ihr Array hinreichend groß ist.

Als Erstes müssen wir uns überlegen, ob wir die Operatoren als Klassenmember oder extern deklarieren. Allgemein gilt für Zuweisungsoperatoren, ob nun zusammengesetzt oder nicht, dass Sie als Klassenmember deklariert sind. Vergleichs- und Verknüpfungsoperatoren werden als Klassenmember deklariert, wenn Sie ausnahmslos mit Objekten der gleichen Klasse benutzt werden, andernfalls werden Sie extern deklariert. Ein Zugriffoperator liefert üblicherweise einen Wert aus der Klasse oder erlaubt sogar direkten Zugriff, somit ist er natürlich ein Klassenmember. Ein- und Ausgabeoperatoren sind extern, aber darauf wird später, wie schon erwähnt, noch explizit eingegangen. Im Folgenden werden einmal alle Deklarationen aufgeführt, die wir für unser Beispiel implementieren möchten, inklusive der (sehr kurz gefassten) Klassendefinition von `MyString`.

```
#include <cstddef>

class MyString{
public:
    static std::size_t const max_length = 1024;

    MyString();
    MyString(MyString const& string);
    MyString(char const*    c_string);

    MyString& operator=(MyString const& string);
    MyString& operator=(char const*    c_string);
    MyString& operator+=(MyString const& string);
    MyString& operator+=(char const*    c_string);

    char const& operator[](std::size_t index) const;
    char&      operator[](std::size_t index);

    std::size_t length() const{ return length_; }

private:
    char      data_[max_length];
    std::size_t length_;

    void selfcopy(MyString const& string);
    void selfcopy(char const*    c_string);
};

MyString operator+(MyString const& lhs, MyString const& rhs);
MyString operator+(MyString const& lhs, char const rhs[]);
MyString operator+(char const lhs[],    MyString const& rhs);

bool operator==(MyString const& lhs, MyString const& rhs);
bool operator==(MyString const& lhs, char const rhs[]);
bool operator==(char const lhs[],    MyString const& rhs);

bool operator!=(MyString const& lhs, MyString const& rhs);
bool operator!=(MyString const& lhs, char const rhs[]);
bool operator!=(char const lhs[],    MyString const& rhs);

bool operator<(MyString const& lhs, MyString const& rhs);
bool operator<(MyString const& lhs, char const rhs[]);
bool operator<(char const lhs[],    MyString const& rhs);

bool operator<=(MyString const& lhs, MyString const& rhs);
bool operator<=(MyString const& lhs, char const rhs[]);
bool operator<=(char const lhs[],    MyString const& rhs);
```

```

bool operator>(MyString const& lhs, MyString const& rhs);
bool operator>(MyString const& lhs, char const rhs[]);
bool operator>(char const lhs[], MyString const& rhs);

bool operator>=(MyString const& lhs, MyString const& rhs);
bool operator>=(MyString const& lhs, char const rhs[]);
bool operator>=(char const lhs[], MyString const& rhs);

```

Als Erstes werden wir uns kurz Gedanken um unsere Datenstruktur machen. Die Zeichen werden, wie gesagt, in der Membervariable `data_` gespeichert. `length_` speichert die Anzahl der momentan in `data_` gültigen Zeichen. Der Standardkonstruktor setzt einfach `length_` auf 0 und erstellt somit einen leeren String. Der Kopierkonstruktor übernimmt die Stringlänge von dem an ihn übergebenen Argument und nutzt `std::memcpy`, um die benötigten Arrayelemente zu kopieren. Natürlich könnte man die Arrays auch in einer `for`-Schleife durchlaufen und die Elemente einzeln kopieren, aber `std::memcpy` ist um einiges schneller, da hier gleich ganze Blöcke von Elementen kopiert werden. Der verbleibende Konstruktor erhält einen C-String (nullterminierter String, wird also durch ein Nullzeichen abgeschlossen) und verfährt somit fast analog, mit dem Unterschied, dass ganz am Anfang erst einmal die Stringlänge ermittelt werden muss.

```

#include <cstring>

MyString::MyString():length_(0){}

MyString::MyString(MyString const& string):
    length_(string.length_)
{
    std::memcpy(data_, string.data_, length_*sizeof(data_[0]));
}

MyString::MyString(char const* c_string){
    length_ = max_length - 1;
    for(std::size_t i = 0; i < max_length; ++i){
        if(c_string[i] != '\0'){
            continue;
        }

        length_ = i;
        break;
    }

    std::memcpy(data_, c_string, length_*sizeof(data_[0]));
}

```

Die Funktion `std::memcpy()` erwartet 3 Argumente, als Erstes die Zielspeicheradresse, auf die geschrieben werden soll, als Zweites die Quelladresse, von der gelesen werden soll und schließlich

die Anzahl der Bytes, die kopiert werden sollen. Die Zeigertypen werden hierbei implizit nach `void*` umgewandelt, daher ist die Funktion mit einiger Umsicht zu verwenden. Die Multiplikation von `length_` mit `sizeof(data_[0])` ist eigentlich nicht nötig, da ein `char` immer die Größe 1 Byte hat. Es wird empfohlen in Zusammenhang mit der Funktion `memcpy` dennoch immer die Datentypgröße durch Multiplikation von `sizeof(Variablenname)` zu berücksichtigen, da sich auf diese Weise der Typ von Variable ändern kann, ohne dass eine Änderung des folgenden Codes nötig ist. So könnte in diesem Beispiel problemlos der Typ von `char` auf `wchar_t` geändert werden. Beachten Sie, dass wirklich die Variable und nicht deren Typ an `sizeof` übergeben wird.

Möglicherweise wundern Sie sich ein wenig über den Schleifenaufbau im letzten Konstruktor. Den Rest der Schleife zu überspringen, sobald eine Bedingung erfüllt ist, ist eine übliche Designvariante um Quellcode übersichtlicher zu gestalten. Es vermeidet eine zusätzlich Codeeintrückung und drückt deutlich aus, dass dies eine Abbruchbedingung ist und sofort mit dem nächsten Schleifendurchlauf fortgefahren wird.

Da die Konstruktoren in diesem Beispiel das gleiche tun wie die Zuweisungsoperatoren und in den Konstruktoren weder etwas besonderes zu beachten, noch großartige Optimierungsmöglichkeiten bestehen, werden wir den Code für beide zusammenfassen. Dies geschieht in einer privaten Hilfsfunktion. Der Rückgabewert der Zuweisungsoperatoren ist immer eine Referenz auf das aktuelle Objekt, was wir durch ein `return *this` realisieren.

```
#include <cstring>

MyString::MyString(MyString const& string){
    selfcopy(string);
}

MyString::MyString(char const* c_string){
    selfcopy(c_string);
}

MyString& MyString::operator=(MyString const& string){
    selfcopy(string);
    return *this;
}

MyString& MyString::operator=(char const* c_string){
    selfcopy(c_string);
    return *this;
}

void MyString::selfcopy(MyString const& string){
    length_ = string.length_;
    std::memcpy(data_, string.data_, length_*sizeof(data_[0]));
}

void MyString::selfcopy(char const* c_string){
    length_ = max_length - 1;
```

```

for(std::size_t i = 0; i < max_length; ++i){
    if(c_string[i] != '\0'){
        continue;
    }

    length_ = i;
    break;
}

std::memcpy(data_, c_string, length_*sizeof(data_[0]));
}

```

Beachten Sie, dass Sie nicht versuchen sollten, einen Konstruktor durch den Aufruf des Zuweisungsoperators oder umgekehrt zu implementieren. Speziell für unser Beispiel würde zwar beides funktionieren, wobei Sie die Syntax für letzteres noch nicht kennen, aber sobald die Klasse in irgendeiner Weise erweitert wird, könnte sich dadurch undefiniertes Verhalten ergeben. Die Auslagerung der gemeinsamen Arbeit erhöht außerdem die Lesbarkeit des Quellcodes und ist ein allgemein übliches Vorgehen um Codeverdopplung zu vermeiden.

Für die verknüpfende Zuweisung implementieren wir zunächst die Version, die wieder ein Objekt unserer Klasse übernimmt. Wir kopieren den Inhalt des übergebenen Strings an das derzeitige Ende (also die durch `length_` gekennzeichnete Stelle) des aktuellen Strings und addieren anschließend die Länge des übergebenen zu der des aktuellen Strings.

Die andere verknüpfende Zuweisung übernimmt einen C-String als Argument. Hier können und sollten wir uns die Implementierung wieder leicht machen. Wir wandeln den C-String in einen `MyString` um, indem wir ein temporäres Objekt durch einen Aufruf des entsprechenden Konstruktors erstellen. Anschließend rufen wir mit diesem temporären Objekt die eben erstellte verknüpfende Zuweisung auf. Wir nutzen also den vorhandenen Code zweier anderer Methoden der Klasse und vermeiden so Codeverdopplung. Gleichzeitig hat das noch den angenehmen Nebeneffekt, dass wir weniger Schreibarbeit haben. Der Rückgabewert ist bei beiden Versionen wieder eine Referenz auf das aktuelle Objekt.

```

#include <cstring>

MyString& MyString::operator+=(MyString const& string){
    std::memcpy(
        data_ + length_,           // Adresse des Datenfeldes + Anzahl vorhandener
        Zeichen
        string.data_,             // Adresse des Datenfeldes des Arguments
        string.length_*sizeof(data_[0]) // Anzahl der Zeichen im Datenfeld des Arguments
    );
    length_ += string.length_;    // Längen addieren

    return *this;
}

MyString& MyString::operator+=(char const* c_string){

```

```
    return *this += MyString(c_string);  
}
```

Wie Sie sehen ist die Implementierung der zweiten Operatorvariante mit einer Zeile wirklich sehr kompakt. Da die Rückgabe der anderen verknüpfenden Zuweisung mit jener Version identisch ist, kann diese durch ein `return` vor dem Aufruf einfach weitergegeben werden.

Da wir nun die verknüpfende Zuweisung zur Verfügung haben, wenden wir uns als Nächstes dem Verknüpfungsoperator an sich zu. Da bei diesem der erste Operand nicht zwingendermaßen ein Objekt von `MyString` sein muss, ist dieser Operator außerhalb der Klasse deklariert. Daraus ergibt sich für uns leider auch ein Nachteil, denn wir haben in den Operatorimplementierungen keinen Zugriff auf `private`- und `protected`-Member der Klasse. Allerdings benötigen wir diesen letztlich auch gar nicht, denn wir werden wiederum vorhandenen Code für die Implementierung benutzen.

Der Rückgabewert dieses Operators ist das aus der Verknüpfung resultierende Objekt der Klasse `MyString`. Die Strategie, die wir für die Implementierung benutzen, sieht so aus, dass wir den Verknüpfungsaufruf durch einen Aufruf der verknüpfenden Zuweisung ausdrücken. Da bei dieser aber immer das „erste Argument“ (also das Objekt, für das die Operatormethode aufgerufen wird) modifiziert wird, müssen wir zunächst eine Kopie unseres ersten Arguments erstellen. Für diese Kopie können wir dann die verknüpfende Zuweisung mit unserem zweiten Argument als Argument aufrufen. Da dieser Aufruf wiederum eine Referenz auf unsere Kopie zurück gibt, können wir diese sofort weitergeben, denn die Kopie ist nun das resultierende Objekt unserer Verknüpfung.

```
MyString operator+(MyString const& lhs, MyString const& rhs){  
    return MyString(lhs) += rhs;  
}  
  
MyString operator+(MyString const& lhs, char const rhs[]){  
    return MyString(lhs) += rhs;  
}  
  
MyString operator+(char const lhs[], MyString const& rhs){  
    return MyString(lhs) += rhs;  
}
```

Wie Sie sehen ist die Implementierung wesentlich einfacher, als die langatmige Erklärung eben vielleicht vermuten ließ. Die Aufrufhierarchie sieht nun folgendermaßen aus:

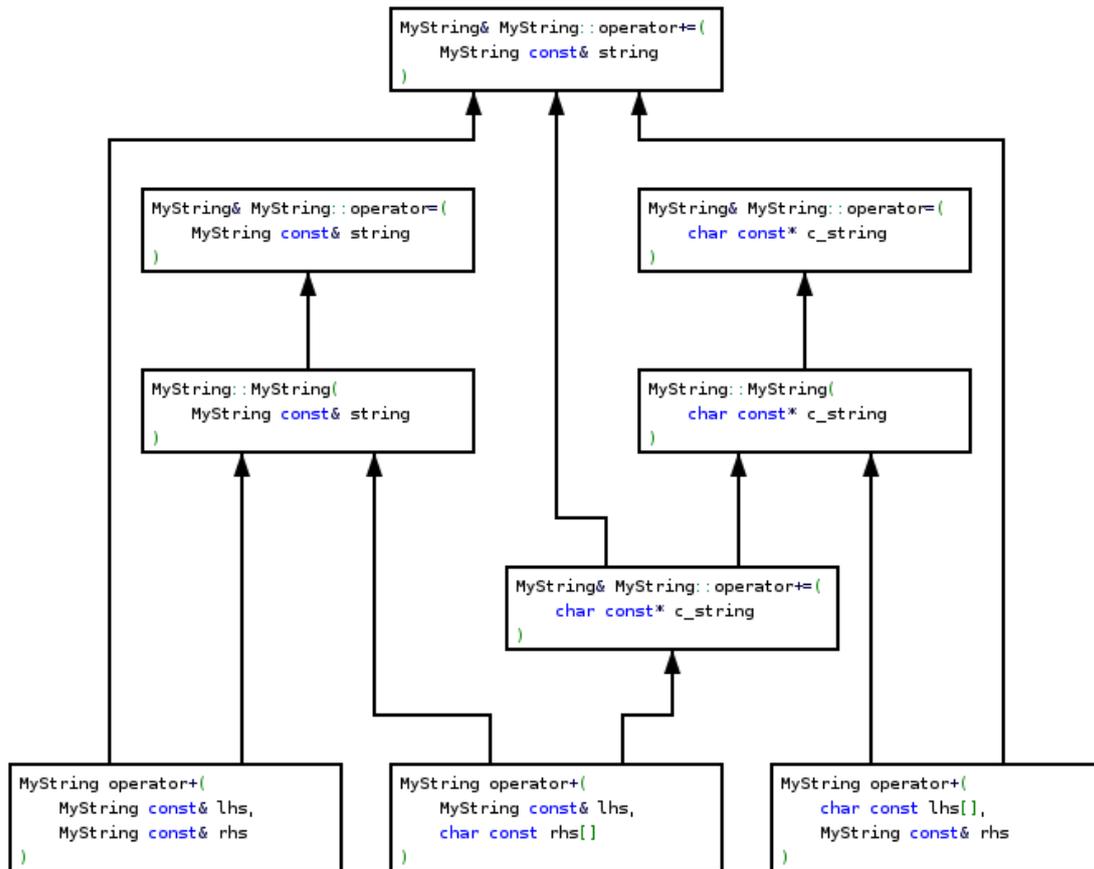


Abb. 3

28.1.1 Zugriffoperator

Den Zugriffoperator gibt es in einer konstanten und einer nicht-konstanten Version. In Abhängigkeit davon, ob das konkrete Objekt für das er aufgerufen wird konstant ist oder nicht, entscheidet der Compiler, welche Version aufgerufen wird. Die beiden Versionen unterscheiden sich ausschließlich im Rückgabebetyp. Die konstante Version liefert eine Referenz auf ein konstantes Zeichen, während die nicht-konstante Version eine Referenz auf ein Zeichen liefert und somit schreibenden Zugriff auf das gewählte Zeichen bietet. Wir werden uns zunächst um die konstante Version kümmern.

Der Zugriffoperator übernimmt üblicherweise einen Index. In unserem Fall ist dies eine positive ganze Zahl, über welche Zugriff auf das entsprechende Zeichen unserer Zeichenkette gewährt wird. Wir überprüfen ob der Index innerhalb unserer Zeichenkette liegt. Da der Index positiv sein soll, brauchen wir den negativen Bereich nicht zu prüfen, lediglich wenn der Index größer ist, als der `MyString` lang, müssen wir etwas unternehmen. Üblicherweise würde man in einem solchen Fall eine Exception werfen, da dies aber noch nicht behandelt wurde, werden wir stattdessen das letzte Zeichen unseres `MyString`s zurückliefern. Ist der `MyString` leer, so liefern wir das Zeichen an der Position 0, obgleich dieses eigentlich nicht gültig ist.

```
char const& MyString::operator[](std::size_t index) const{
    if(length_ == 0)
        return data_[0];

    if(index >= length_)
        return data_[length_ - 1];

    return data_[index];
}
```

Wie bereits erwähnt, ist die Implementierung für die nicht-konstante Version identisch, daher wäre es sinnvoll, wenn wir diese die konstante Version aufrufen und anschließend den Rückgabotyp modifizieren lassen könnten. Tatsächlich ist dies möglich, da dies jedoch ein wenig Wissen über die verschiedenen Möglichkeiten zur Typumwandlung in C++ erfordert, wird die Technik hier lediglich verwendet und nicht erläutert. Wenn Sie mehr darüber erfahren möchten, lesen Sie das Kapitel „Casts“¹.

```
char& MyString::operator[](std::size_t index){
    return const_cast< char& >(
        static_cast< MyString const& >(*this)[index]
    );
}
```

28.1.2 Vergleichsoperatoren

Die 6 Vergleichsoperatoren sollen `MyString`-Objekte untereinander, sowie in Kombination mit C-Strings vergleichen. Somit werden Sie außerhalb der Klasse deklariert, da der erste Parameter eines Vergleichs ein C-String sein kann. Es sind insgesamt 18 Überladungen (6 Operatoren, mit je 3 Prototypen) erforderlich um alle möglichen Vergleiche abzudecken. Wie eben schon gezeigt werden wir auch in diesem Fall Codeverdopplung soweit es geht vermeiden, indem sich die Operatoren gegenseitig aufrufen. Alle Vergleichsoperatoren liefern einen logischen Wert (`true` oder `false`) zurück.

Als Erstes beschäftigen wir uns mit dem Gleichheitsoperator `==`. Zwei `MyStrings` sollen gleich sein, wenn Sie die gleiche Länge haben und alle enthaltenen Zeichen gleich sind. Die Implementierung überprüft entsprechend diese beiden Bedingungen, wobei ein Vergleich der einzelnen Zeichen natürlich nur nötig (und möglich) ist, wenn die Länge übereinstimmt. Dementsprechend ist die Überprüfung dieser Bedingung natürlich auch der erste Schritt.

Die Versionen die einen C-String übernehmen, erstellen aus dem C-String ein temporäres `MyString`-Objekt und rufen anschließend die Überladung auf, die zwei `MyString`-Objekte übernimmt. Nur für diese eine Version werden wir den vergleichenden Code schreiben.

¹ <http://de.wikibooks.org/wiki/C%2B%2B-Programmierung%2F%20N%C3%9Ctzliches%2F%20Casts>

```

bool operator==(MyString const& lhs, MyString const& rhs){
    if(lhs.length() != rhs.length()){
        return false;
    }

    for(std::size_t i = 0; i < lhs.length(); ++i){
        if(lhs[i] != rhs[i]){
            return false;
        }
    }

    return true;
}

bool operator==(MyString const& lhs, char const rhs[]){
    return lhs == MyString(rhs);
}

bool operator==(char const lhs[], MyString const& rhs){
    return MyString(lhs) == rhs;
}

```

Der Ungleichheitsoperator `!=` liefert als Umkehrfunktion immer den genau den gegenteiligen Wert des Gleichheitsoperators. Entsprechend werden wir einfach immer den Gleichheitsoperator aufrufen und das Ergebnis negieren.

```

bool operator!=(MyString const& lhs, MyString const& rhs){
    return !(lhs == rhs);
}

bool operator!=(MyString const& lhs, char const rhs[]){
    return !(lhs == rhs);
}

bool operator!=(char const lhs[], MyString const& rhs){
    return !(lhs == rhs);
}

```

Als Nächstes müssen wir uns überlegen, wann ein `MyString` kleiner als ein anderer sein soll. Die Länge ist dafür zunächst einmal uninteressant, wir schauen uns stattdessen die einzelnen Zeichen an. Wenn das erste Zeichen des ersten `MyString`s kleiner als das erste Zeichen des zweiten `MyString`s ist, so ist das gesamte erste `MyString`-Objekt kleiner als das Zweite. Ist das erste Zeichen größer, so ist entsprechend das gesamte erste `MyString`-Objekt größer. Sind beide Zeichen gleichgroß, so sehen wir uns entsprechend das jeweils zweite Zeichen der beiden Objekte an und vollführen den gleichen Vergleich. Dies setzen wir solange fort, bis in einem der beiden `MyString`s

keine Zeichen mehr zur Verfügung stehen. Sind bis zu dieser Position alle Zeichen gleich, so ist der kürzere String der kleinere. Sind beide `MyStrings` gleich lang, so sind sie insgesamt gleich.

