

C++

*Begleitheft zum Kurs für Mathematisch-technische
Assistentinnen und Assistenten*

Wilhelm Hanrath
RWTH Aachen

Quelle:

<http://www-ma2.rz.rwth-aachen.de/material/hanrath/cplpl/cplpl.ps>

Hinweis: der Text