Ist der erste `MyString` nicht kleiner, so ist er entsprechend größer oder gleich dem zweiten `MyString`. Die Umkehroperation zum Kleiner-Operator `<` ist dementsprechend der Größer-Gleich-Operator `>=`. Äquivalent dazu sind auch der Größer-Operator `>` und der Kleiner-Gleich-Operator `<=` entgegengesetzte Funktionen.

```
bool operator<(MyString const& lhs, MyString const& rhs){
    std::size_t min_length = std::min(lhs.length(), rhs.length());

    for(std::size_t i = 0; i < min_length; ++i){
        if(lhs[i] < rhs[i]){
            return true;
        }
        if(lhs[i] > rhs[i]){
            return false;
        }
    }

    if(lhs.length() >= rhs.length()){
        return false;
    }

    return true;
}

bool operator<(MyString const& lhs, char const rhs[]){
    return lhs < MyString(rhs);
}

bool operator<(char const lhs[], MyString const& rhs){
    return MyString(lhs) < rhs;
}

bool operator>=(MyString const& lhs, MyString const& rhs){
    return !(lhs < rhs);
}

bool operator>=(MyString const& lhs, char const rhs[]){
    return !(lhs < rhs);
}

bool operator>=(char const lhs[], MyString const& rhs){
    return !(lhs < rhs);
}
```

```

bool operator<=(MyString const& lhs, MyString const& rhs){
    return !(lhs > rhs);
}

bool operator<=(MyString const& lhs, char const rhs[]){
    return !(lhs > rhs);
}

bool operator<=(char const lhs[], MyString const& rhs){
    return !(lhs > rhs);
}

```

Was jetzt noch fehlt, ist die Implementierung für den Größer-Operator, auf den wir eben bereits in der Implementierung für den Kleiner-Gleich-Operator zurückgegriffen haben. Es gibt für die Implementierung zwei Möglichkeiten, zum einen könnten wir ihn als Kombination aus den anderen Operatoren implementieren, zum anderen könnten wir aber auch eine Implementierung schreiben, welche dem Kleiner-Operator ähnlich ist. Ersteres vermeidet Codeverdopplung, letzteres ist dafür deutlich performanter.

```

#include <algorithm> // Für std::min()

// Performante Implementierung
bool operator>(MyString const& lhs, MyString const& rhs){
    std::size_t min_length = std::min(lhs.length(), rhs.length());

    for(std::size_t i = 0; i < min_length; ++i){
        if(lhs[i] > rhs[i]){
            return true;
        }
        if(lhs[i] < rhs[i]){
            return false;
        }
    }

    if(lhs.length() <= rhs.length()){
        return false;
    }

    return true;
}

// Codeverdopplung vermeidene Implementierung
bool operator>(MyString const& lhs, MyString const& rhs){
    return !(lhs < rhs || lhs == rhs);
}

```

Die Implementierungen für die C-String-Varianten rufen in jedem Fall wieder die eben implementierte auf, um so Codeverdopplung zu vermeiden.

```
bool operator>(MyString const& lhs, char const rhs[]){
    return lhs > MyString(rhs);
}

bool operator>(char const lhs[], MyString const& rhs){
    return MyString(lhs) > rhs;
}
```

28.1.3 Ein-/Ausgabeoperator

Für den Ein- bzw. Ausgabeoperator müssen wir uns zunächst darüber im Klaren sein, wie die Ein- und Ausgabe in C++ typischerweise funktioniert. Alle Ein- und Ausgaben werden über Streams vorgenommen, für die Standardein-/ausgabe gibt es die Objekt `std::cin` (Klasse `std::istream`), `std::cout`, `std::cerr` und `std::clog` (alle Klasse `std::ostream`). Wie genau diese Objekte die Ein-/Ausgabe letztlich verarbeiten, muss uns an dieser Stelle nicht kümmern. Objekte der Klassen `std::ifstream` bzw. `std::ofstream` kümmern sich um die Ein-/Ausgabe bezüglich Dateien und mit `std::istringstream` und `std::ostringstream` stehen uns zwei Klassen für temporäre Speicherbereiche zur Verfügung. Die 4 letztgenannten Klassen sind von `std::istream` oder `std::ostream` abgeleitet, daher müssen wir uns nicht explizit um sie kümmern.

Als Nächstes muss uns klar sein, wie die entsprechenden Prototypen für die Operatoren aussehen müssen. Was wir erreichen wollen, ist eine Ein-/Ausgabe mit folgender Syntax:

```
MyString string_1, string_2;
std::cin >> string_1 >> string_2;           // Eingabe zweier MyStrings
std::cout << string_1 << string_2 << std::endl; // Ausgabe zweier MyStrings + Zeilenumbruch
```

Es werden gleich 2 Objekte gelesen bzw. geschrieben, um auch eine Sinnvolle Aussage über den Rückgabedatentyp treffen zu können. Wenn wir Klammern für die beiden Ausdrücke setzen, wird offensichtlich, welche Typen die binären Operatoren als Parameter erhalten müssen.

```
MyString string_1, string_2;
((std::cin >> string_1) >> string_2);
(((std::cout << string_1) << string_2) << std::endl);
```

Wie in der Mathematik wird die innerste Klammer zuerst aufgelöst. Der `operator>>` erhält als Parametertypen also einen `std::istream` und einen `MyString`. Da wir etwas von `std::istream` lesen wollen und es dabei aus dem Eingabepuffer entfernt werden muss, damit es später nicht erneut eingelesen wird, müssen wir eine Referenz auf `std::istream` übergeben. `MyString` muss bei der Eingabe ebenfalls verändert werden, somit ist auch das `MyString`-Objekt

als Referenz zu übergeben. Wenn wir uns nun der zweiten Klammer zuwenden, kennen wir bereits die beiden Paramertypen. Da der erste Parameter die Rückgabe der ersten Klammer ist, folgt logisch, dass unser Ausgabeoperator als Rückgabetyt den Typ des ersten Parameters (Referenz auf `std::istream`) haben muss.

Für die Ausgabe gilt ähnliches, allerdings soll `MyString` hierbei nicht verändert werden. Wir können somit wahlweise eine Kopie oder eine Referenz auf `const MyString` übergeben. Da Letzteres effizienter ist, wählen wir dies. In unserem Beispiel oben gibt es für die Ausgabe noch eine dritte Klammer. Diese erhält als zweiten Parameter kein `MyString`-Objekt sondern `std::endl`. Folglich muss es auch mit diesen Prototypen eine Überladung geben, sie befindet sich in der Headerdatei `ostream` welche von `iostream` eingebunden wird.

Allgemein lässt sich somit also für die Ein-/Ausgabe sagen, der erste Parameter ist ebenso wie der Rückgabetyt immer eine Referenz auf `std::istream` bzw. `std::ostream` und der zweite Parameter hat den Typ, für den die Operatoren überladen werden sollen. Für die Eingabe wird als zweiter Parameter eine Referenz übergeben, für die Ausgabe eine Kopie oder typischerweise eine Referenz auf `const`. In unserem konkreten Fall sehen die Prototypen somit so aus:

```
#include <iostream> // Am begin der Headerdatei einbinden
std::istream& operator>>(std::istream& is, MyString& rhs);
std::ostream& operator<<(std::ostream& os, MyString const& rhs);
```

Tip

Wenn Sie die Deklarationen in einer Headerdatei und die Funktionsdefinitionen in einer Quell-datei machen, so genügt es im Header die Datei `iosfwd` einzubinden und erst in der Quell-datei `iostream`. Erstere enthält nur die (Forward-)Deklarationen von `std::istream` und `std::ostream`, was die Compilierzeit beim Einbinden ihrer Headerdatei verringert.

Für die konkrete Ein-/Ausgabe nutzen wir nun die Member unserer `MyString`-Klasse. Da die Ausgabe sich etwas einfacher gestaltet, wird sie zuerst erläutert. Die Methode `length()` liefert uns die Anzahl der gültigen Zeichen in unserem String und `operator[]` liefert uns per Index ein Zeichen vom Datentyp `char`. Für diesen ist ein Ausgabeoperator deklariert, also geben wir einfach in einer Zählschleife `length()` viele Zeichen aus. Am Ende geben wir noch unseren `std::ostream` zurück.

```
std::ostream& operator<<(std::ostream& os, MyString const& rhs){
    for(std::size_t i = 0; i < rhs.length(); ++i){
        os << rhs[i];
    }
    return os;
}
```

Für die Eingabe haben wir keine Information wie lang der zu lesende String werden soll. Der `std::string` verhält sich so, dass bis zum nächsten Leerraumzeichen eingelesen wird. Wir werden immer bis zum nächsten Zeilenumbruch lesen, maximal jedoch `MyString::max_length - 1` viele Zeichen. Praktischerweise bietet uns `std::istream` mit der Methode

`getline()` genau das, was wir brauchen. `getline()` bekommt als ersten Parameter einen Zeiger auf `char`, was als Zeiger auf die Anfangsadresse eines Arrays von `chars` zu verstehen ist. Als zweiter Parameter wird die Größe des Arrays übergeben. Die `-1` bei `MyString::max_length - 1` ist leider nötig, weil `getline()` für C-Strings (Nullterminiert) gedacht ist und somit wird das letzte Zeichen im Puffer immer auf `'\0'` setzt. Die Methode `gcount()` liefert uns im Anschluss die Information, wie viele Zeichen nun wirklich gelesen wurden. Danach bleibt nur noch den `std::istream` zurückzugeben.

```
std::istream& operator>>(std::istream& is, MyString& rhs){
    is.getline(rhs.data_, MyString::max_length);
    rhs.length_ = is.gcount();
    return is;
}
```

Wie Ihnen zweifellos auffällt, wird hier auf private Klassenmember zugegriffen. Dies ist für Ein- und Ausgabeoperatoren oft sinnvoll, aber nur möglich wenn sie zuvor auch innerhalb der Klasse als `friend` deklariert wurden. Ergänzen Sie daher die Klassendefinition folgendermaßen:

```
class MyString{
// ...
friend std::istream& operator>>(std::istream& is, MyString& rhs);
};
```

Dass die Zeile nicht eingerückt wurde, soll andeuten, dass es für eine `friend`-Deklaration unerheblich ist, in welchem Sichtbarkeitsbereich sie stattfindet.

Bei den Ein-/Ausgabeoperatoren handelt es sich eigentlich um den Rechtsshift-`operator>>` und den Linksshift-`operator`. Eine solche Entfremdung bezüglich der Funktionalität eines Operators sollte eigentlich nicht stattfinden. Wenn es allerdings konsequent umgesetzt wird, wie es beim Ein-/Ausgabesystem von C++ der Fall ist, kann es nach einer kurzen Einlernphase für den Nutzer (Sie als Programmierer) die Übersichtlichkeit erheblich steigern.

Tip

Ein weiteres gutes Beispiel für eine Bibliothek die Operatoren sinnvollerweise zweckentfremdet ist `Boost::Spirit`. Mit den Boost Bibliotheken sollten Sie sich übrigens früher oder später vertraut machen. `Spirit` ist beispielsweise speziell für die Ein-/Ausgabe extrem nützlich. Sie finden die Boost-Bibliotheken unter boost.org^a.

^a <http://boost.org>

28.1.4 Cast-Operatoren

Wie Ihnen bereits bekannt ist, kann ein Datentyp in einen anderen konvertiert werden, wenn letzter einen Konstruktor bereitstellt der ein Objekt des ersten übernimmt. Oft kommt es jedoch vor, dass Sie auf den Datentyp, in den Umgewandelt werden soll, keinen Zugriff haben, sprich keinen zusätzlichen

Konstruktor einführen können. Bei den elementaren Datentypen ist dies beispielsweise grundsätzlich der Fall.

Aus diesem Grund gibt es die Cast-Operatoren. Wenn Sie nun, etwa mittels `static_cast`, eine Umwandlung vornehmen, wird entweder der passende Konstruktor aufgerufen oder ein passender Cast-Operator, wobei nur eines von beiden Deklariert sein darf, weil sich der Compiler sonst über Mehrdeutigkeit beschwert. Ein Cast-Operator wird immer in der Klasse deklariert, die man umwandeln möchte. Die Syntax lautet wie folgt:

```
operator «Zieldatentyp»();
```

Auffällig ist, dass für diese „Funktionen“ kein Rückgabotyp angegeben wird. Diese wurde (analog zum Konstruktor) weggelassen, da der Rückgabotyp offensichtlich ist. Ein Konstruktor liefert ein Objekt seiner Klasse und die Typumwandlungsoperator gibt natürlich ein Objekt des Zieldatentyps zurück.

Im folgenden möchten wir unserer `MyString`-Klasse noch einen Cast-Operator nach `double` spendieren. Die Deklaration sieht folgendermaßen aus:

```
class MyString{
public:
    // ...
    operator double()const;
// ...
};
```

Wie Sie sehen, wurde der Operator als konstanter Member deklariert. Dies ist für die meisten Typumwandlungsoperatoren sinnvoll, denn Typischerweise wollen wir das Originalobjekt ja nicht verändern. Analog dazu werden einem Konstruktor typischerweise Referenzen auf konstante Objekt übergeben. Die Definition kann beispielsweise folgendermaßen aussehen:

```
#include <sstream>

MyString::operator double()const{
    std::stringstream stream;
    stream << *this; // Text temporär ausgeben
    double result;
    stream >> result; // Als Zahl wieder einlesen
    return result;
}
```

Nun können Sie `MyString`-Objekt implizit und explizit in `doubles` umwandeln.

```
// Klassendefinition
```

```
int main() {
    MyString zahl = "5.5";
    double test = 3.2;
    test += zahl; // implizite Typumwandlung
    std::cout << test << std::endl;
    test = 5 + static_cast< double >(zahl); // explizite Typumwandlung
    std::cout << test << std::endl;
}
```

Ausgabe

```
8.7
10.5
```

In diesem Fall wäre die explizite Typumwandlung nicht zwingend nötig gewesen, der Compiler hätte auch implizit (ohne `static_cast< double >`) das richtige getan, es gibt jedoch reichlich Fälle in denen eine explizite Umwandlung nötig ist.

28.2 Spaß mit „falschen“ Überladungen

Mit der Operatorüberladung lässt sich auch einiger Unsinn anstellen. Die folgende kleine Klasse verhält sich wie ein normaler `int`, allerdings rechnet der Plusoperator minus, und umgekehrt. Gleiches gilt für die Punktoperationen. Natürlich sollten Sie solch unintuitives Verhalten in ernst gemeinten Klassen unbedingt vermeiden, aber an dieser Stelle demonstriert es noch einmal schön die Syntax zur Überladung.

```
// Zum Überladen von Ein-/Ausgabe
#include <iostream>

class Int{
public:
    // Konstruktor (durch Standardparameter gleichzeitig Standardkonstruktor)
    Int(int value = 0): m_value(value) {}

    // Überladene kombinierte Rechen-/Zuweisungsoperatoren
    Int const& operator+=(Int const& rhs) { m_value -= rhs.m_value; return *this; }
    Int const& operator-=(Int const& rhs) { m_value += rhs.m_value; return *this; }
    Int const& operator*=(Int const& rhs) { m_value /= rhs.m_value; return *this; }
    Int const& operator/=(Int const& rhs) { m_value *= rhs.m_value; return *this; }

private:
    int m_value;

    // Friend-Deklarationen für Ein-/Ausgabe
    friend std::istream& operator>>(std::istream& is, Int& rhs);
    friend std::ostream& operator<<(std::ostream& os, Int const& rhs);
};
```

```
// Überladene Rechenoperatoren rufen kombinierte Versionen auf
Int operator+(Int const& lhs, Int const& rhs) { return Int(lhs) += rhs; }
Int operator-(Int const& lhs, Int const& rhs) { return Int(lhs) -= rhs; }
Int operator*(Int const& lhs, Int const& rhs) { return Int(lhs) *= rhs; }
Int operator/(Int const& lhs, Int const& rhs) { return Int(lhs) /= rhs; }

// Definition Ein-/Ausgabeoperator
std::istream& operator>>(std::istream& is, Int& rhs) { return is >> rhs.m_value; }
std::ostream& operator<<(std::ostream& os, Int const& rhs) { return os << rhs.m_value; }
```

28.3 Präfix und Postfix

Für einige Klassen kann es sinnvoll sein, den Increment- und den Decrement-Operator zu überladen. Da es sich um unäre Operatoren handelt, sie also nur einen Parameter haben, werden sie als Klassenmember überladen. Entsprechend übernehmen sie überhaupt keine Parameter mehr, da dieser ja durch das Klassenobjekt, für das sie aufgerufen werden ersetzt wird. Als Beispiel nutzen wir noch einmal unsere Int-Klasse, diesmal aber ohne die Operatoren mit der falschen Funktionalität zu überladen.

```
class Int{
public:
// ...

    Int& operator++() { ++m_value; return *this; }
    Int& operator--() { --m_value; return *this; }

// ...
};

// ...

int main(){
    Int value(5);

    std::cout << value << std::endl;    // value ist 5, Ausgabe ist 5

    std::cout << ++value << std::endl; // value ist 6, Ausgabe ist 6
    std::cout << --value << std::endl; // value ist 5, Ausgabe ist 5
}
```

So weit, so gut, aber wir können nicht `value++` oder `value--` schreiben. Diese Variante überlädt also nur die Präfix-Version der Operatoren, aber wie schreibt man nun die Postfix-Überladung? Wie Sie wissen, kann eine Funktion nur anhand ihrer Parameter oder, im Fall einer Methode, auch ihres

`const`-Qualifiziertes überladen werden. Aber weder die Präfix- noch die Postfixvariante übernehmen einen Parameter und auch der `const`-Qualifizier ist in keinem Fall gegeben, weil die Operatoren ja immer das Objekt verändern.

Aus diesem Grund hat man sich eine Kompromisslösung ausdenken müssen. Die Postfix-Operatoren übernehmen einen `int`, der jedoch innerhalb der Funktionsdefinition nicht verwendet wird. Er dient einzig, um dem Compiler mitzuteilen, dass es sich um eine Postfix-Operatorüberladung handelt.

```
class Int{
public:
// ...

// Präfix (also ++Variable)
Int& operator++() { ++m_value; return *this; }
Int& operator--() { --m_value; return *this; }

// Postfix (also Variable++)
Int const operator++(int) { Int tmp = *this; ++m_value; return tmp; }
Int const operator--(int) { Int tmp = *this; --m_value; return tmp; }

// ...
};

// ...

int main(){
    Int value(5);

    std::cout << value << std::endl;    // value ist 5, Ausgabe ist 5

    std::cout << ++value << std::endl; // value ist 6, Ausgabe ist 6
    std::cout << --value << std::endl; // value ist 5, Ausgabe ist 5

    std::cout << value++ << std::endl; // value ist 6, Ausgabe ist 5
    std::cout << value-- << std::endl; // value ist 5, Ausgabe ist 6

    std::cout << value << std::endl;    // value ist 5, Ausgabe ist 5
}
```

Im folgenden wird auf den Inkrementoperator (`++`) Bezug genommen, für den Dekrementoperator (`--`) gilt natürlich das Gleiche, mit dem Unterschied, dass die Werte erniedrigt werden.

In diesem Beispiel sehen Sie auch schön den Unterschied zwischen Präfix- und Postfix-Variante. Bei der Präfix-Variante wird der Wert um 1 erhöht und anschließend zurückgegeben. Der Postfix-Operator erhöht den Wert zwar um 1, gibt anschließend aber den Wert zurück, den das Objekt vor der Erhöhung hatte. Entsprechend sind auch die Rückgabewerte der Operatoren. Bei Präfix wird eine Referenz auf sich selbst zurückgeliefert, bei Postfix hingegen ein konstantes temporäres Objekt, das den alten Wert des Objektes enthält, für das der Operator aufgerufen wurde. Die Implementierung

ist entsprechend. Die Präfix-Version erhöht den Wert und liefert sich dann selbst zurück, während die Postfix-Variante zunächst eine Kopie der aktuellen Objektes erstellt, dann den Wert erhöht und schließlich den zwischengespeicherten, alten Wert zurückgibt.

Wie früher schon einmal erwähnt, ist es effizienter den Präfix-Operator zu benutzen, wenn prinzipiell beide Varianten möglich wären. Bei den Basisdatentypen, kann der Compiler einen Performance-Nachteil wegoptimieren. Bei Klassen für die diese Operatoren überladen wurden, kann sich der Compiler jedoch nicht darauf verlassen, dass die Präfix-Variante das gleiche macht, wie die Postfix-Variante. Daher ist er bei Klassen nicht oder zumindest nur sehr eingeschränkt in der Lage an dieser Stelle zu optimieren. Aus Gründen der Einheitlichkeit sollte daher, auch bei Basisdatentypen, die Präfix-Variante bevorzugt werden.

Der Rückgabewert der Postfix-Variante ist übrigens aus gutem Grund konstant. Den Präfix-Operator können Sie beliebig oft hintereinander für das gleiche Objekt aufrufen. Auch wenn in den meisten Fällen der Aufruf der Plus-Zuweisung (`+=`) möglich und effizienter ist. Für die Postfix-Variante ist ein mehrfacher Aufruf nicht sinnvoll, wie im folgenden Demonstriert.

```
int i = 5;

+++++i; // i ist 8
// Effizienter wäre "i += 3;"

i++++++; // i ist 9
// Lässt sich nur übersetzen, wenn Rückgabewert nicht const
```

Beim mehrfachen Aufruf der Postfix-Variante werden bis auf den ersten, alle Aufrufe für temporäre Objekte aufgerufen. Hinzu kommt noch, dass der Rückgabewert ja immer der alte Wert ist. Somit ist die Operation für jeden Folgeaufruf die Gleiche, während der berechnete Wert, als temporäres Objekt, sogleich verworfen wird. Ist der Rückgabewert konstant, kann kein Nicht-Konstanter-Operator für das Objekt aufgerufen werden und eine solch unsinnige Postfix-Operatorkette resultiert in einem Compilerfehler.

29 Klassen als Datenelemente einer Klasse

Analog zu Basistyp-Variablen, können Sie auch Klassenobjekte als Member einer Klasse deklarieren. Dies funktioniert genau so, wie Sie es schon kennen.

```
class A{
    int zahl;
};

class B{
    A a_objekt;
};
```

Nun enthält die Klasse B ein Objekt der Klasse A.

29.1 Verweise untereinander

Manchmal ist es nötig, dass zwei Klassen einen Verweis auf die jeweils andere enthalten. In diesem Fall muss die Klassendeklaration von der Klassendefinition getrennt werden. Wie auch bei Funktionen gilt, dass jede Definition immer auch eine Deklaration ist und dass Deklarationen beliebig oft gemacht werden können, während Definition nur einmal erfolgen dürfen.

```
// Deklaration von A
class A;

// Definition von A
class A{
    int zahl;
};
```

Um ein Klassenobjekt zu deklarieren, muss die gesamte Klassendefinition bekannt sein. Wenn Sie hingegen nur einen Zeiger oder eine Referenz auf ein Klassenobjekt deklarieren möchten, genügt es, wenn die Klasse deklariert wurde.

```
class A; // Deklaration von A

class B{
    // B enthält eine Referenz auf ein A-Objekt
```

```
    A& a_objekt;  
};  
  
class A{  
    // A enthält ein B-Objekt  
    B objekt;  
};
```

Falls Ihnen jetzt die Frage im Hinterkopf herumschwirrt, ob es möglich ist, dass beide Klassen ein Objekt der jeweils anderen enthalten, dann denken Sie noch mal darüber nach, was diese Aussage bedeuten würde. Die Antwort auf diese Frage ist selbstverständlich nein.

30 Rechnen mit Brüchen

Das Rechnen mit Ganzzahlen und Gleitkommazahlen haben Sie inzwischen gelernt. In diesem Kapitel kehren wir das Prinzip mal um. Jetzt werden Sie Ihrem Rechner beibringen mit Brüchen zu rechnen! Davon ausgehend, dass Sie wissen was Brüche¹ sind, sollte das kein allzu großes Problem darstellen. Die Klasse, die am Ende dieses Abschnittes entstanden ist, wird noch nicht perfekt sein, aber sie wird zum normalen Rechnen ausreichen. Sie dürfen sie natürlich jederzeit weiterentwickeln und verbessern.

30.1 Was diese Klasse bieten soll

Bevor wir nun Hals über Kopf anfangen, Quellcode zu schreiben, sollten wir uns erst klar machen, was genau wir überhaupt erreichen wollen.

Brüche bestehen aus einem Zähler und einem Nenner. Diese sind Ganzzahlen, allerdings ist der Nenner eine natürliche Zahl (ohne Vorzeichen) und der Zähler eine ganze Zahl (mit Vorzeichen). Daher werden wir den entsprechenden Variablen passende Datentypen zuordnen.

Da es beim Rechnen leichter ist, die Brüche gleich in ihrer kleinsten (nicht mehr zu kürzenden) Form zu sehen, werden wir den Bruch nach jeder Rechenoperation kürzen. Weiterhin wollen wir mit den Brüchen rechnen können wie mit den eingebauten Datentypen für Zahlen. Natürlich brauchen wir eine Ein- und Ausgabe für unsere Brüche. Schließlich und endlich wollen wir sie beim Rechnen mit den eingebauten Datentypen mischen.

30.2 Was diese Klasse nicht kann

Die logische Folge des automatischen Kürzens ist natürlich, dass sich die Brüche nicht mit beliebigen Zahlen erweitern und kürzen lassen. Es ist natürlich möglich, beides gleichzeitig zu realisieren, aber es ist nicht sinnvoll, also machen wir das auch nicht. Eine Umwandlung von Gleitkommazahlen in gebrochene Zahlen ist ebenfalls nicht vorgesehen.

30.3 Ein erster Blick

Gleich werden Sie zum ersten Mal einen Blick auf die Bruch-Klasse werfen, also machen Sie sich bereit:

¹ <http://de.wikipedia.org/wiki/Bruchrechnung>

```
class Bruch{
public:
    Bruch(int zaehler = 0, unsigned int nenner = 1);

private:
    int          zaehler_;
    unsigned int nenner_;
};
```

Das war nicht besonders aufregend, die Klasse hat bisher nur einen Konstruktor und die Variablen `zaehler_` und `nenner_`. Beachten Sie aber, dass `zaehler_` vom Datentyp `int` (vorzeichenbehaftet) ist, während `nenner_` den Datentyp `unsigned int` (vorzeichenlos) hat.

Auf die Standardparameter des Konstruktors werden wir in einem späteren Kapitel noch einmal zu sprechen kommen. Für den Moment soll es reichen, seine Deklaration zu zeigen.

31 Die Methoden

In diesem Kapitel werden Sie mit den Methoden bekannt gemacht, die unsere Bruch-Klasse zur Verfügung stellt. Grundlegend tauchen im Zusammenhang mit Brüchen zwei Begriffe auf: „Erweitern“ und „Kürzen“. Im Zusammenhang mit diesen beiden finden sich wiederum die Begriffe „kleinstes gemeinsames Vielfaches“ und „größter gemeinsamer Teiler“. Wir haben uns bei den Vorüberlegungen dafür entschieden, das Erweitern wegzulassen, bei der Addition und Subtraktion brauchen wir es aber. Im folgenden werden das „kleinste gemeinsame Vielfache“ und der „größte gemeinsame Teiler“ mit kgV und ggT abgekürzt.

Wir haben also die folgenden Methoden:

- ggT
- kgV
- kuerzen

Dazu kommen noch zwei Zugriffsfunktionen, da sich der Benutzer unserer Klasse vielleicht für Zähler und Nenner des Bruchs interessiert:

- zaehler
- nenner

Zwei dieser fünf Methoden müssen aber gleich wieder gestrichen werden. ggT und kgV sind zwar in Zusammenhang mit Brüchen hilfreich, aber sie beschreiben allgemeine mathematische Funktionen, die nicht ausschließlich in Zusammenhang mit Brüchen eingesetzt werden. Daher sind diese beiden *Funktionen* unter der Überschrift Methoden etwas fehlpositioniert, da sie keine Member der Klasse Bruch sind.

31.1 Zugriff

Die beiden Zugriffsmethoden tun nichts weiter, als Zähler und Nenner des Bruchs zurückzugeben, daher schreiben wir sie direkt in die Klassendeklaration.

```
class Bruch{
public:
    Bruch(int zaehler = 0, unsigned int nenner = 1);

    int          zaehler()const {return zaehler_;}
    unsigned int nenner()const  {return nenner_;}

private:
    int          zaehler_;
```

```
    unsigned int nenner_;\n};
```

Wie Sie sehen, haben die beiden Methoden den gleichen Rückgabetyt wie die Variablen, die sie repräsentieren. Achten Sie bitte auch darauf, dass beide als `const` deklariert sind, da sie keine Variable innerhalb der Klasse verändern.

31.2 ggT()

In der Schule haben Sie sicher gelernt, dass sich der ggT (größte gemeinsame Teiler) durch Primfaktorzerlegung ermitteln lässt. Dieses Verfahren ist allerdings denkbar ungünstig um es auf einem Rechner umzusetzen. Sie müssten zunächst einmal die Primzahlen berechnen und das dauert, zumal Sie ja gar nicht wissen, wie viele Primzahlen Sie überhaupt benötigen.

Deshalb bedienen wir uns eines anderen Verfahrens, des euklidischen Algorithmus. Eine Verbesserung dieses Verfahrens ist der steinsche Algorithmus, aber für unsere Zwecke reicht ersterer. Sie dürfen die Klasse natürlich gerne dahingehend verbessern, dass Sie den steinschen Algorithmus einsetzen.

Der euklidische Algorithmus beruht auf zwei Eigenschaften des größten gemeinsamen Teilers:

1. $\text{ggT}(q \cdot b + r, b) = \text{ggT}(b, r)$
2. $\text{ggT}(a, 0) = a$

Wenn Sie eine genaue Beschreibung wünschen, dann schauen Sie sich doch mal den entsprechenden Wikipediaartikel¹ an.

```
unsigned int ggT(unsigned int a, unsigned int b){\n    if(b == 0)\n        return a;\n    else return ggT(b, a % b);\n}
```

Beachten Sie, dass diese Funktion rekursiv funktioniert. Die Implementierung als einfache Schleife wäre ebenfalls möglich gewesen, aber sie ist etwas unübersichtlicher und dank Optimierung ist die rekursive Variante nur unwesentlich langsamer. Wenn Sie nicht mehr wissen, was rekursive Funktionen sind, dann werfen Sie noch mal einen Blick auf das Kapitel Rekursion² im Abschnitt „Weitere Grundelemente“.

¹ <http://de.wikipedia.org/wiki/Euklidischer%20Algorithmus>

² Kapitel 18 auf Seite 107

31.3 kgV()

Das kgV lässt sich ganz einfach berechnen, wenn Sie den ggT kennen. Daher schreiben wir die Funktion mit Hilfe der eben erstellten ggT-Funktion.

```
unsigned int kgV(unsigned int a, unsigned int b){
    return (a * b) / ggT(a, b);
}
```

Hinweis

Damit `kgV()` auf `ggT()` zugreifen kann, muss dieses natürlich bekannt sein. Sie müssen die Deklaration immer geschrieben haben, bevor Sie das entsprechende Element verwenden.

31.4 kuerzen()

`kuerzen()` ist nun endlich mal wirklich eine Memberfunktion (oder auch Methode) von `Bruch`. Da sie somit direkten Zugriff auf die Variablen hat, die sie verändern soll, braucht sie keine Argumente und auch keinen Rückgabewert.

```
class Bruch{
public:
    Bruch(int zaehler = 0, unsigned int nenner = 1);

    int          zaehler()const {return zaehler_;}
    unsigned int nenner()const  {return nenner_;}

private:
    void kuerzen();

    int          zaehler_;
    unsigned int nenner_;
};

void Bruch::kuerzen(){
    unsigned int tmp(ggT(zaehler_, nenner_));
    zaehler_ /= tmp;
    nenner_  /= tmp;
}
```

Da `kuerzen()` eine Methode ist, die nur innerhalb der Klasse nach einer Rechenoperation aufgerufen wird, deklarieren wir sie im privaten Bereich der Klasse.

32 Die Rechenoperationen

Addition, Subtraktion, Multiplikation und Division, das sind 4 Rechenoperationen. C++ bietet aber die mit der Zuweisung kombinierten Kurzschreibweisen, womit wir insgesamt auf 8 kommen.

32.1 Addition

Um zwei Brüche zu addieren, müssen die Nenner gleich sein. Wenn wir bereits Brüche haben, die sich nicht weiter kürzen lassen, (und dafür sorgt unsere Klasse,) dann erhalten wir wiederum einen unkürzbaren Bruch, wenn wir mit dem kgV erweitern. Das Ganze sieht also so aus:

```
ErgebnisNenner = kgV(Bruch1Nenner, Bruch2Nenner);
ErgebnisZaehler = Bruch1Zaehler * (ErgebnisNenner / Bruch1Nenner) +
    Bruch2Zaehler * (ErgebnisNenner / Bruch2Nenner);
```

32.2 Subtraktion

Für die Subtraktion gelten die gleichen Regeln wie bei der Addition.

```
ErgebnisNenner = kgV(Bruch1Nenner, Bruch2Nenner);
ErgebnisZaehler = Bruch1Zaehler * (ErgebnisNenner / Bruch1Nenner) -
    Bruch2Zaehler * (ErgebnisNenner / Bruch2Nenner);
```

32.3 Multiplikation

Bei der Multiplikation werden einfach die Zähler und die Nenner multipliziert. Danach muss der Bruch wieder gekürzt werden.

```
ErgebnisZaehler = Bruch1Zaehler * Bruch2Zaehler;
ErgebnisNenner = Bruch1Nenner * Bruch2Nenner;
kuerzen();
```

32.4 Division

Die Division stimmt mit der Multiplikation fast überein, aber statt die Zähler und die Nenner miteinander zu multiplizieren, werden sie gegeneinander multipliziert.

```
ErgebnisZaehler = Bruch1Zaehler * Bruch2Nenner;  
ErgebnisNenner = Bruch1Nenner * Bruch2Zaehler;  
kuerzen();
```

32.5 Kombination

Da C++ wie schon gesagt neben den normalen Rechenoperatoren noch die mit der Zuweisung kombinierten zur Verfügung stellt, werden wir einen kleinen Trick anwenden, um uns doppelte Arbeit zu ersparen. Wir werden die eigentlichen Rechenoperationen in den Zuweisungskombioperatoren implementieren und dann innerhalb der normalen Rechenoperatoren temporäre Objekte anlegen, für welche wir die Kombinationsoperatoren aufrufen. Das ist ein übliches und viel angewandtes Verfahren, welches einige Vorteile zu bieten hat. Sie sparen sich die doppelte Schreibarbeit und müssen sich bei Veränderungen nur um die Kombioperatoren kümmern, da sich die anderen ja genauso verhalten.

Die umgekehrte Variante, also von den Kombioperatoren die normalen aufrufen zu lassen, ist übrigens nicht zu empfehlen, da die Kombinationsoperatoren immer schneller sind, sie benötigen schließlich keine temporären Objekte. Außerdem ist es in vielen Klassen nötig, die normalen Rechenoperatoren außerhalb der Klasse zu deklarieren. Wenn sie nicht als `friend` deklariert sind, haben sie keinen Zugriff auf die privaten Member der Klasse, rufen Sie dagegen die Kombioperatoren auf, brauchen Sie gar keinen Zugriff.

Die normalen Rechenoperatoren werden als Nicht-Member implementiert. Dies ist nötig, damit für beide Parameter eine implizite Typumwandlung erfolgen kann. Andernfalls könnten Sie zwar `Bruch(1, 2)+3` schreiben, aber nicht `3+Bruch(1, 2)` und das ist natürlich nicht gewollt. Die Deklaration als Nicht-Member hat zur Folge, dass wir zwei Parameter übergeben müssen. Der erste wird als Kopie („Call by value“) übergeben, da wir ja sowieso eine temporäre Kopie benötigen. Den Zweiten übergeben wir wie üblich als Referenz auf `const`. Da die Implementierung so kurz ist, sind diese Funktionen natürlich `inline`. Der Rückgabebetyp lautet immer `Bruch const`. Dies verhindert, dass Sie so etwas wie `a + b = 4` schreiben können, obwohl doch `a + b == 4` gemeint war.

32.6 Abschluss

So, nun haben Sie wirklich genug Theorie gehört, es wird Zeit zu zeigen wie das ganze im Quelltext aussieht. Der Code lässt sich zwar noch nicht ausführen, weil der Konstruktor noch nicht definiert ist, aber es lohnt sich trotzdem schon mal einen Blick darauf zu werfen.

```

unsigned int ggT(unsigned int a, unsigned int b){
    if(b == 0)                // Wenn b gleich 0
        return a;            // ggT gefunden
    else return ggT(b, a % b); // andernfalls weitersuchen
}

unsigned int kgV(unsigned int a, unsigned int b){
    // Das kgV zweier Zahlen, ist ihr Produkt geteilt durch ihren ggT
    return a * b / ggT(a, b);
}

class Bruch{
public:
    Bruch(int zaehler = 0, unsigned int nenner = 1); // noch nicht definiert

    int        zaehler()const { return zaehler_; } // Gibt Zähler zurück
    unsigned int nenner()const { return nenner_; }  // Gibt Nenner zurück

    Bruch& operator+=(Bruch const& lvalue);
    Bruch& operator-=(Bruch const& lvalue);
    Bruch& operator*=(Bruch const& lvalue);
    Bruch& operator/=(Bruch const& lvalue);

private:
    void kuerzen(); // kürzt weitestmöglich

    int        zaehler_;
    unsigned int nenner_;
};

// Diese Methoden erstellen eine temporäre Kopie (lhs) ihres Objekts, führen
// die Rechenoperation auf ihr aus und geben sie dann zurück
inline Bruch const operator+(Bruch lhs, Bruch const &rhs){ return lhs += rhs; }
inline Bruch const operator-(Bruch lhs, Bruch const &rhs){ return lhs -= rhs; }
inline Bruch const operator*(Bruch lhs, Bruch const &rhs){ return lhs *= rhs; }
inline Bruch const operator/(Bruch lhs, Bruch const &rhs){ return lhs /= rhs; }

void Bruch::kuerzen(){
    const unsigned int tmp = ggT(zaehler_, nenner_); // ggT in tmp speichern
    zaehler_ /= tmp; // Zähler durch ggT teilen
    nenner_ /= tmp; // Nenner durch ggT teilen
}

Bruch& Bruch::operator+=(Bruch const &lvalue){
    const unsigned int tmp = kgV(nenner_, lvalue.nenner_);
    zaehler_ = zaehler_ * (tmp / nenner_) + lvalue.zaehler_ * (tmp / lvalue.nenner_);
}

```

```
        nenner_ = tmp;
        return *this; // Referenz auf sich selbst zurückgeben
    }

Bruch& Bruch::operator--(Bruch const &lvalue){
    const unsigned int tmp = kgV(nenner_, lvalue.nenner_);
    zaehler_ = zaehler_ * (tmp / nenner_) - lvalue.zaehler_ * (tmp / lvalue.nenner_);
    nenner_ = tmp;
    return *this; // Referenz auf sich selbst zurückgeben
}

Bruch& Bruch::operator*=(Bruch const &lvalue){
    zaehler_ *= lvalue.zaehler_;
    nenner_ *= lvalue.nenner_;
    kuerzen(); // Bruch wieder kürzen
    return *this; // Referenz auf sich selbst zurückgeben
}

Bruch& Bruch::operator/=(Bruch const &lvalue){
    zaehler_ *= lvalue.nenner_;
    nenner_ *= lvalue.zaehler_;
    kuerzen(); // Bruch wieder kürzen
    return *this; // Referenz auf sich selbst zurückgeben
}
```

33 Umwandlung aus anderen Datentypen

Um aus einer Variable eines anderen Datentyps in einen Bruch umzuwandeln, übergibt man einem Konstruktor diese Variable und lässt ihn die Umwandlung durchführen. Sie erinnern sich bestimmt noch daran, dass im ersten Kapitel dieses Abschnittes stand, die Standardparameter des Konstruktors würden später besprochen. Dieser Zeitpunkt ist nun gekommen. Zur Erinnerung, seine Deklaration innerhalb der Klasse lautete:

```
class Bruch{
public:
    Bruch(int zaehler = 0, unsigned int nenner = 1);

    // ...
};

Bruch::Bruch(int zaehler, unsigned int nenner):
    zaehler_(zaehler),
    nenner_(nenner){
    kuerzen();
}
```

Die Definition steht, wie Sie oben sehen, außerhalb der Klasse. Innerhalb der Initialisierungsliste, welche durch einen Doppelpunkt eingeleitet wird, werden die Membervariablen mit den übergebenen Parametern initialisiert. Innerhalb des Funktionsrumpfes wird die Methode `kuerzen()` aufgerufen, um den Bruch falls nötig zu kürzen.

Beachten Sie, dass die Standardparameter nur bei der Deklaration, nicht aber bei der Definition einer Funktion angegeben werden. Unser Konstruktor übernimmt 2 `int`-Werte und beide besitzen einen Standardwert, daher kann er in 3 Formen aufgerufen werden:

```
Bruch bruch1() // erzeugt (0/1)
Bruch bruch2(5) // erzeugt (5/1)
Bruch bruch3(3, 4) // erzeugt (3/4)
```

Die erste Form entspricht einem Defaultkonstruktor und setzt den Wert des Bruches auf $(0/1)$, was dem ganzzahligen Wert 0 entspricht. In der zweiten Form wird 1 `int`-Wert übergeben, der resultierende Bruch entspricht diesem Wert, da der Nenner dank des Standardparameters auf 1 gesetzt wird. Die dritte Form übernimmt schließlich 2 `int`-Werte, der erste gibt den Zähler und der zweite den Nenner des Bruches an.

Auch an dieser Stelle soll noch einmal darauf hingewiesen werden, dass der Nenner nicht negativ sein kann, daher ist dieser Parameter auch vom Typ `unsigned int` welcher nur positive Werte zulässt.

Thema wird später näher erläutert...

Bitte beachten Sie, dass der Nenner eines Bruches normalerweise nicht 0 sein kann. Normalerweise würde man dem in C++ mit einer Ausnahme begegnen. Da wir Ausnahmen aber bisher nicht besprochen haben und es auch noch eine Weile dauern wird, bis wir dieses Thema behandeln, werden wir dem Umstand, dass es möglich ist dem Konstruktor von `Bruch` eine 0 als Nenner zu übergeben, erst einmal ignorieren. Später kommen wir auf diesen Punkt aber noch einmal zurück.

33.1 Gleitkommazahl wird Bruch

Wir haben jetzt die Umwandlung von ganzen Zahlen in Brüche besprochen. Als nächstes wollen wir eine Gleitkommazahl in einen Bruch umwandeln. Hierfür benötigen wir einen zweiten Konstruktor, welcher eine Gleitkommazahl übernimmt und sie in einen Bruch umwandelt:

```
class Bruch{
public:
    Bruch(int zaehler = 0, unsigned int nenner = 1);
    Bruch(double wert);

// ...
};

Bruch::Bruch(double wert):
    zaehler_(static_cast<int>(wert*1000000.0+0.5)),
    nenner_(1000000){
    kuerzen();
}
```

Es ist kaum möglich, einen `double`-Wert zuverlässig in einen `Bruch` umzuwandeln, da die Datentypen, die wir für Zähler (`int`) und Nenner (`unsigned int`) verwenden, den Wertebereich unserer Brüche gewaltig einschränken. Dieser Konstruktor ist daher kaum praxistauglich, aber als Beispiel sollte er genügen. Einmal vorausgesetzt, dass ein `int` 4 Byte groß ist, beachtet er drei Vorkomma- und sechs Nachkommastellen.

Wir setzen den Nenner des Bruches auf 1000000, dann multiplizieren wir den übergebenen Wert mit 1 000 000. Würde man jetzt Zähler durch Nenner teilen, hätte man wieder exakt den Wert, der übergeben wurde. Da der Zähler aber ein `int`- und kein `double`-Wert sein muss, müssen wir ihn noch umwandeln. Da eine solche Umwandlung aber alle Nachkommastellen abschneidet, anstatt kaufmännisch korrekt zu runden, addieren wir den `double`-Wert vorher mit 0.5, was dazu führt, dass die Zahl nach dem Abschneiden der Nachkommastellen kaufmännisch gerundet wurde.

Alles was über 6 Nachkommastelle hinausgeht ist also für uns nicht relevant, da es korrekt gerundet wird. Das Problem besteht darin, dass der Bruch keinen Zähler aufnehmen kann, der größer als der größtmögliche `int`-Wert (bei 4 Byte 2147483647) ist. Leider lässt sich dieses Problem nicht ohne weiteres lösen, daher werden wir damit leben, das dies nur ein Beispiel für die Umwandlung einer Gleitkommazahl in einen Bruch ist und keine perfekte Implementierung.

Anschließend wird im Funktionsrumpf wieder einmal die Funktion `kuerzen()` aufgerufen. Sie können nun also auch schreiben:

```
Bruch bruch1(0.25)          // erzeugt (1/4)
Bruch bruch2(7.0)           // erzeugt (7/1)
Bruch bruch3(999.999999)    // erzeugt (999999999/1000000)
```

33.2 (K)ein Kopierkonstruktor

Für unsere `Bruch`-Klasse werden wir keinen Kopierkonstruktor anlegen, da unser Compiler uns diese Arbeit mit einem zufriedenstellenden Ergebnis abnimmt. Oder anders ausgedrückt, Sie können, ohne eine einzige Zeile Code zur Klasse hinzuzufügen, schreiben:

```
Bruch bruch1(1, 5); // erzeugt (1/5)
Bruch bruch2(bruch1); // erzeugt (1/5)
```


34 Ein- und Ausgabe

Als nächstes wollen wir dafür sorgen, dass unsere Brüche ebenso einfach ein- und ausgegeben werden können, wie die elementaren Datentypen. Sie wissen ja bereits, dass Sie hierfür nur den Ein- und Ausgabeoperator überladen müssen. Wir wollen die Brüche in der folgenden Form schreiben und lesen:

```
(Zähler/Nenner)
```

Die beiden folgenden Operatorfunktionen können Sie nach der Deklaration der `Bruch`-Klasse einfügen. Include-Dateien gehören natürlich an die übliche Stelle am Dateianfang.

34.1 Ausgabe

Die Ausgabe lässt sich ziemlich simpel realisieren, wir geben einfach das von uns gewünschte Format auf dem Ausgabestream aus, den wir mittels Parameter erhalten. Dann geben wir den Stream zurück. Beachten Sie allerdings, dass Sie die Headerdatei „`iostream`“ inkludieren müssen, damit Sie die Standardstreams überladen können.

```
#include <iostream>

std::ostream& operator<<(std::ostream &os, Bruch const &bruch){
    return os << '(' << bruch.zaehler() << '/' << bruch.nenner() << ')';
}
```

34.2 Eingabe

Die Eingabe ist etwas schwieriger. Wir müssen sicherstellen, dass das von uns vorgegebene Format eingehalten wird und falls nötig, den Eingabestream auf einen Fehlerstatus zu setzen.

```
#include <iostream>

std::istream& operator>>(std::istream &is, Bruch &bruch){
    char tmp;
    int zaehler;
    unsigned int nenner;
```

```
is >> tmp;
if(tmp == '('){
    is >> zaehler;
    is >> tmp;
    if(tmp == '/'){
        is >> nenner;
        is >> tmp;
        if(tmp == ')'){
            bruch = Bruch(zaehler, nenner);
            return is;
        }
    }
}
is.setstate(std::ios_base::failbit);
return is;
}
```

Wie Sie sehen können, wird diesmal eine nicht-konstante Referenz auf einen Bruch übergeben, da dieser ja geändert wird, wenn man ihm einen neuen Wert zuweist. Da die Klasse `Bruch` keine Möglichkeit bereitstellt, `zaehler` oder `nenner` einzeln zu ändern, erstellen wir nach dem Einlesen beider einen entsprechenden `Bruch` und weisen ihn an unseren zu. Auf diese Weise stellen wir auch sicher, dass sich an dem Bruch nichts verändert, wenn während des Einlesens irgendetwas schief geht. Falls etwas daneben geht, wird das `failbit` gesetzt und anschließend der Stream zurückgegeben.

35 Umwandlung in andere Datentypen

In diesem Kapitel werden wir unseren Bruch in Gleitkommazahlen umwandeln. Hierfür müssen wir einfach den Zähler durch den Nenner teilen. Da jedoch beide einen integralen Typ haben, müssen wir vor dem Teilen einen der Werte in eine Gleitkommazahl umwandeln, damit keine Ganzzahldivision durchgeführt wird. Bei Ganzzahldivisionen würden natürlich die Nachkommastellen abgeschnitten.

```
class Bruch{
public:
// ...
    operator float()      {return static_cast<float>(zaehler_) / nenner_;}
    operator double()     {return static_cast<double>(zaehler_) / nenner_;}
    operator long double(){return static_cast<long double>(zaehler_) / nenner_;}
// ...
};
```

Sie können den Bruch jetzt auf die folgende Weise in eine Gleitkommazahl umwandeln:

```
#include <iostream>

// Alles was zur Bruch-Klasse gehört

int main(){
    std::cout << static_cast<double>(Bruch(1, 8));
}
Ausgabe
0.125
```


36 Der Taschenrechner geht zur Schule

Im Abschnitt „Einführung in C++“ gab es ein Kapitel über einen einfachen Taschenrechner¹. In diesem Kapitel werden wir ihn ganz leicht modifizieren, so dass er mit Brüchen statt mit Gleitkommazahlen rechnet und die Ausgabe als Brüche und als Gleitkommazahlen gemacht wird.

```
#include <iostream>

// Alles was zur Bruch-Klasse gehört

int main(){
    Bruch zahl1, zahl2, ergebnis;           // Variablen für Zahlen vom Typ Bruch
    char rechenzeichen;                    // Variable fürs Rechenzeichen

    std::cout << "Geben Sie eine Rechenaufgabe ein: "; // Eingabeaufforderung ausgeben
    std::cin >> zahl1 >> rechenzeichen >> zahl2;     // Aufgabe einlesen

    switch(rechenzeichen){                 // Wert von rechenzeichen ermitteln
        case '+': ergebnis = zahl1+zahl2; break;    // entsprechend dem
        case '-': ergebnis = zahl1-zahl2; break;    // Rechenzeichen
        case '*': ergebnis = zahl1*zahl2; break;    // das Ergebnis
        case '/': ergebnis = zahl1/zahl2; break;    // berechnen
        // Fehlerausgabe und Programm beenden, falls falsches Rechenzeichen
        default: std::cout << "unbekanntes Rechenzeichen...\n"; return 1;
    }

    // Aufgabe noch mal komplett ausgeben
    std::cout << zahl1 << ' ' << rechenzeichen << ' ' << zahl2 << " = " << ergebnis << '\n';
    std::cout << static_cast<double>(zahl1) << ' ' // Ausgabe als
        << rechenzeichen << ' ' // Gleitkommawerte
        << static_cast<double>(zahl2) << " = "
        << static_cast<double>(ergebnis) << '\n';
}
```

Ausgabe

```
Geben Sie eine Rechenaufgabe ein: <Eingabe>(1/4)*(1/2)</Eingabe>
(1/4) * (1/2) = (1/8)
0.25 * 0.5 = 0.125
```

In der Zusammenfassung finden Sie noch einmal das gesamte Programm.

¹ Kapitel 14 auf Seite 79

37 Zusammenfassung

Hier finden Sie noch einmal das komplette Programm, inklusive der Ausgabe eines Durchlaufs.

```
#include <iostream>

unsigned int ggT(unsigned int a, unsigned int b){
    if(b == 0)
        return a;
    else return ggT(b, a % b);
}

unsigned int kgV(unsigned int a, unsigned int b){
    return a/ggT(a,b) * b;
}

class Bruch{
public:
    Bruch(int zaehler = 0, unsigned int nenner = 1); // Konstruktoren
    Bruch(double wert); // dieser ist nicht perfekt

    int zaehler()const {return zaehler_;} // Gibt Zähler zurück
    unsigned int nenner()const {return nenner_;} // Gibt Nenner zurück

    Bruch& operator+=(Bruch const &lvalue);
    Bruch& operator-=(Bruch const &lvalue);
    Bruch& operator*=(Bruch const &lvalue);
    Bruch& operator/=(Bruch const &lvalue);

    // Umwandlung in Gleitkommatypen
    operator float() {return static_cast<float>(zaehler_)/nenner_;}
    operator double() {return static_cast<double>(zaehler_)/nenner_;}
    operator long double(){return static_cast<long double>(zaehler_)/nenner_;}

private:
    void kuerzen(); // kürzt weitestmöglich

    int zaehler_;
    unsigned int nenner_;
};
```

```
// Diese Methoden erstellen eine temporäre Kopie (lhs) ihres Objekts, führen
// die Rechenoperation auf ihr aus und geben sie dann zurück
inline Bruch const operator+(Bruch lhs, Bruch const &rhs){ return lhs += rhs; }
inline Bruch const operator-(Bruch lhs, Bruch const &rhs){ return lhs -= rhs; }
inline Bruch const operator*(Bruch lhs, Bruch const &rhs){ return lhs *= rhs; }
inline Bruch const operator/(Bruch lhs, Bruch const &rhs){ return lhs /= rhs; }

Bruch::Bruch(int zaehler, unsigned int nenner):
    zaehler_(zaehler),
    nenner_(nenner){
    kuerzen();
}

Bruch::Bruch(double wert):
    zaehler_(static_cast<int>(wert*1000000.0+0.5)),
    nenner_(1000000){
    kuerzen();
}

void Bruch::kuerzen(){
    unsigned int tmp = ggT(zaehler_, nenner_); // ggT in tmp speichern
    zaehler_ /= tmp;                          // Zähler durch ggT teilen
    nenner_ /= tmp;                            // Nenner durch ggT teilen
}

Bruch& Bruch::operator+=(Bruch const &lvalue){
    unsigned int tmp = kgV(nenner_, lvalue.nenner_);
    zaehler_ = zaehler_ * (tmp / nenner_) + lvalue.zaehler_ * (tmp / lvalue.nenner_);
    nenner_ = tmp;
    return *this; // Referenz auf sich selbst zurückgeben
}

Bruch& Bruch::operator-=(Bruch const &lvalue){
    unsigned int tmp = kgV(nenner_, lvalue.nenner_);
    zaehler_ = zaehler_ * (tmp / nenner_) - lvalue.zaehler_ * (tmp / lvalue.nenner_);
    nenner_ = tmp;
    return *this; // Referenz auf sich selbst zurückgeben
}

Bruch& Bruch::operator*=(Bruch const &lvalue){
    zaehler_ *= lvalue.zaehler_;
    nenner_ *= lvalue.nenner_;
    kuerzen(); // Bruch wieder kürzen
    return *this; // Referenz auf sich selbst zurückgeben
}
```

```
Bruch& Bruch::operator/=(Bruch const &lvalue){
    zaehler_ *= lvalue.nenner_;
    nenner_  *= lvalue.zaehler_;
    kuerzen(); // Bruch wieder kürzen
    return *this; // Referenz auf sich selbst zurückgeben
}

std::ostream& operator<<(std::ostream &os, Bruch const &bruch){
    return os << '(' << bruch.zaehler() << '/' << bruch.nenner() << ')';
}

std::istream& operator>>(std::istream &is, Bruch &bruch){
    char tmp;
    int zaehler, nenner;
    is >> tmp;
    if(tmp=='('){
        is >> zaehler;
        is >> tmp;
        if(tmp=='/'){
            is >> nenner;
            is >> tmp;
            if(tmp==' '){
                bruch=Bruch(zaehler, nenner); // Bruch erzeugen und Wert übernehmen
                return is;
            }
        }
    }
    is.setstate(std::ios_base::failbit); // Fehlerstatus setzen
    return is;
}

int main(){
    Bruch zahl1, zahl2, ergebnis; // Variablen für Zahlen vom Typ Bruch
    char rechenzeichen; // Variable fürs Rechenzeichen

    std::cout << "Geben Sie eine Rechenaufgabe ein: "; // Eingabeaufforderung ausgeben
    std::cin >> zahl1 >> rechenzeichen >> zahl2; // Aufgabe einlesen

    switch(rechenzeichen){ // Wert von rechenzeichen ermitteln
        case '+': ergebnis = zahl1+zahl2; break; // entsprechend dem
        case '-': ergebnis = zahl1-zahl2; break; // Rechenzeichen
        case '*': ergebnis = zahl1*zahl2; break; // das Ergebnis
        case '/': ergebnis = zahl1/zahl2; break; // berechnen
        // Fehlerausgabe und Programm beenden, falls falsches Rechenzeichen
        default: std::cout << "unbekanntes Rechenzeichen...\n"; return 1;
    }

    // Aufgabe noch mal komplett ausgeben
```

```
std::cout << zahl1 << ' ' << rechenzeichen << ' ' << zahl2 << " = " << ergebnis << '\n';
std::cout << static_cast<double>(zahl1) << ' ' // Ausgabe als
        << rechenzeichen << ' ' // Gleitkommawerte
        << static_cast<double>(zahl2) << " = "
        << static_cast<double>(ergebnis) << '\n';
}
```

Ausgabe

Geben Sie eine Rechenaufgabe ein: <Eingabe>(1/5)-(3/4)</Eingabe>

(1/5) - (3/4) = (-11/20)

0.2 - 0.75 = -0.55

38 Methoden

Auch für Klassen gilt üblicherweise: Verwenden Sie `const`, wann immer es möglich ist. Wie Sie bereits wissen, sollte `const` immer verwendet werden, wenn eine Variable nach der Initialisierung nicht mehr verändert werden soll.

Da Klassen Datentypen sind, von denen Instanzen (also Variablen) erstellt werden können, ist es natürlich möglich ein Objekt zu erstellen, das konstant ist. Da der Compiler jedoch davon ausgehen muss, dass jede Methode der Klasse die Daten (und somit das Objekt) verändert, sind Methodenaufrufe für konstante Objekte nicht möglich. Eine Ausnahme bilden jene Methoden, die ebenfalls als `const` gekennzeichnet sind. Eine solche Methode kann zwar problemlos Daten aus dem Objekt lesen, aber niemals darauf schreiben und auch für Dritte keine Möglichkeit bereitstellen, objektinterne Daten zu verändern.

38.1 Konstante Methoden

Unsere Beispielklasse `Auto` enthält die konstante Methode `info()`, sie greift nur lesend auf die Membervariablen zu, um ihre Werte auszugeben. Wenn Sie ein konstantes `Auto` Objekt erstellen, können Sie `info()` problemlos aufrufen. Versuchen Sie jedoch `fahren()` oder `tanken()` aufzurufen, wird Ihr Compiler dies mit einer Fehlermeldung quittieren.

```
class Auto{
// ...
    void info()const;

    bool fahren(int km);
    void tanken(float liter);
// ...
};
```

Wie Sie an diesem Beispiel sehen, lässt sich eine Methode als konstant auszeichnen, indem nach der Parameterliste das Schlüsselwort `const` angegeben wird. Diese Auszeichnung folgt also auch der einfachen Regel: `const` steht immer rechts von dem, was konstant sein soll, in diesem Fall die Methode. Da `const` zum Methodenprototyp zählt, muss es natürlich auch bei der Definition der Methode angegeben werden.

Es sei noch einmal explizit darauf hingewiesen, dass sich die Konstantheit einer Methode lediglich auf die Membervariablen der zugehörigen Klasse auswirkt. Es ist problemlos möglich, eine als Parameter übergebene Variable zu modifizieren.

Eine konstante Methode kann ausschließlich andere konstante Methoden des eigenen Objektes aufrufen, denn der Aufruf einer nicht-konstanten Methode könnte ja Daten innerhalb des Objektes ändern.

38.2 Sinn und Zweck konstanter Objekte

Vielleicht haben Sie sich bereits gefragt, wofür es gut sein soll, ein Objekt als konstant auszuzeichnen, wenn der Zweck eines Objektes doch darin besteht, mit den enthaltenen Daten zu arbeiten. Beim Erstellen eines konstanten Objektes können Sie es einmalig über den Konstruktor mit Werten belegen. In diesem Fall haben Sie von einem konstanten Objekt das gleiche, wie bei konstanten Variablen von Basisdatentypen.

Oft ist es jedoch nicht möglich, alle Einstellungen zu einem Objekt über den Konstruktoraufruf festzulegen. Es fördert die Übersichtlichkeit schließlich nicht, wenn man etwa 20 verschiedene Konstruktoren mit je etwa 50 Parametern hat. Der Ansatz, Klassen so zu gestalten, dass man immer alle Werte über den Konstruktor festlegen kann, hat also seine Grenzen. In diesem Fall hat es einfach keinen Sinn, ein Objekt bei der Erstellung konstant zu machen, denn die Einstellungen werden erst nach dem Erstellen des Objektes vorgenommen.

Wenn Sie ein so erstelltes Objekt nun allerdings an eine Funktion übergeben und diese Funktion keine Veränderungen an dem Objekt vornimmt, ist die Wahrscheinlichkeit groß, dass der Parameter ein konstantes Objekt ist. Innerhalb einer solchen Funktion wäre das Objekt also konstant.

38.3 Zugriffsmethoden

Zugriffsmethoden sollten eigentlich vermieden werden, aber manchmal sind sie nützlich. Eine Zugriffsmethode macht nichts anderes, als eine Membervariable lesend oder schreibend zugreifbar zu machen:

```
class A{
public:
    void SetWert(int wert) { m_wert = wert; }
    int  GetWert()const    { return m_wert; }

private:
    int m_wert;
};
```

Get-Methoden sind immer konstant, da sie den Wert ja nur lesend zugreifbar machen sollen. Eine Set-Methode kann dagegen nie mit einem konstanten Objekt benutzt werden. Im Normalfall sollten Sie jedoch keine „Getter“ oder „Setter“ benötigen, wenn doch, müssen Sie sich Gedanken darüber machen, ob das Objekt die Logik möglicherweise nicht ausreichend kapselt.

Solche Einzeiler werden normalerweise einfach direkt in die Funktionsdeklaration geschrieben, dadurch sind sie auch gleich automatisch als `inline` ausgezeichnet. Dazu müssen Sie nur das

Semikolon durch den Funktionsrumpf ersetzen. Sollten Sie dennoch lieber eine eigene Definition für solche Methoden machen wollen, dann achten Sie darauf, diese bei der Definition als `inline` zu kennzeichnen. Falls Sie mit Headerdateien arbeiten, dann beachten Sie, dass der Funktionsrumpf bei `inline`-Methoden während des Kompilierens bekannt sein muss. Die Definition muss also mit in die Headerdatei, nicht wie gewöhnlich in die Quelldatei.

38.4 Überladen

Sie haben bereits gelernt, dass Funktionen überladen werden können, indem für den gleichen Funktionsnamen mehrere Deklarationen mit verschiedenen Parametern gemacht werden. Auch bei Klassenkonstruktoren haben Sie Überladung bereits kennengelernt. Für gewöhnliche Memberfunktionen ist eine Überladung ebenfalls nach den Ihnen bereits bekannten Kriterien möglich. Zusätzlich können Sie Memberfunktionen aber anhand des eben vorgestellten `const`-Modifizierers überladen.

```
class A{
public:
    void methode();          // Eine Methode
    void methode()const;    // Die überladene Version der Methode für konstante Objekte
};
```

Natürlich können auch Methoden mit Parametern auf diese Weise überladen werden. Die nicht-konstante Version wird immer dann aufgerufen, wenn Sie mit einem nicht-konstanten Objekt der Klasse arbeiten. Analog dazu wird die konstante Version aufgerufen, wenn Sie mit einem konstanten Objekt arbeiten. Wenn Sie nur eine konstante Version deklarieren, wird immer diese aufgerufen.

Sinnvoll ist diese Art der Überladung vor allem dann, wenn Sie einen Zeiger oder eine Referenz auf etwas innerhalb des Objekts zurückgeben. Wie Sie sich sicher erinnern, kann eine Überladung nicht anhand des Rückgabetyps einer Funktion (oder Methode) gemacht werden. Das folgende Beispiel wird Ihnen zeigen, wie Sie eine `const`-Überladung nutzen können, um direkte Manipulation von Daten innerhalb eines Objekts nur für nicht-konstante Objekte zulassen.

```
#include <iostream>

class A{
public:
    A():m_b(7) {} // m_b mit 7 initialisieren

    int&      B()      { return m_b; } // Zugriff lesend und schreibend
    int const& B()const { return m_b; } // Zugriff nur lesend

private:
    int m_b; // Daten
};

int main(){
```

```
A      objekt;          // Ein Objekt von A
A const objekt_const; // Ein konstantes Objekt von A

std::cout << objekt.B()          << std::endl; // Gibt 7 aus
std::cout << objekt_const.B() << std::endl; // Gibt 7 aus

objekt.B() = 9; // setzt den Wert von m_b auf 9
// objekt_const.B() = 9; // Produziert einen Kompilierfehler

std::cout << objekt.B()          << std::endl; // Gibt 9 aus
std::cout << objekt_const.B() << std::endl; // Gibt 7 aus
}
```

Im Kapitel über Operatoren überladen¹ werden Sie noch ein Beispiel zu dieser Technik kennenlernen, welches in der Praxis oft zu sehen ist.

38.5 Code-Verdopplung vermeiden

Im Beispiel von eben geben die beiden Funktionen lediglich eine Referenz auf eine Membervariable innerhalb des Objekts zurück. In der Regel wird eine solche Funktionen natürlich noch etwas mehr tun. Daher wäre es nötig, zwei Funktionen zu schreiben, die den gleichen Code enthalten. Das wiederum ist ausgesprochen schlechter Stil. Stellen Sie sich nur vor, Sie möchten die Funktion später aus irgendwelchen Gründen ändern, dann müssten Sie alle Änderungen an zwei Stellen im Code vornehmen.

Daher ist es sinnvoll, wenn eine Variante die andere aufruft. Hierfür sind einige Tricks nötig, da Sie einer der beiden Varianten beibringen müssen, eine Methode aufzurufen, die eigentlich nicht zum `const`-Modifizierer des aktuellen Objekts passt. Die konstante Variante verspricht, niemals eine Änderung am Objekt vorzunehmen, sie ist in ihren Möglichkeiten also stärker eingeschränkt. Die nicht konstante Version darf hingegen alles, was auch die konstante Version darf. Somit ist es sinnvoll, die konstante Version von der nicht-konstanten aufrufen zu lassen.

Um nun der nicht-konstanten Version beizubringen, dass sie ihr konstantes Äquivalent aufrufen soll, müssen wir zunächst einmal aus dem aktuellen Objekt ein konstantes Objekt machen. Jede Klasse enthält eine spezielle Variable, den sogenannten `this`-Zeiger, der innerhalb einer Membervariable einen Zeiger auf das aktuelle Objekt repräsentiert. Diesen `this`-Zeiger casten wir in einen Zeiger auf ein konstantes Objekt.

```
// nicht-konstante Version von B()
int A::B() {
    A const* objekt_const = static_cast< A const* >(this); // Konstantheit dazu casten
}
```

¹ Kapitel 28 auf Seite 173

Nun haben wir einen Zeiger auf das Objekt, über den wir nur konstante Methoden aufrufen können. Das Problem ist nun, dass die aufgerufene Methode natürlich auch eine Referenz auf eine konstante Variable aufruft.

```
int const& result = objekt_const->B(); // konstante Methodenversion aufrufen
```

Da wir ja wissen, dass das aktuelle Objekt eigentlich gar nicht konstant ist, können wir die Konstantheit für die zurückgegebene Referenz guten Gewissens entfernen. Allerdings ist der `static_cast`, den Sie bereits kennen, nicht dazu in der Lage. Um Konstantheit zu entfernen, benötigen Sie den `const_cast`. Beachten Sie jedoch, dass dieser Cast wirklich nur auf Variablen angewendet werden darf, die eigentlich nicht konstant sind!

```
return const_cast< int& >(result); // Konstantheit vom Rückgabewert wegcasten
}
```

Wenn Sie diese Anweisungen in einer zusammenfassen, sieht Ihre Klasse nun folgendermaßen aus.

```
class A{
public:
    // m_b mit 7 initialisieren
    A():m_b(7) {}

    // Ruft B()const auf
    int& B() { return const_cast< int& >( static_cast< A const* >(this)->B() ); }
    int const& B()const { return m_b; }

private:
    int m_b; // Daten
};
```

Wie schon gesagt, sieht diese Technik in unserem Beispiel überdimensioniert aus. Aber auch an dieser Stelle möchte ich Sie auf das Beispiel im Kapitel zur Operatorüberladung verweisen, wo sie, aufgrund der umfangreicheren konstanten Version, bereits deutlich angenehmer erscheint. Mit Performanceeinbußen haben Sie an dieser Stelle übrigens nicht zu rechnen. Ihr Compiler wird die casting-Operationen wegoptimieren.

38.6 Call by reference

Im Gegensatz zu den vorangegangenen Kapiteln dieses Abschnitts geht es diesmal nicht darum, wie man Objekte aufbaut, sondern wie man mit ihnen arbeitet. Im Kapitel über Funktionen hatten Sie schon ersten Kontakt mit der Wertübergabe als Referenz. Wie dort bereits erwähnt, ist es für Klassenobjekte effizienter, sie als Referenz an eine Funktion zu übergeben.

Bei der Übergabe einer Variablen als Wert muss von dieser Variable erst eine Kopie angefertigt werden. Zusätzlich führt der Kopierkonstruktor an dieser Stelle möglicherweise noch irgendwelche zusätzlichen Operationen aus, die Zeit kosten. Bei einer Übergabe als Referenz muss hingegen nur die Speicheradresse des Objekts kopiert werden. Wenn wir dann noch dafür sorgen, dass die Referenz auf ein konstantes Objekt verweist, haben wir eine fast kostenlose Übergabe und gleichzeitig die Sicherheit, dass die übergebene Variable innerhalb der Funktion nicht verändert wird.

```
class A{
public:
    int a, b, c, d;
};

class B{
public:
    // Das übergebene A-Objekt ist innerhalb der Methode konstant
    void methode(A const& parameter_name);
};
```

Die Methode von B kann auf die 4 Variablen von A lesend zugreifen, sie aber nicht verändern. Die Übergabe als Wert ist dann sinnvoll, wenn innerhalb einer Methode ohnehin eine Kopie der Variablen benötigt wird.

```
class B{
public:
    // Es wird eine Kopie des A-Objekts übergeben
    void methode(A parameter_name);
};
```

Nun können auf das A-Objekt beliebige lesende oder schreibende Operationen angewendet werden. Da sie auf einer Kopie ausgeführt werden, bleibt auch hier das Originalobjekt unverändert, aber eben zu dem Preis, dass zusätzlich Zeit und Speicherplatz benötigt werden, um eine Kopie des Objekts zu erstellen. In den meisten Fällen ist es nicht nötig, innerhalb einer Methode Schreiboperationen auszuführen. Verwenden Sie daher nach Möglichkeit die „Call by reference“-Variante.

Bei Rückgabewerten sollten Sie natürlich auf Referenzen verzichten, es sei denn, Sie wissen wirklich, was Sie tun. Andernfalls kann es schnell passieren, dass Sie eine Referenz auf eine Variable zurückgeben, die außerhalb der Methode gar nicht mehr existiert. Das wiederum führt zufallsbedingt zu Laufzeitfehlern und Programmabstürzen.

38.7 Wir empfehlen inline

In den Kapiteln über Funktionen² und Headerdateien³ haben Sie bereits einiges über `inline`-Funktionen erfahren. An dieser Stelle soll dieses Wissen nun auf den Gebrauch von `inline`-Klassenmethoden übertragen werden. Das folgende kleine Beispiel zeigt eine Klasse mit einer `inline`-Methode.

```
class A{
public:
    int methode()const;

private:
    int a_;
};

inline int A::methode()const{
    return a_;
}
```

Das Schlüsselwort `inline` steht vor der Methodendefinition, es kann auch in der Deklaration innerhalb der Klasse angegeben werden. Letzteres ist jedoch unübersichtlich und daher nicht zu empfehlen. Damit der Compiler den Methodenaufruf auch durch den Methodenrumpf ersetzen kann, steht die Methodendefinition ebenso wie die Klassendefinition in der Headerdatei und nicht in einer, eventuell zugehörigen, Implementierungsdatei. Wenn Sie eine Funktion direkt innerhalb einer Klassendefinition definieren, ist sie implizit `inline`, da solche Funktionen nahezu immer sehr kurz sind und sich somit gut zur Ersetzung eignen.

```
class A{
public:
    // methode ist implizit inline
    int methode()const { return a_; }

private:
    int a_;
};
```

Tip

Einige moderne Compilersuiten sorgen, je nach Einstellungen, dafür, dass Methoden `inline` verwendet werden, auch wenn sie nicht als `inline` deklariert wurden! In vielen Fällen können die Compiler ohnehin am besten selbst entscheiden, welche Funktion gute `inline`-Kandidaten sind und welche nicht.

² Kapitel 16 auf Seite 87

³ Kapitel 24 auf Seite 151

39 Vererbung

39.1 Einleitung

Wenn von Objektorientierung gesprochen wird, fällt früher oder später auch das Stichwort Vererbung. Auf dieser Seite lernen Sie anhand eines Beispiels die Grundprinzipien der Vererbung kennen.

39.2 Die Ausgangslage

Stellen wir uns vor, dass wir in einer Firma arbeiten und in einem Programm sämtliche Personen verwalten, die mit dieser Firma in einer Beziehung stehen. Über jede Person sind folgende Daten auf jeden Fall bekannt: Name, Adresse, Telefonnummer. Um alles zusammenzufassen, haben wir uns eine Klasse geschrieben, mit deren Hilfe wir eine einzelne Person verwalten können: (Aus Gründen der Einfachheit benutzen wir `strings`)

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

class Person{
    public:
    Person(string Name, string Adresse, string Telefon) :m_name(Name), m_adr(Adresse),
        m_tel(Telefon){}
    string getName(){ return m_name; }
    string getAdr(){ return m_adr; }
    string getTel(){ return m_tel; }
    void info(){ cout << "Name: " << m_name << " Adresse: " << m_adr << " Telefon: " << m_
tel <<
        endl; }

    private:
    string m_name;
    string m_adr;
    string m_tel;
};
```

Dies ist natürlich eine sehr minimale Klasse, aber schließlich geht es hier ja auch um die Vererbung.

39.3 Gemeinsam und doch getrennt

Die obige Klasse funktioniert ja eigentlich ganz gut, nur gibt es ein kleines Problem. In unserer Firma gibt es Mitarbeiter, Zulieferer, Kunden, Chefs, Diese sind zwar alle Personen, sie haben jedoch jeweils noch zusätzliche Attribute (z. B. ein Mitarbeiter hat einen Lohn, während ein Kunde eine KundenNr. hat). Jetzt könnten wir natürlich für jeden unterschiedlichen Typ eine eigene Klasse schreiben, den bereits vorhandenen Code der Klasse `Person` hineinkopieren und schließlich die entsprechenden Erweiterungen vornehmen. Dieser Ansatz hat jedoch einige Probleme:

- Er ist unübersichtlich.
- Es kann leicht zu Kopierfehlern kommen.
- Soll `Person` geändert werden, so muss jede Klasse einzeln bearbeitet werden.

Zum Glück bietet uns C++ aber ein mächtiges Hilfsmittel in Form der Vererbung. Anstatt alles zu kopieren, können wir den Compiler anweisen, die Klasse `Person` als Grundlage zu verwenden. Dies wird durch ein Beispiel klarer.

39.3.1 Beispiel

```
class Mitarbeiter : public Person {
    public:
Mitarbeiter(string Name, string Adresse, string Telefon, int Gehalt, int
MANr):Person(Name,Adresse,Telefon),
        m_gehalt(Gehalt), m_nummer(MANr) {}

    int getGehalt(){ return m_gehalt; }
int getNummer(){ return m_nummer; }

    private:
int m_gehalt;
int m_nummer;
};
```

39.4 Erläuterung des Beispiels

Nun wollen wir uns der Analyse des Beispiels zuwenden, um genau zu sehen, was passiert ist.

Das wichtigste im Code steht ganz am Anfang: `class Mitarbeiter : public Person`. Damit weisen wir den Compiler an, alle Elemente aus `Person` auch in `Mitarbeiter` zu übernehmen (z.B. hat `Mitarbeiter` jetzt auch eine `info`-Funktion und eine Membervariable `m_name`). Eine solche Klasse nennen wir abgeleitet/Kindklasse von `Person`.

Des Weiteren rufen wir im Konstruktor auch den Konstruktor von `Person` auf. Wir sparen uns also sogar diesen Code.

Benutzen können wir die Klasse wie gewohnt, mit der Ausnahme, dass wir jetzt auch alle Methoden von `Person` aufrufen können:

```
Mitarbeiter Mit("Erika Mustermann", "Heidestraße 17, Köln", "123/454", 4523, 12344209);
Mit.info();
cout << Mit.getGehalt() << endl;
cout << Mit.getName() << endl;
```

Thema wird später näher erläutert...

Warum wir das Schlüsselwort `public` verwenden, wird im Kapitel „Private und geschützte Vererbung“^a erklärt.

^a Kapitel 41 auf Seite 235

39.5 protected

Zum Schluss werden wir noch das Prinzip der Datenkapselung auf die Vererbung erweitern. Bis jetzt kennen wir die Schlüsselwörter `public` und `private`. Machen wir folgendes Experiment: Schreiben Sie eine neue Membermethode von `Mitarbeiter` und versuchen Sie auf `m_name` aus der `Person`-Klasse zuzugreifen. Der Compiler wird einen Fehler ausgeben. Warum?

`m_name` wurde in der `Person`-Klasse als `private` deklariert, das heißt, es kann nur von Objekten dieser Klasse angesprochen werden, nicht aber von abgeleiteten Klassen wie z. B. `Mitarbeiter`. Um zu vermeiden, dass wir `m_name` als `public` deklarieren müssen, gibt es `protected`. Es verhält sich ähnlich wie `private`, mit dem Unterschied, dass auch Objekte von Kindklassen auf mit diesem Schlüsselwort versehene Member zugreifen können.

40 Methoden (nicht) überschreiben

Dieses Kapitel enthält mehrere inhaltliche Fehler. Ich kümmere mich schnellstmöglich um die Behebung.

Freundliche Grüße --Prog¹ 22:46, 16. Jun. 2011 (CEST)

40.1 Einleitung

Als Ausgangspunkt nehmen wir die beiden Klassen `Person` und `Mitarbeiter` aus dem vorhergehenden Abschnitt.

40.2 Zwei gleiche Namen, ein Aufruf

Wir haben der Klasse `Mitarbeiter` zwar schon neue Funktionen hinzugefügt. Bis jetzt hatten diese einzigartige Namen, die nicht in der Basisklasse `Person` vorgekommen sind (z. B. `get_salary()`). Nun machen wir folgenden Versuch: Wir fügen `Mitarbeiter` eine Methode `void info()` hinzu. Da ein solcher Member bereits in `Person` existiert erwarten wir, dass uns der Übersetzer einen Fehler ausgibt.

Entgegen unserer Erwartung jedoch wird alles fehlerfrei übersetzt. Dies kommt daher, dass die Methode aus der Elternklasse ganz einfach überschrieben wird. (Dies ist aber nur der Fall, wenn die Signaturen² der Methoden, also Name, Parameter und Rückgabewert, übereinstimmen). Das heißt aber auch, dass wir aufpassen müssen, welches `void info()` wir aufrufen. Wenn wir nämlich in einer Memberfunktion von `Mitarbeiter` `void info()` aufrufen, so wird nicht die ursprüngliche Funktion aus `Person`, sondern die „neue“ Implementierung aus `Mitarbeiter` ausgeführt.

40.3 Die Methode des Vaters aufrufen

Um eine überschriebene Methode der Basisklasse aufzurufen, benutzen wir folgenden Syntax:

```
«Klassenname»::«Methodenname»(«Parameter...»);
```

¹ <http://de.wikibooks.org/wiki/Benutzer%3AProg>

² <http://de.wikipedia.org/wiki/Signatur%20%28Programmierung%29>

Hinweis

Obwohl es möglich ist, jede Funktion einer Basisklasse zu überschreiben, ist dies nicht empfehlenswert. Besser ist die Benutzung von virtuellen Methoden^a, die genau zu diesem Zweck vorhanden sind.

^a <http://de.wikibooks.org/wiki/C%2B%2B-Programmierung%2F%20Objektorientierte%20Programmierung%2F%20Virtuelle%20Methoden>

Hinweis

Wir wollen hier noch einmal in Erinnerung rufen, dass Methoden nicht nur überschrieben, sondern auch überladen^a werden können. Dabei wird dieselbe Signatur für eine andere Funktionalität verwendet.

^a Kapitel 38.4 auf Seite 223

41 Private und geschützte Vererbung

41.1 Einleitung

Bis jetzt haben wir alle Vererbungen mit dem Schlüsselwort `public` vorgenommen (dies wird „öffentliche Vererbung“ genannt). Nun werden wir lernen, was passiert, wenn statt `public` `private` bzw. `protected` verwendet wird.

41.2 Private Vererbung

Verwenden wir `private`, so bedeutet dies, dass alle geschützten und öffentlichen Membervariablen bzw. Memberfunktion aus der Basisklasse privat werden, von außen also nicht sichtbar sind.

Wenn kein Schlüsselwort zur Ableitung angegeben wird, handelt es sich automatisch um `private` Vererbung.

41.3 Geschützte Vererbung

Geschützte Vererbung verläuft analog zur privaten Vererbung und sagt aus, dass alle geschützten und öffentlichen Member der Elternklasse im Bereich `protected` stehen.

41.4 Wann wird was benutzt?

Um festzustellen wann welche Art von Vererbung eingesetzt wird, gibt es zwei unterschiedliche Arten wie eine abgeleitete Klasse im Verhältnis zu ihrer Basisklasse stehen kann.

1. ist-Beziehung: "Klasse B ist eine Klasse A" → Klasse B erbt `public` von A. (Beispiel: Ein Arbeiter ist eine Person)
2. hat-Beziehung: "Klasse B hat ein Klasse A Objekt" → Klasse B erbt `private` von A. (Beispiel: Ein Mensch hat ein Herz)

Hinweis

Meistens wird mithilfe von `public` vererbt. Andere Typen von Vererbung werden nur selten bis gar nicht benutzt. Oft ist es sinnvoll statt einer privaten Vererbung ein Memberobjekt der entsprechenden Klasse zu verwenden.

42 Funktionstemplates

Ein *Funktionstemplate* ist eine „Mustervorlage“ oder „Schablone“, die dem Compiler mitteilt, wie eine Funktion erzeugt werden soll. Aus einem Template können semantisch gleichartige Funktionen mit verschiedenen Parametertypen erzeugt werden. Für alle, die eben in Panik zu Wiktionary¹ oder einem noch schwereren Wörterbuch gegriffen haben: semantisch heißt hier so viel, wie allgemein gehalten in Bezug auf den Datentyp. Die Funktion hat also immer die gleiche Parameteranzahl, die Parameter haben aber unterschiedliche Datentypen. Das ist zwar nicht ganz korrekt ausgedrückt, aber eine genaue Definition würde das Wort wieder so unverständlich machen, dass es ohne Erklärung weniger Schrecken verbreitet hätte. Lesen Sie einfach das Kapitel, dann werden Sie auf einige der „fehlenden Klauseln“ selbst aufmerksam werden.

Das nachfolgende Template kann Funktionen generieren, welche die größere von zwei Zahlen zurückliefern. Der Datentyp spielt dabei erst einmal keine Rolle. Wichtig ist nur, dass es *zwei* Variablen *gleichen* Typs sind. Die Funktion liefert dann einen Wert zurück, der ebenfalls dem Datentyp der Parameter entspricht.

```
template <typename T>
T maxi(T obj1, T obj2){
    if(obj1 > obj2)
        return obj1;
    else
        return obj2;
}
```

Das Template wird mit dem gleichnamigen, aber kleingeschriebenen Schlüsselwort `template` eingeleitet. Danach folgt eine spitze Klammer und ein weiteres Schlüsselwort namens `typename`. Früher hat man an dieser Stelle auch das gleichbedeutende Schlüsselwort `class` benutzt. Das funktioniert heute immer noch, ist aber nicht mehr üblich. Erstens Symbolisiert `typename` besser was folgt, nämlich einen Platzhalter, der für einen Datentyp steht, und zweitens ist es einfach übersichtlicher. Dies gilt insbesondere bei Klassentemplates, die aber Thema des nächsten Kapitels sind.

Als Nächstes folgt wie schon gesagt der Platzhalter. Er kann die gleichen Namen haben wie eine Variable sie haben könnte. Es ist aber allgemein üblich, den Platzhalter als `T` (für Typ) zu bezeichnen. Das soll natürlich nur ein Vorschlag sein – viele Programmierer haben da auch ihren eigenen Stil. Wichtig ist nur, dass Sie sich nicht zu viele Bezeichner suchen, es sei denn Sie haben ein ausgesprochenes Talent dafür, aussagekräftige Namen zu finden. Am Ende dieser Einführung wird die spitze Klammer geschlossen. Dann wird ganz normal die Funktion geschrieben mit der einen Ausnahme, dass anstatt eines genauen Datentypes einfach der Platzhalter angegeben wird.

¹ <http://de.wiktionary.org/wiki/semantisch>

Sie können diese Funktion dann z.B. mit `int`- oder `float`-Werten aufrufen:

```
int a = 3, b = 5;
int m = maxi(a, b);           // m == 5
m = maxi(3, 7);              // m == 7

float a = 3.3, b = 3.4;
float m = maxi(a, b);        // m == 3.4
m = maxi(a, 6.33);          // m == 6.33
```

Bei jedem ersten Aufruf einer solchen Funktion erstellt der Compiler aus dem Template eine echte Funktion, wobei er den Platzhalter durch den tatsächlichen Datentyp ersetzt. Welcher Datentyp das ist, entscheidet der Compiler anhand der übergebenen Argumente. Bei diesem Aufruf findet keine Typumwandlung statt. Es ist also nicht möglich, einen `int`- und einen `float`-Wert an die Templatefunktion zu übergeben. Das nächste Beispiel wird das verdeutlichen:

```
int a = 3;
float b = 5.4;
float m;                       // Ergebnis

m = maxi(a, b);                // funktioniert nicht
m = maxi(4.5, 3);              // funktioniert nicht
m = maxi(4.5, 3.0);           // funktioniert
m = maxi(a, 4.8);             // funktioniert nicht
m = maxi(7, b);                // funktioniert nicht
m = maxi(a, 5);                // funktioniert
m = maxi(7.0, b);              // funktioniert
```

Möchten Sie eine andere Variante als jene, die anhand der Argumente aufgerufen wird benutzen, müssen Sie die gewünschte Variante explizit angeben:

```
float a = 3.0, b = 5.4;
float m = maxi<int>(a, b);     // m == 5.0
int n = maxi<int>(a, b);      // n == 5
```

Diese Templatefunktion kann nun mit allen Datentypen aufgerufen werden, die sich nach der Ersetzung des Platzhalters durch den konkreten Datentyp auch übersetzen lassen. In unserem Fall wäre die einzige Voraussetzung, dass auf den Datentyp der Operator `>` angewandt werden kann und ein Kopierkonstruktor vorhanden ist, da die Werte mit „call by value“, also als Kopie übergeben werden.

Das ist beispielsweise auch bei C++-Zeichenketten, also Objekte der Klasse `std::string` mit Headerdatei: `string`, der Fall. Daher ist folgendes problemlos möglich:

```
std::string s1("alpha");
std::string s2("omega");
std::string smax = maxi(s1, s2);           //smax == omega
```

42.1 Spezialisierung

Im Gegensatz zu normalen Funktionen können Templatefunktionen nicht einfach überladen werden, da das Template ja eine Schablone für Funktionen ist und somit bereits für *jeden* Datentyp eine Überladung bereitstellt. Manchmal ist es aber so, dass das allgemein gehaltene Template für bestimmte Datentypen keine sinnvolle Funktion erzeugt. In unserem Beispiel würde dies für C-Strings zutreffen:

```
const char* maxi(const char* str1, const char* str2){
    if(str1 > str2)
        return str1;
    else
        return str2;
}
```

Beim Aufruf von `maxi("Ich bin ein String!", "Ich auch!")` würde die obenstehende Funktion aus dem Template erzeugt. Das Ergebnis wäre aber nicht wie gewünscht der dem ASCII-Code² nach kleinere Zeichenkette, sondern einfach die, dessen Adresse im Arbeitsspeicher kleiner ist. Um C-Strings zu vergleichen gibt es eine Funktion in der Headerdatei `cstring` (oder in der veralteten Fassung `string.h`).

Um nun dem Compiler beizubringen, wie er `maxi()` für C-Strings zu implementieren hat, muss diese Variante spezialisiert werden. Das heißt, es wird für einen bestimmten Datentyp (in diesem Fall `const char*`) eine von der Mustervorlage (also dem Template) abweichende Version definiert. Für das nächste Beispiel muss die Headerdatei `cstring` inkludiert werden.

```
template <>
const char* maxi(const char* str1, const char* str2){
    if(strcmp(str1, str2) > 0) // strcmp vergleicht zwei C-Strings
        return str1;
    else
        return str2;
}
```

Eine Spezialisierung leitet man immer mit dem Schlüsselwort `template` gefolgt von einer öffnenden und einer schließenden spitzen Klammer ein. Der Name der Funktion ist identisch mit dem des Templates (`maxi`), und alle `T` werden durch den konkreten Typ `const char*` ersetzt. Der Rumpf

² <http://de.wikipedia.org/wiki/ASCII>

der Funktion kann dann völlig beliebig gestaltet werden. Natürlich ist es Sinn und Zweck der Sache, ihn so zu schreiben, dass die Spezialisierung das gewünschte sinnvolle Ergebnis liefert.

Übrigens wäre es auch möglich, den als Beispiel dienenden Aufruf von oben als `maxi<std::string>("Ich bin ein String!", "Ich auch!")` zu schreiben. Dann würden die beiden C-Strings vor dem Aufruf in C++-Strings umgewandelt, und die können ja problemlos mit `>` verglichen werden. Diese Methode hat aber drei entscheidende Nachteile gegenüber einer Spezialisierung für C-Strings:

- Die Headerdatei `string` muss jedes Mal inkludiert werden
- Der Aufruf von `maxi("Ich bin ein String!", "Ich auch!")` ist weiterhin problemlos möglich und liefert auch weiterhin ein dem Zufall überlassenes Ergebnis
- Diese Art der Lösung ist viel langsamer als die Variante mit der Spezialisierung

42.2 Zwischen Spezialisierung und Überladung

Die folgende *Überladung* der Templatefunktion `maxi()` ist ebenfalls problemlos möglich und scheint auf den ersten Blick prima zu funktionieren:

```
// template <> - ohne das ist es keine Spezialisierung, sondern eine Überladung
const char* maxi(const char* str1, const char* str2){
    if(strcmp(str1, str2) > 0)
        return str1;
    else
        return str2;
}
```

Hinweis

Um es gleich vorwegzunehmen: das hier beschriebene Problem lässt sich leicht herbeiführen, ist aber nur sehr schwer zu finden. Daher achten Sie darauf, es zu vermeiden. Vergessen Sie auf keinen Fall `template <>` voranzustellen, wenn Sie ein Template spezialisieren.

Das Ausführen der nächsten beiden Codezeilen führt zum Aufruf verschiedener Funktionen. Die Funktionen haben identische Parametertypen und Namen sowie die gleiche Parameteranzahl, aber eben *nicht* den gleichen Code.

```
// Überladene Version - Ergebnis korrekt
cout << maxi("Ich bin ein String!", "Ich auch!");

// Vom Template erzeugte Version - Ergebnis zufällig (Speicheradressen)
cout << maxi<const char*>("Ich bin ein String!", "Ich auch!");
```

So etwas sollten Sie um jeden Preis vermeiden! Selbst ein erfahrener Programmierer dürfte über einen Fehler dieser Art erst einmal sehr erstaunt sein. Sie können die beiden Codezeilen ja jemandem zeigen, der sich gut auskennt und ihn Fragen, was der Grund dafür sein könnte, dass die Varianten

zwei verschiedene Ergebnisse liefern. Ohne die Implementierung von `maxi()` zu sehen wird er höchst wahrscheinlich nicht darauf kommen. Ein solcher Fehler ist nur schwer zu finden. Es ist bereits eine Katastrophe, ihn überhaupt erst einmal zu machen. Merken Sie sich also unbedingt, dass man `template <>` nur vor eine Funktion schreibt, wenn man für einen bestimmten Datentyp die Funktionsweise ändern will, weil die Standardvorlage kein sinnvolles Ergebnis liefert.

42.3 Überladen von Template-Funktionen

Ein Template ist in der Regel eine Vorlage für Funktionen, deren Parameter sich lediglich im Typ unterscheiden. Das Überladen von Funktionen ist aber auch durch eine andere Parameterzahl oder eine andere Anordnung der Parametertypen möglich. Unter dieser Überschrift wird die `maxi()`-Template um einen optionalen dritten Parameter erweitert. Was bisher vorgestellt wurde, wird in diesem Beispielprogramm zusammengefasst:

```
#include <iostream>           // Ein und Ausgabe
#include <string>             // C++-Strings
#include <cstring>           // Vergleiche von C-Strings

using namespace std;

// Unsere Standardvorlage für maxi()
template <typename T>
T maxi(T obj1, T obj2){
    if(obj1 > obj2)
        return obj1;
    else
        return obj2;
}

// Die Spezialisierungen für C-Strings
template <>                 // Spezialisierung
const char* maxi(const char* str1, const char* str2){ // statt T, const char*
    if(strcmp(str1, str2) > 0)
        return str1;
    else
        return str2;
}

int main(){
    int    a = 3;           // Ganzzahlige Variable
    float  b = 5.4;        // Gleitkommazahl-Variablen

    int    m;              // Ganzzahl-Ergebnis
    float  n;              // Kommazahl-Ergebnis

    // Je nachdem, wo die Typumwandlungen stattfinden,
```

```
// sind die Ergebnisse unterschiedlich
m = maxi<int>(a, b);           // m == 5
n = maxi<int>(a, b);           // n == 5.0
m = maxi<float>(a, b);         // m == 5
n = maxi<float>(a, b);         // n == 5.4

// Aufgerufen wird unabhängig vom Ergebnistyp m:
m = maxi(3, 6);               // maxi<int>()
m = maxi(3.0, 6.0);           // maxi<float>()
m = maxi<int>(3.0, 6);         // maxi<int>()
m = maxi<float>(3, 6.0);       // maxi<float>()

// Aufruf von maxi<std::String>()
string s1("alpha");
string s2("omega");
string smax = maxi(s1,s2);     // smax == omega

// Aufruf von maxi<const char*>()
smax = maxi("alpha","omega");  // Spezialisierung wird aufgerufen

return 0;                      // Programm erfolgreich durchlaufen
}
```

Eine Ausgabe enthält das Programm noch nicht, aber das sollten Sie ja problemlos schaffen. Die Datei `iostream` ist auch schon inkludiert.

Nun aber zur Überladung der Templatefunktion: um von `maxi()` zu verlangen, den größten von drei Werten zurückzugeben, muss der Parameterliste ein weiterer Platzhalter, also noch ein `T obj3` hinzugefügt werden. Gleiches gilt für unsere C-String-Spezialisierung. Da wir ja aber auch die Möglichkeit haben wollen, nur den größeren von zwei Werten zu bestimmen, müssen wir die bisherige Variante stehen lassen und die mit drei Argumenten hinzufügen. Das Ganze sieht dann so aus:

```
#include <iostream>           // Ein und Ausgabe
#include <string>              // C++-Strings
#include <cstring>             // Vergleiche von C-Strings

using namespace std;

// Unsere Standardvorlage für maxi() mit 2 Argumenten
template <typename T>
T maxi(T obj1, T obj2){
    if (obj1 > obj2)
        return obj1;
    else
        return obj2;
}
```

```
// Die Spezialisierungen für C-Strings mit zwei Argumenten
template <> // Spezialisierung
const char* maxi(const char* str1, const char* str2){
    if(strcmp(str1, str2) > 0)
        return str1;
    else
        return str2;
}

// Unsere Standardvorlage für maxi() mit drei Argumenten
template <typename T>
T maxi(T obj1, T obj2, T obj3){
    if(obj1 > obj2)
        if(obj1 > obj3)
            return obj1;
        else
            return obj3;
    else if(obj2 > obj3)
        return obj2;
    else
        return obj3;
}

// Die Spezialisierungen für C-Strings mit drei Argumenten
template <> // Spezialisierung
const char* maxi(const char* str1, const char* str2, const char* str3){
    if(strcmp(str1, str2) > 0){
        if(strcmp(str1, str3) > 0)
            return str1;
        else
            return str3;
    }
    else if(strcmp(str2, str3) > 0)
        return str2;
    else
        return str3;
}

int main(){
    cout << "Beispiele für Ganzzahlen:\n";
    cout << "      (2, 7, 4) -> " << maxi(2, 7, 4) << endl;
    cout << "      (2, 7) -> " << maxi(2, 7) << endl;
    cout << "      (7, 4) -> " << maxi(7, 4) << endl;

    cout << "\nBeispiele für Kommazahlen:\n";
    cout << "(5.7, 3.3, 8.1) -> " << maxi(5.7, 3.3, 8.1) << endl;
    cout << "      (7.7, 7.6) -> " << maxi(7.7, 7.6) << endl;
}
```

```
cout << "      (1.9, 0.4) -> " << maxi(1.9, 0.4)      << endl;

cout << "\nBeispiele für C-Strings:\n";
cout << "(ghi, abc, def) -> " << maxi("ghi", "abc", "def") << endl;
cout << "      (ABC, abc) -> " << maxi("ABC", "abc")      << endl;
cout << "      (def, abc) -> " << maxi("def", "abc")      << endl;

return 0;                                // Programm erfolgreich durchlaufen
}
```

Ausgabe

Beispiele für Ganzzahlen:

```
(2, 7, 4) -> 7
(2, 7) -> 7
(7, 4) -> 7
```

Beispiele für Kommazahlen:

```
(5.7, 3.3, 8.1) -> 8.1
(7.7, 7.6) -> 7.7
(1.9, 0.4) -> 1.9
```

Beispiele für C-Strings:

```
(ghi, abc, def) -> ghi
(ABC, abc) -> abc
(def, abc) -> def
```

So Sie verstanden haben, wie aus einer Template eine konkrete Funktion erzeugt wird, fragen Sie sich vielleicht, ob das wirklich so einfach ist. Glücklicherweise kann diese Frage bejaht werden. Man stellt sich einfach das Funktionstemplate als mehrere separate Funktionen vor, die durch unterschiedliche Parametertypen überladen sind und wendet dann die üblichen Regeln zur Überladung von Funktionen an. Wenn man, wie in diesem Fall getan, ein weiteres Funktionstemplate anlegt, werden dadurch natürlich auch gleich wieder eine ganze Reihe einzelner Funktionen hinzugefügt. Im Endeffekt überlädt man also nicht das Template, sondern immer noch die aus ihm erzeugten Funktionen. Was Spezialisierungen angeht, gehören die natürlich immer zu dem Template, das eine Verbindung zu seiner semantische Parameterliste besitzt.

42.4 Templates mit mehreren Parametern

Natürlich kann ein Template auch mehr als nur einen Templateparameter übernehmen. Nehmen Sie an, Sie benötigen eine kleine Funktion, welche die Summe zweier Argumente bildet. Über den Typ der Argumente wissen Sie nichts, aber der Rückgabetyt ist Ihnen bekannt. Eine solche Funktion könnte folgendermaßen implementiert werden:

```
#include <iostream>                                // Ein und Ausgabe
```

```
using namespace std;

// Unsere Vorlage für die Summenberechnung
template <typename R, typename Arg1, typename Arg2>
R summe(Arg1 obj1, Arg2 obj2){
    return obj1 + obj2;
}

int main(){
    float a = 5.4;                // Gleitkommazahl-Variable
    int b = -3;                   // Ganzzahlige Variable

    cout << summe<int>(a, b)      << endl; // int           summe(float obj1, int obj2);
    cout << summe<float>(a, b)    << endl; // float          summe(float obj1, int obj2);
    cout << summe<double>(a, b)  << endl; // double         summe(float obj1, int obj2);
    cout << summe<unsigned long>(a, b) << endl; // unsigned long summe(float obj1, int obj2);

    cout << summe<float, int, int>(a, b) << endl; // float          summe(int obj1, int obj2);

    cout << summe<float, long>(a, b) << endl; // float          summe(long obj1, int obj2);

    return 0;                     // Programm erfolgreich durchlaufen
}

```

Ausgabe

```
2
2.4
2.4
2
2
2

```

Beim fünften Aufruf sind alle Argumente explizit angegeben. Wie Sie unschwer erkennen werden, kann der Compiler bei dieser Funktion den Rückgabotyp nicht anhand der übergebenen Funktionsparameter ermitteln. Daher müssen Sie ihn immer explizit angeben, wie die ersten vier Aufrufe es zeigen.

Die Templateargumente werden beim Funktionsaufruf in der gleichen Reihenfolge angegeben, wie sie bei der Deklaration des Templates in der Templateargumentliste stehen. Wenn dem Compiler weniger Argumente übergeben werden als bei der Deklaration angegeben, versucht er die übrigen anhand der Funktionsargumente zu bestimmen.

Das bedeutet für Sie, dass Argumente, die man immer explizit angegeben werden muss, auch bei der Deklaration an vorderster Stelle der Templateargumentliste stehen müssen. Das folgende Beispiel verdeutlicht die Folgen bei Nichteinhaltung dieser Regel:

```
#include <iostream>                // Ein und Ausgabe

using namespace std;

```

```
// Unsere Vorlage für die Summenberechnung
template <typename Arg1, typename Arg2, typename R> // Rückgabetyt als letztes angegeben
R summe(Arg1 obj1, Arg2 obj2){
    return obj1 + obj2;
}

int main(){
    float a = 5.4;           // Gleitkommazahl-Variable
    int b = -3;             // Ganzzahlige Variable

    cout << summe<int>(a, b) << endl; // ??? summe(int obj1, int obj2);
    cout << summe<float>(a, b) << endl; // ??? summe(float obj1, int obj2);
    cout << summe<double>(a, b) << endl; // ??? summe(double obj1, int obj2);
    cout << summe<unsigned long>(a, b) << endl; // ??? summe(unsigned long obj1, int obj2);

    cout << summe<float, int, int>(a, b) << endl; // int summe(float obj1, int obj2);

    cout << summe<float, long>(a, b) << endl; // ??? summe(float obj1, long obj2);

    return 0;               // Programm erfolgreich durchlaufen
}
```

Nur der Aufruf (Nr. 5), bei dem auch das letzte Templateargument, also der Rückgabetyt explizit angegeben wurde, ließe sich übersetzen. Alle anderen bemängelt der Compiler. Sie müssten also alle Templateargumente angeben, um den Rückgabetyt an den Compiler mitzuteilen. Das schmeißt diese wundervolle Gabe des Compilers, selbständig anhand der Funktionsparameter die richtige Templatefunktion zu erkennen, völlig über den Haufen. Hinzu kommt in diesem Beispiel noch, dass die Reihenfolge bei einer expliziten Templateargumentangabe alles andere als intuitiv ist.

42.5 Nichttypargumente

Neben Typargumenten können Templates auch ganzzahlige Konstanten als Argument übernehmen. Die folgende kleine Templatefunktion soll alle Elemente eines Arrays (Datenfeldes) beliebiger Größe mit einem Startwert initialisieren.

```
#include <cstddef>           // Für den Typ size_t

template <std::size_t N, typename T>
void array_init(T (&array)[N], T const &startwert){
    for(std::size_t i=0; i!=N; ++i)
        array[i]=startwert;
}
```

Die Funktion übernimmt eine Referenz auf ein Array vom Typ `T` mit `N` Elementen. Jedem Element wird der Wert zugewiesen, welcher der Funktion als Zweites übergeben wird. Der Typ `std::size_t` wird übrigens von der C++-Standardbibliothek zur Verfügung gestellt. Er ist in der Regel als `typedef` auf `unsigned long` deklariert.

Ihr Compiler sollte in der Lage sein, sowohl den Typ `T` als auch die Größe des Arrays `N` selbst zu ermitteln. In einigen Fällen kann es jedoch vorkommen, dass der Compiler nicht in der Lage ist, die Größe `N` zu ermitteln. Da sie als erstes Templateargument angegeben wurde, reicht es aus, nur sie explizit anzugeben und den Compiler in diesen Fällen zumindest den Typ `T` selbst ermitteln zu lassen.

42.6 inline

Obwohl Funktionstemplates im Header definiert werden müssen, sind sie nicht automatisch auch `inline`. Wenn Sie eine Funktionstemplate als `inline` deklarieren möchten, geben Sie das Schlüsselwort zwischen Templateparameterliste und dem Prototypen an.

43 Container

43.1 Containerklassen

In diesem Kapitel soll das Konzept generischer Container vorgestellt werden. Die C++ Standardbibliothek stellt folgende Typen zur Verfügung:

Sequentielle Containerklassen

- `vector`
- `deque`
- `list`

Assoziative Containerklassen

- `hash_map`
- `hash_multimap`
- `hash_multiset`
- `hash_set`
- `map`
- `multimap`
- `multiset`
- `set`

Containeradapterklassen (keine C++ Standardbibliothek-Iteratoren)

- `priority_queue`
- `queue`
- `stack`

Container sind Behälter für Objekte. Dabei wird immer eine Kopie des Objektes in dem Container gespeichert. Das hat den Vorteil, dass sich die Speicherverwaltung vereinfacht, denn das Objekt verliert seine Gültigkeit, wenn das Containerobjekt aufgelöst wird. Der Nachteil liegt auf der Hand - durch das Erstellen einer Kopie geht Zeit und Speicherplatz verloren. Es gibt aber auch Wege, Zeiger in Containern zu speichern, was aber an späterer Stelle erläutert werden soll. Ein weiterer Vorteil bei der Verwendung von Containern ist ihre einheitliche Schnittstelle zu Elementfunktionen. Das vereinfacht den Einsatz von bestimmten Algorithmen, wie sie ebenfalls in der STL zu finden sind.

43.1.1 Containereigenschaften

Ein *vector* ist im Wesentlichen ein dynamisches Feld, das je nach Bedarf seine Größe dynamisch verändern kann. Allerdings sind dabei einige Dinge zu beachten. Auf ein beliebiges Objekt innerhalb des Feldes kann sehr effizient unter direkter Adressierung zugegriffen werden - man spricht daher auch von konstantem Aufwand, da sich die Zugriffszeit nicht ändert, wenn das Feldobjekt wächst.

Ebenso ist das Anhängen und Entfernen von Objekten am Ende effizient, nicht so hingegen am Anfang oder in der Mitte (linearer Aufwand). Der Container *deque* beherrscht ebenso wie *vector* das schnelle Einfügen von Objekten am Ende und dazu noch am Anfang des Feldes. Hingegen ist auch hier das Einfügen von Objekten in der Mitte sehr aufwendig. Auch dieser Containertyp unterstützt *Random-Access-Operatoren*, d.h. der Zugriff auf ein beliebiges Element ist sehr effizient. Der dritte sequenzielle Container ist der *list*-Container. Dieser Typ unterstützt nur sogenannte *Bidirectional-Iteratoren*. Dadurch ist ein direkter Indexzugriff wie bei den anderen Containern nicht möglich. Der Vorteil dieses Containers ist allerdings des effiziente Einfügen und Entfernen von Objekten an beliebigen Positionen des Feldes. Durch diese deutlichen Unterschiede sollte sich der Programmierer schon bei der Planung des Programms darüber im Klaren sein, welcher Containertyp am besten zu seinen Anforderungen passt.

43.1.2 Grundlegende Mitgliedsfunktionen der sequentiellen Containerklassen

Sequenzielle Container besitzen Funktionen wie `front()` oder auch `back()`. Damit kann auf das erste bzw. letzte Element des Containers zugegriffen werden. Darüber hinaus gibt es Funktionen, die das Anhängen und Entfernen von Elementen am Ende eines Feldes erlauben (`push_back()`, `pop_back()`). Das Einfügen/Entfernen von Objekten am Anfang wird nur von den Typen *deque* und *list* beherrscht (mittels: `push_front()` und `pop_front()`). Das direkte Indizieren beherrschen *vector* und *deque* (mittels `at()` bzw. mit `operator[]`). Die folgende Tabelle soll dies vergleichend veranschaulichen.

	vector	deque	list
<code>front()</code>	x	x	x
<code>back()</code>	x	x	x
<code>push_back()</code>	x	x	x
<code>pop_back()</code>	x	x	x
<code>push_front()</code>	/	x	x
<code>pop_front()</code>	/	x	x
<code>at()</code>	x	x	/
<code>operator[]</code>	x	x	/

Darüber hinaus sind weitere Funktionen definiert, deren konkrete Implementierung vom jeweiligen Containertyp abhängig sind. Als Beispiel sei hier der *vector*-Container gewählt.

Funktionsname	Beschreibung
<code>assign</code>	Zuweisen von Elementen zum <i>vector</i>
<code>begin</code>	gibt einen Iterator auf das erste Element zurück
<code>capacity</code>	gibt Anzahl an Elementen zurück die vom <i>vector</i> aufgenommen werden können
<code>clear</code>	löscht alle Elemente
<code>empty</code>	gibt <i>wahr</i> zurück wenn Container leer ist
<code>end</code>	gibt einen Iterator auf das letzte Element zurück

<code>erase</code>	löscht das Element oder eine Reihe von Elementen
<code>insert</code>	fügt Elemente in den Container ein
<code>max_size</code>	gibt die maximal mögliche Zahl von Elementen des Containers an
<code>rbegin</code>	gibt einen <i>reverse</i> Iterator auf den Anfang zurück
<code>rend</code>	gibt einen <i>reverse</i> Iterator auf das Ende zurück
<code>reserve</code>	setzt eine minimale Kapazität des Containers
<code>resize</code>	ändert die Größe des Containers
<code>size</code>	gibt aktuelle Größe des Feldes zurück
<code>swap</code>	tauscht den Inhalt des Containers mit einem anderen

43.1.3 Adaptive Container

Dies sind spezielle Container, die auch Containeradapter genannt werden. Insbesondere lassen sich auf diese Container keine Iteratoren definieren. Der Zugriff erfolgt ausschließlich über die Elementfunktionen dieser Typen. In der STL finden sich folgende Typen:

- `stack`
- `queue`
- `priority_queue`

44 Interne Zahlendarstellung

Wie bereits in der Einführung der Variablen¹ erwähnt, sind Variablen nichts weiter als eine bequeme Schreibweise für eine bestimmte Speicherzelle bzw. eine Block von Speicherzellen. Die kleinste Speicherzelle, welche direkt vom Prozessor "adressiert" werden kann, ist ein Byte. Ein Byte besteht (heutzutage) aus 8 Bit. So stellen Sie ein Byte grafisch dar:

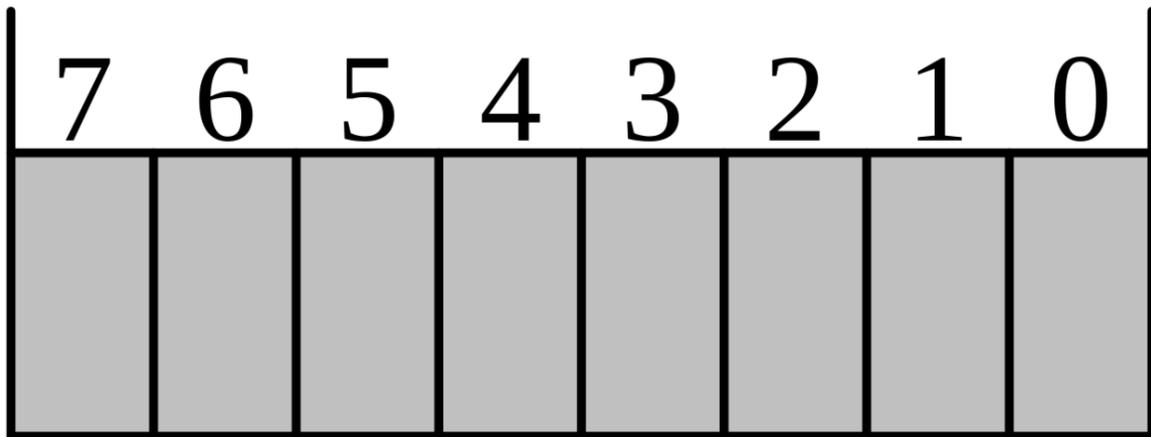


Abb. 4

Das unterste Bit, Bit 0, wird als "niederwertigstes Bit" bezeichnet (englisch: "least significant bit: LSB"), das oberste Bit, Bit 7, als "höchstwertiges Bit" (englisch: "most significant bit: MSB").

Jedes dieser acht Bits kann den Wert 0 oder 1 annehmen. Damit kann ein Byte $2^8=256$ verschiedene Werte annehmen: Wie diese 256 verschiedenen Werte "interpretiert" werden, ist abhängig vom Datentyp. Der C++ Datentyp, der genau ein Byte repräsentiert, ist `char`, also zu Deutsch: "Zeichen". Jeder Buchstabe, jede Ziffer, jedes der so genannten "Sonderzeichen" kann in einem solchen `char` gespeichert werden. (Zumindestens sofern Sie in Westeuropa bleiben. Für die vielen verschiedenen Zeichen z.B. der asiatischen Sprachen genügt ein `char` nicht mehr, doch dafür gibt es den Datentyp `wchar_t`, der größer als ein Byte ist und somit mehr Zeichen darstellen kann.)

Das Zeichen 'A' wird z.B. durch die Bitkombination 01000001 dargestellt, das Zeichen '&' durch 00100110. Die Zuordnung von Bitwerten zu Zeichen ist im (ASCII-Code²) festgeschrieben, den heute eigentlich alle Computer verwenden. Im Anhang D ist der ASCII-Code aufgelistet.

Die Bitkombinationen eines Bytes können Sie natürlich auch als Zahlen auffassen. Die Bitkombinationen 00000000 bis 01111111 entsprechen dann den Zahlen 0 bis 127. Leider ist nun die Frage, welche Zahlen den Bitfolgen 10000000 bis 11111111 entspricht, nicht mehr so eindeutig.

¹ Kapitel 9 auf Seite 33

² <http://de.wikipedia.org/wiki/American%20Standard%20Code%20for%20Information%20Interchange>

Naheliegender wären ja die Zahlen 128 bis 255. Es könnten aber auch die Zahlen -128 bis -1 sein. Um die zugrunde liegenden Zusammenhänge zu erläutern muss etwas weiter ausgeholt werden

Wie auch bei der C++-Programmierung/ Einführung in C++/ Variablen, Konstanten und ihre Datentypen#Ganzzahlen³ (und char gehört schon ein wenig mit dazu), wird zwischen "vorzeichenbehafteten" (englisch: "signed") und "vorzeichenlosen" (englisch: "unsigned") Zahlen unterschieden. Diese Unterscheidung existiert jedoch nur im "Kopf des Programmierers". Der Prozessor weiß nicht, ob der Wert einer Speicherzelle vorzeichenbehaftet ist oder nicht. Zum Addieren und Subtrahieren muss er das auch nicht wissen. Die gewählte "Zweierkomplement-Darstellung" gestattet es, mit vorzeichenbehafteten Zahlen genauso zu rechnen wie mit vorzeichenlosen. Nur beim Multiplizieren und Dividieren müssen Sie dem Prozessor mitteilen, ob er eine normale (vorzeichenlose) oder eine vorzeichenbehaftete Operation ausführen soll.

3 Kapitel 9.1.3 auf Seite 34

Dezimal	Bit-Stellen							
	7	6	5	4	3	2	1	0
...								
252	1	1	1	1	1	1	0	0
253	1	1	1	1	1	1	0	1
254	1	1	1	1	1	1	1	0
255	1	1	1	1	1	1	1	1
256	1	0	0	0	0	0	0	0
257	1	0	0	0	0	0	0	1
258	1	0	0	0	0	0	1	0
259	1	0	0	0	0	0	1	1
...								

Zur besseren Veranschaulichung die beispielhafte Binärdarstellung der Zahlen um 255 herum:

Sie sehen, dass die Zahlen ab 256 nicht mehr in die 8 Bits eines Bytes passen, das Byte "läuft über". Wenn Sie nun z.B. zu 255 eine 1 addieren, und das Ergebnis in einem Byte speichern wollen, wird das oberste Bit abgeschnitten und übrig bleibt eine 0, die dann gespeichert wird.

Da bei dem Übergang von 255 zu 0 die Zahlen wieder von vorne anfangen, können Sie sich den Zahlenbereich auch als Kreis vorstellen. Entgegen dem Uhrzeigersinn (in Pfeilrichtung) "wachsen" die Zahlen, in die andere Richtung fallen sie. An irgendeiner Stelle erfolgt dann ein "Sprung". Wenn diese Stelle bei einer Berechnung überschritten wird, wird dies einen "Überlauf" genannt. Bei "unsigned" Zahlen unterhalb der 0, bei den "signed" Zahlen genau gegenüber:

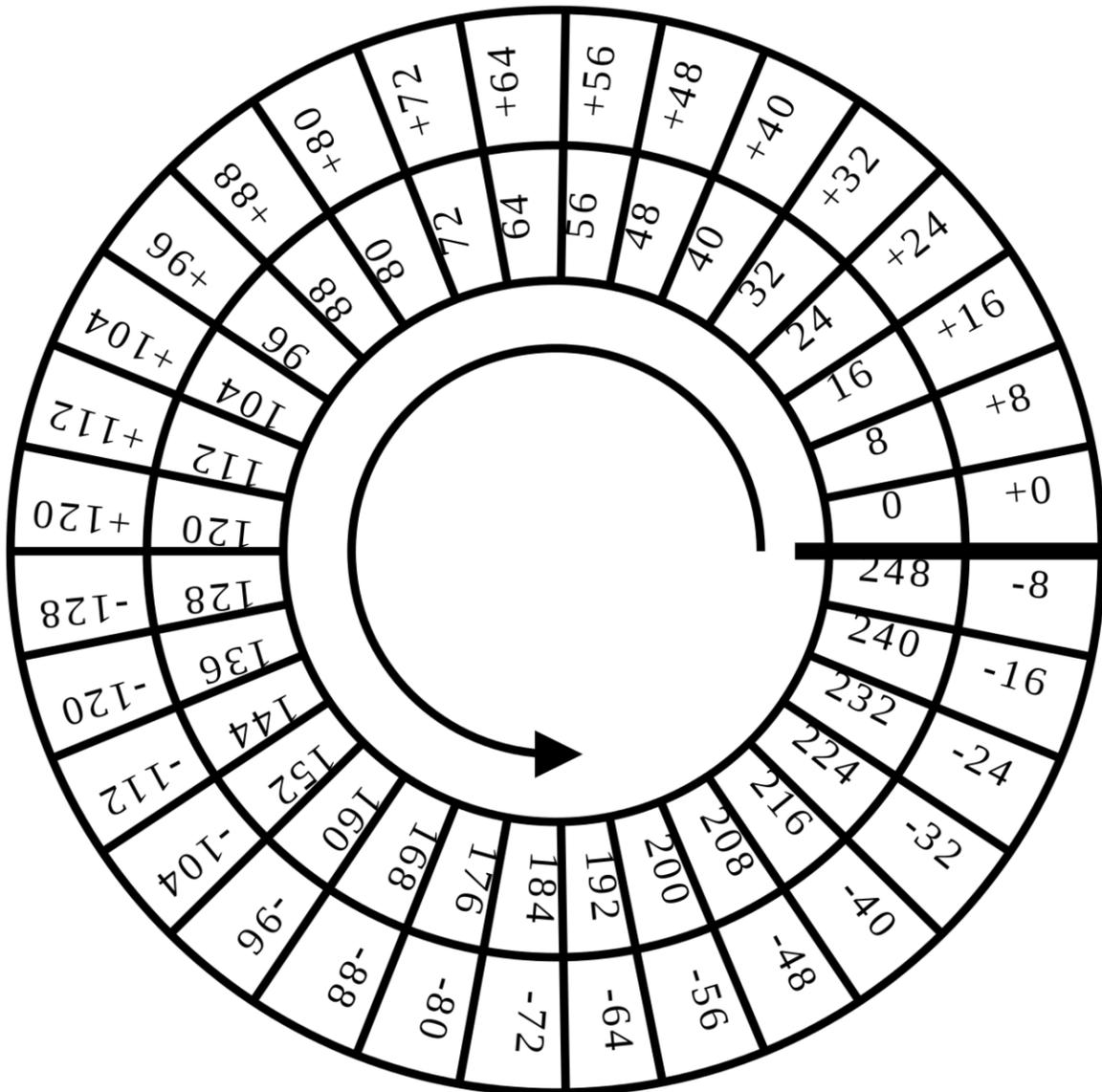


Abb. 6

Diese Darstellung negativer Zahlen hat einige Vorteile:

1. Der Prozessor muss beim Addieren und Subtrahieren nicht darauf achten, ob eine Zahl vom Programmierer als vorzeichenbehaftet oder vorzeichenlos interpretiert wird. Beim Addieren zählt er entsprechend viele Zahlen "links herum", beim Subtrahieren entsprechend andersherum. Falls dabei ein Überlauf auftritt, ist das dem Prozessor "egal".
2. Die "Sprungstelle", also der Überlauf, ist möglichst weit weg von der Null und den kleinen Zahlen, die am meisten benutzt werden, so dass Sie beim Rechnen (hoffentlich) nicht in den Bereich dieses Überlaufes kommen.

Diese Darstellung negativer Zahlen wird auch bei den anderen Ganzzahltypen angewendet, nur ist der Kreis dort wesentlich größer, und enthält mehr darstellbare Zahlen.

Während nun bei den Ganzzahltypen `int`, `short` und `long` festgelegt ist, dass sie standardmäßig immer "signed" sind (so können Sie in C++ auch `signed int` statt `int` usw. schreiben, es ist der gleiche Datentyp), konnte man sich bei `char` nicht einigen, ob mit oder ohne Vorzeichen praktischer ist. Folglich kann jeder Compilerhersteller dies nach eigenem Gusto entscheiden. Ob auf einem System der `char`-Datentyp nun vorzeichenbehaftet ist oder nicht, ist jedoch in der Praxis nicht wichtig, sofern Sie nicht mit ihnen rechnen wollen. (Zum Rechnen werden `char`-Variablen und -Literele stets erst in ein `int` umgewandelt. Erst dann wird z.B. aus einem Byte mit den Bits 11111111 der Wert 255 oder -1, je nach dem.)

Sie können jedoch explizit `signed char` oder `unsigned char` schreiben, wenn Sie sicher gehen wollen, dass Sie ein vorzeichenbehaftetes bzw. vorzeichenloses `char` bekommen. (Kurioserweise ist ein `char` ein eigener Datentyp, auch wenn er stets entweder einem `signed char` oder einem `unsigned char` gleicht. Es gibt also in C++ drei verschiedene `char`-Datentypen!)

Die übrigen Ganzzahltypen belegen mehrere Bytes, wie viele genau, können Sie herausbekommen, in dem Sie ein kleines Programm schreiben, das folgendes ausgibt:

```
std::cout << "Größe eines short: " << sizeof(short) << std::endl;
std::cout << "Größe eines int : " << sizeof(int) << std::endl;
std::cout << "Größe eines long : " << sizeof(long) << std::endl;
```

`sizeof(Typ_oder_Ausdruck)` gibt die Größe eines Typs oder Ausdrucks in Byte zurück. (Genauer: `sizeof()` gibt an, wie viele Bytes der Datentyp im Vergleich zu einem `'char'` benötigt. `sizeof(char)` ist definitionsgemäß gleich 1, da keine kleinere Speichergröße direkt angesprochen werden kann.)

Die Größe der Datentypen ist jedoch nicht genau festgelegt und variiert von Compiler zu Compiler und von Prozessor zu Prozessor! Sie dürfen sich also nicht darauf verlassen, dass ein `int` immer soundsoviel Bytes groß ist! Das einzige, das im C++ Standard festgelegt ist, ist folgende Größenrelation:

```
sizeof(char) := 1
sizeof(long) ≥ sizeof(int) ≥ sizeof(short) ≥ 1
sizeof( X ) = sizeof( signed X ) = sizeof( unsigned X )    (für X =
    int, short, long, char)
```

Wissen Sie jedoch die Größe eines Ganzzahltyps, wüssten Sie auch, wie groß der Wertebereich ist, den er aufnehmen kann:

Größe in Byte	Wertebereich	
		unsigned

1 (8bit)	0 ... 255	-128 ... +127
2 (16bit)	0 ... 65.536	-32.768 ... +32.767
4 (32bit)	0 ... 4.294.967.296	-2.147.483.648 ... +2.147.483.647
8 (64bit)	0 ... 18.446.744.073.709.551.616	-9.223.372.036.854.775.808 ... +9.223.372.036.854.775.807

Warum hat man die Größe der Datentypen nicht festgelegt?

Ganz einfach. Es existieren sehr verschiedene Prozessortypen⁴. Es gibt so genannte 16-Bit-Prozessoren, das heißt, diese können 16-bittige Zahlen "in einem Rutsch" verarbeiten. Heutige PC-Prozessoren sind meist 32-Bit-Prozessoren, sie können 32-Bit-Zahlen besonders schnell verarbeiten. Inzwischen kommen auch mehr und mehr 64-Bit-Prozessoren auf, vielleicht werden sie ja in ein paar Jahren die 32-Bit-Prozessoren verdrängen.

Wäre jetzt festgelegt, dass ein int immer 32-bit groß ist, wären C++ Programme auf 16-Bit-Prozessoren sehr langsam, da sie nur "häppchenweise" mit 32-bit-Werten rechnen können. Darum wurde festgelegt, dass ein int immer so groß sein soll, wie die so genannte "Wortbreite" des Prozessors. Das garantiert, dass Berechnungen mit int's immer am schnellsten ausgeführt werden können.

Mal als Überblick, welche Größen die Ganzzahl-Datentypen auf verschiedenen Plattformen haben:

Plattform	Größe des Datentyps (in Byte)		
	short	int	long
DOS (16 bit "Real Mode")	2	2	4
Linux (auf 32bit-Plattformen wie x86, PowerPC u.a.) OS/2 (ab 3.0) MS-Windows (ab 9x/NT/2k)	2	4	4
Linux (auf 64bit-Plattformen wie Alpha, amd64 u.a.)	2	4	8

Sie sehen, dass unter Linux auch auf einem 64bit-System ein int nur 32 Bit groß ist. Vermutlich wurde sich für 32-bittige int's entschieden, weil es zu viele C- und C++-Programme gibt, die darauf vertrauen, dass ein int immer 32 bit groß ist. :(Es ist aber nicht sicher, dass das auch in Zukunft so bleiben wird.

Wenn Sie low-level Code schreiben wollen/müssen, und auf die genaue Größe seiner Ganzzahltypen angewiesen sind, gibt es spezielle typedef's in dem Standard-Header <stdint.h>, auf welche später genauer eingegangen wird.

Zusammenfassend lässt sich also aussagen, dass ein int mehrere aufeinander folgende Bytes im Speicher belegt. Wie bereits erläutert, werden die Bits in einem Byte von 0 bis 7 gezählt, das Bit 0 ist dabei die "Einer-Stelle", das Bit 7 hat die Wertigkeit 128 (bzw. -128 bei vorzeichenbehafteten Bytes). Bei einem 2-Byte-short int ergeben sich also 16 Bits, und das Ganze sieht dann wie folgt aus:

⁴ <http://de.wikipedia.org/wiki/Prozessor%20%28Hardware%29>

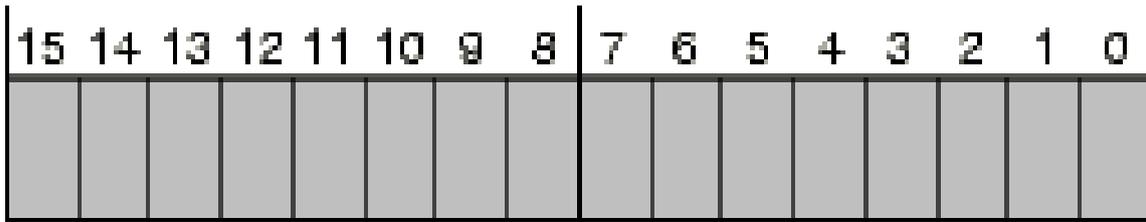


Abb. 7

Das MSB ist in diesem Falle das Bit 15. Die Bits 0..7 bilden das so genannte "Low Byte", die Bits 8..15 das "High Byte".

So weit so gut. Das Problem ist jetzt die Byte-Reihenfolge⁵, also wie dieses "Doppelbyte" abgespeichert wird. Die einen sagen: Low-Byte zuerst, also das "untere Byte" an die "kleinere Adresse", das High-Byte in die darauf folgende, dies wird "Little Endian" genannt. Andere finden es natürlicher, wenn das High-Byte "zuerst" gespeichert wird, also an der kleinere Adresse im Speicher liegt. Dies nennt sich "Big Endian"-Speicherung. Diese beiden Methoden der Speicherung von Mehrbyte-Werten existieren auch bei 32- und 64-Bit-Werten.

Da sich in dieser Frage die verschiedenen Prozessor-Hersteller nicht einig wurden, existieren bis heute beide Varianten. Der eine Prozessor macht es so herum, ein anderer andersherum:

Prozessor-Plattform	"Endianness"
Intel x86 und kompatible	Little Endian
Intel Itanium (64-Bit-CPU)	Little Endian
Motorola 68k	Big Endian
PowerPC	Vom Betriebssystem einstellbar
Sun SPARC	Big Endian
DEC Alpha	Vom Betriebssystem einstellbar

Wichtig wird diese so genannte "Byte order" immer dann, wenn Daten in eine Datei gespeichert oder über ein Netzwerk auf andere Computer geschickt werden sollen. Denn falls die Datei später von einem Computer mit anderer Byte Order gelesen wird, versteht er die gespeicherten Daten nicht mehr richtig. Sie müssen also bei der Speicherung oder Übermittlung von Daten stets vorher festlegen, welche Byte Order man benutzt. Im Internet wurde dafür der Begriff "network byte order" geprägt. Dieser bedeutet: Big Endian.

44.1 Gleitkommazahlen

C++-Programmierung/ Einführung in C++/ Variablen, Konstanten und ihre Datentypen#Gleitkommazahlen⁶ werden heutzutage meist nach dem so genannten IEEE 754 Standard⁷ abgespeichert. Nach diesem Standard ist ein 32bit- (einfache Genauigkeit) und ein 64bit- (doppelte

⁵ <http://de.wikipedia.org/wiki/Byte-Reihenfolge>

⁶ Kapitel 9.1.4 auf Seite 38

⁷ <http://de.wikipedia.org/wiki/IEEE%20754>

Genauigkeit) Gleitkommatyp definiert. Der 32-bit Gleitkommatyp ist dabei wie folgt aufgebaut:

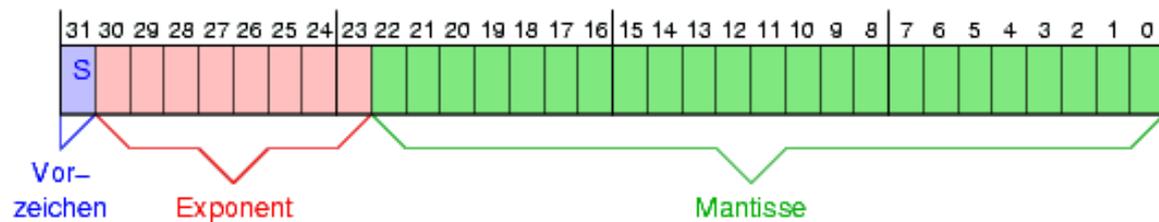


Abb. 8

Das Vorzeichen-Bit gibt dabei an, ob die Zahl positiv (Vorzeichen=0) oder negativ (Vorzeichen=1) ist. In der Mantisse werden nur die Binärstellen nach dem Komma gespeichert. Vor dem Komma steht implizit stets eine 1. Der gespeicherte Wert für den Exponenten ist um 127 größer als der echte Exponent. Die gespeicherten Werte von 0..255 für den Exponenten entsprechen also einem realen Wertebereich von -127...+128. Dabei sind die Werte -127 und +128 "reserviert" für spezielle Gleitkommawerte wie "null" oder "unendlich" (und noch einige andere, so genannte NaN's und 'denormalisierte Zahlen', worauf hier aber nicht weiter eingegangen wird). Der Wert einer 32-bit-Gleitkommazahl errechnet sich dabei wie folgt:

$$\text{Wert} = (-1)^V * 2^{(exp-127)} * 1.\text{mantisse}$$

Der Wert "null" wird dabei mit Exponent 0 und Mantisse 0 gespeichert. Dabei kann das Vorzeichen 0 oder 1 sein, es wird also zwischen +0.0 und -0.0 unterschieden. Damit kann man z.B. erkennen, ob das Ergebnis einer Berechnung ein "sehr kleiner positiver Wert" war, der auf 0 gerundet wurde, oder ein "sehr kleiner negativer Wert". Ansonsten sind beide Werte identisch. Ein Vergleich (+0.0 == -0.0) ergibt also den Wert "true", also "wahr".

Der Wert "unendlich" wird mit Exponent 255 und Mantisse 0 gespeichert. "Unendlich" bedeutet dabei "größer als die größte darzustellende Zahl". Auch hier wird wieder zwischen "+unendlich" und "-unendlich" unterschieden. "Unendlich" kann dabei durchaus sinnvoll sein. So liefert die mathematische Funktion atan() (arcus tangens) für "unendlich" genau den Wert Pi/2.

45 Autoren

Edits	User
1	5gon12eder ¹
1	Axelf ²
1	Axum ³
1	Banco ⁴
12	Bernina ⁵
15	Bulldog98 ⁶
4	Ce2 ⁷
41	Chfreund ⁸
2	Cilania ⁹
1	Daniel B ¹⁰
1	Daniel L.F. ¹¹
21	Dirk Huenniger ¹²
17	E^(nix) ¹³
2	Einfach Toll ¹⁴
1	EphemeralJun ¹⁵
6	Etlam ¹⁶
19	Extendable ¹⁷
43	Fencer ¹⁸
1	Fleshgrinder ¹⁹
24	Gate-Man ²⁰
3	Gorlingor ²¹

1 <http://de.wikibooks.org/w/index.php?title=Benutzer:5gon12eder>
2 <http://de.wikibooks.org/w/index.php?title=Benutzer:Axelf>
3 <http://de.wikibooks.org/w/index.php?title=Benutzer:Axum>
4 <http://de.wikibooks.org/w/index.php?title=Benutzer:Banco>
5 <http://de.wikibooks.org/w/index.php?title=Benutzer:Bernina>
6 <http://de.wikibooks.org/w/index.php?title=Benutzer:Bulldog98>
7 <http://de.wikibooks.org/w/index.php?title=Benutzer:Ce2>
8 <http://de.wikibooks.org/w/index.php?title=Benutzer:Chfreund>
9 <http://de.wikibooks.org/w/index.php?title=Benutzer:Cilania>
10 http://de.wikibooks.org/w/index.php?title=Benutzer:Daniel_B
11 http://de.wikibooks.org/w/index.php?title=Benutzer:Daniel_L.F.
12 http://de.wikibooks.org/w/index.php?title=Benutzer:Dirk_Huenniger
13 <http://de.wikibooks.org/w/index.php?title=Benutzer:E%5E%28nix%29>
14 http://de.wikibooks.org/w/index.php?title=Benutzer:Einfach_Toll
15 <http://de.wikibooks.org/w/index.php?title=Benutzer:EphemeralJun>
16 <http://de.wikibooks.org/w/index.php?title=Benutzer:Etlam>
17 <http://de.wikibooks.org/w/index.php?title=Benutzer:Extendable>
18 <http://de.wikibooks.org/w/index.php?title=Benutzer:Fencer>
19 <http://de.wikibooks.org/w/index.php?title=Benutzer:Fleshgrinder>
20 <http://de.wikibooks.org/w/index.php?title=Benutzer:Gate-Man>
21 <http://de.wikibooks.org/w/index.php?title=Benutzer:Gorlingor>

- 7 Gronau²²
- 2 Heuler06²³
- 1 Hjn²⁴
- 1 InselAX²⁵
- 1 Jan Luca²⁶
- 1 Jarling²⁷
- 2 Juetho²⁸
- 1 Jörg Klein²⁹
- 1 K@rl³⁰
- 13 Klaus Eifert³¹
- 1 KornroK³²
- 2 Krje³³
- 1 Lars Andresen³⁴
- 4 Linopolus³⁵
- 1 Lordroran³⁶
- 2 Marlon93³⁷
- 10 Martin Fuchs³⁸
- 1 Matze³⁹
- 8 MetalHeart⁴⁰
- 1 MichaelFrey⁴¹
- 5 MichaelFreyTool⁴²
- 2 Mr. Anderson⁴³
- 2 Muxxxa⁴⁴
- 2 Norro⁴⁵
- 2 O-Dog⁴⁶

-
- 22 <http://de.wikibooks.org/w/index.php?title=Benutzer:Gronau>
 - 23 <http://de.wikibooks.org/w/index.php?title=Benutzer:Heuler06>
 - 24 <http://de.wikibooks.org/w/index.php?title=Benutzer:Hjn>
 - 25 <http://de.wikibooks.org/w/index.php?title=Benutzer:InselAX>
 - 26 http://de.wikibooks.org/w/index.php?title=Benutzer:Jan_Luca
 - 27 <http://de.wikibooks.org/w/index.php?title=Benutzer:Jarling>
 - 28 <http://de.wikibooks.org/w/index.php?title=Benutzer:Juetho>
 - 29 http://de.wikibooks.org/w/index.php?title=Benutzer:J%C3%B6rg_Klein
 - 30 <http://de.wikibooks.org/w/index.php?title=Benutzer:K%40rl>
 - 31 http://de.wikibooks.org/w/index.php?title=Benutzer:Klaus_Eifert
 - 32 <http://de.wikibooks.org/w/index.php?title=Benutzer:KornroK>
 - 33 <http://de.wikibooks.org/w/index.php?title=Benutzer:Krje>
 - 34 http://de.wikibooks.org/w/index.php?title=Benutzer:Lars_Andresen
 - 35 <http://de.wikibooks.org/w/index.php?title=Benutzer:Linopolus>
 - 36 <http://de.wikibooks.org/w/index.php?title=Benutzer:Lordroran>
 - 37 <http://de.wikibooks.org/w/index.php?title=Benutzer:Marlon93>
 - 38 http://de.wikibooks.org/w/index.php?title=Benutzer:Martin_Fuchs
 - 39 <http://de.wikibooks.org/w/index.php?title=Benutzer:Matze>
 - 40 <http://de.wikibooks.org/w/index.php?title=Benutzer:MetalHeart>
 - 41 <http://de.wikibooks.org/w/index.php?title=Benutzer:MichaelFrey>
 - 42 <http://de.wikibooks.org/w/index.php?title=Benutzer:MichaelFreyTool>
 - 43 http://de.wikibooks.org/w/index.php?title=Benutzer:Mr._Anderson
 - 44 <http://de.wikibooks.org/w/index.php?title=Benutzer:Muxxxa>
 - 45 <http://de.wikibooks.org/w/index.php?title=Benutzer:Norro>
 - 46 <http://de.wikibooks.org/w/index.php?title=Benutzer:O-Dog>

2	Onegin ⁴⁷
2	Phbarnacle ⁴⁸
1	Phil239 ⁴⁹
5	Philipp K. ⁵⁰
533	Prog ⁵¹
6	Pumbaa80 ⁵²
24	Quiethoo ⁵³
1	Remohe ⁵⁴
34	RokerHRO ⁵⁵
13	Ronsn ⁵⁶
1	Rothose86 ⁵⁷
26	Sae1962 ⁵⁸
1	Sannaj ⁵⁹
2	Se4598 ⁶⁰
1	Sentropie ⁶¹
3	Sesshu1 ⁶²
1	Shadak ⁶³
1	Skriptkid ⁶⁴
9	Sliver ⁶⁵
1	Steef389 ⁶⁶
11	Stefan Kögl ⁶⁷
2	Szs ⁶⁸
10	ThePacker ⁶⁹
1	Therapiekind ⁷⁰
1	Thomas J ⁷¹

47	http://de.wikibooks.org/w/index.php?title=Benutzer:Onegin
48	http://de.wikibooks.org/w/index.php?title=Benutzer:Phbarnacle
49	http://de.wikibooks.org/w/index.php?title=Benutzer:Phil239
50	http://de.wikibooks.org/w/index.php?title=Benutzer:Philipp_K.
51	http://de.wikibooks.org/w/index.php?title=Benutzer:Prog
52	http://de.wikibooks.org/w/index.php?title=Benutzer:Pumbaa80
53	http://de.wikibooks.org/w/index.php?title=Benutzer:Quiethoo
54	http://de.wikibooks.org/w/index.php?title=Benutzer:Remohe
55	http://de.wikibooks.org/w/index.php?title=Benutzer:RokerHRO
56	http://de.wikibooks.org/w/index.php?title=Benutzer:Ronsn
57	http://de.wikibooks.org/w/index.php?title=Benutzer:Rothose86
58	http://de.wikibooks.org/w/index.php?title=Benutzer:Sae1962
59	http://de.wikibooks.org/w/index.php?title=Benutzer:Sannaj
60	http://de.wikibooks.org/w/index.php?title=Benutzer:Se4598
61	http://de.wikibooks.org/w/index.php?title=Benutzer:Sentropie
62	http://de.wikibooks.org/w/index.php?title=Benutzer:Sesshu1
63	http://de.wikibooks.org/w/index.php?title=Benutzer:Shadak
64	http://de.wikibooks.org/w/index.php?title=Benutzer:Skriptkid
65	http://de.wikibooks.org/w/index.php?title=Benutzer:Sliver
66	http://de.wikibooks.org/w/index.php?title=Benutzer:Steef389
67	http://de.wikibooks.org/w/index.php?title=Benutzer:Stefan_K%C3%B6gl
68	http://de.wikibooks.org/w/index.php?title=Benutzer:Szs
69	http://de.wikibooks.org/w/index.php?title=Benutzer:ThePacker
70	http://de.wikibooks.org/w/index.php?title=Benutzer:Therapiekind
71	http://de.wikibooks.org/w/index.php?title=Benutzer:Thomas_J

- 1 Tschäfer⁷²
- 2 W4R⁷³
- 2 WikiBookPhil⁷⁴
- 2 Yeto⁷⁵

⁷² <http://de.wikibooks.org/w/index.php?title=Benutzer:Tsch%C3%A4fer>

⁷³ <http://de.wikibooks.org/w/index.php?title=Benutzer:W4R>

⁷⁴ <http://de.wikibooks.org/w/index.php?title=Benutzer:WikiBookPhil>

⁷⁵ <http://de.wikibooks.org/w/index.php?title=Benutzer:Yeto>

Abbildungsverzeichnis

- GFDL: Gnu Free Documentation License. <http://www.gnu.org/licenses/fdl.html>
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. <http://creativecommons.org/licenses/by-sa/3.0/>
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. <http://creativecommons.org/licenses/by-sa/2.5/>
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. <http://creativecommons.org/licenses/by-sa/2.0/>
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. <http://creativecommons.org/licenses/by-sa/1.0/>
- cc-by-2.0: Creative Commons Attribution 2.0 License. <http://creativecommons.org/licenses/by/2.0/>
- cc-by-2.0: Creative Commons Attribution 2.0 License. <http://creativecommons.org/licenses/by/2.0/deed.en>
- cc-by-2.5: Creative Commons Attribution 2.5 License. <http://creativecommons.org/licenses/by/2.5/deed.en>
- cc-by-3.0: Creative Commons Attribution 3.0 License. <http://creativecommons.org/licenses/by/3.0/deed.en>
- GPL: GNU General Public License. <http://www.gnu.org/licenses/gpl-2.0.txt>
- LGPL: GNU Lesser General Public License. <http://www.gnu.org/licenses/lgpl.html>
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- LFK: Lizenz Freie Kunst. <http://artlibre.org/licence/lal/de>
- CFR: Copyright free use.

- EPL: Eclipse Public License. <http://www.eclipse.org/org/documents/epl-v10.php>

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses⁷⁶. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrowser.

⁷⁶ Kapitel 46 auf Seite 271

1	Chfreund ⁷⁷	PD
2		PD
3	Prog	cc-by-sa-3.0
4		PD
5		PD
6		PD
7		PD
8		PD

⁷⁷ <http://de.wikibooks.org/wiki/User%3AChfreund>

46 Licenses

46.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <<http://fsf.org/>>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is restricted constantly by software patents. States should not allow patents to threaten development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents can not be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS 0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each license is addressed as "you", "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion. 1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run

the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work. 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable and exclusive; the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary. 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures. 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee. 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

* a) The work must carry prominent notices stating that you modified it, and giving a relevant date. * b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices". * c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. * d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume or a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate. 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

* a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. * b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. * c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you convey the object code with such an offer, in accord with subsection 6b. * d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. * e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects to use, or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying. 7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

* a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or * b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or * c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or * d) Limiting the use of that material for publicity purposes of names of licensors or authors of the material; or * e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or * f) Requiring indemnification of licensors and authors of that material; by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way. 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants

you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so. 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it. 11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by or as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deposit yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law. 12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program. 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of

acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THIS IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PRO-

46.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <<http://fsf.org/>>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a section entitled by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A Secondary Section is named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The Invariant Sections are certain Secondary Sections whose titles are designated in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, presented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text nearest the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section Entitled XYZ means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that

GRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

translates XYZ into another language. (Here XYZ stands for a specific section name mentioned below, such as Acknowledgements, "Dedications", Endorsements, or "History"). To "Preserve the Title" of a section XYZ when you modify the Document means that it remains a section Entitled XYZ according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties; any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. * B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. * C. State on the Title Page the name of the publisher of the Modified Version, as the publisher. * D. Preserve all the copyright notices of the Document. * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. * F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. * H. Include an unaltered copy of this License. * I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous section. * J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions if they were based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or for the original publisher of the version it refers to gives permission. * K. For any section Entitled Acknowledgements or "Dedications", Preserve the Title of the section, and preserve in the section

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file but most effectively state the extension of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <<http://www.gnu.org/licenses/>>.

Also add information on how to contact you by electronic and paper mail.

all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. * L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. * M. Delete any section Entitled Endorsements. Such a section may not be included in the Modified Version. * N. Do not retile any existing section to be Entitled Endorsements to conflict in title with any Invariant Section. * O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled Endorsements, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts. Invariant Sections may be replaced with a single copy. If there are one or more Back-Cover Texts, they may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled Acknowledgements, and any sections Entitled "Dedications". You must delete all sections Entitled Endorsements. 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an aggregate if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <<http://www.gnu.org/licenses/>>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <<http://www.gnu.org/philosophy/why-not-lgpl.html>>.

If a section in the Document is Entitled Acknowledgements, "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License or any later version applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document. 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

Incorporate means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is eligible for relicensing if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing. ADDENDUM: How to use this License for your documents

To use this license in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation, with Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

46.3 GNU Lesser General Public License

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <<http://fsf.org/>>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below. 0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

The “Corresponding Application Code” for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

* a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or * b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under

terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

* a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work, and reverse engineering for debugging such modifications, if you also do each of the following:

* a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the Combined Work with a copy of the GNU GPL and this license document. * c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. * d) Do one of the following: o 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and

under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. o 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version. * e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

* a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. * b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.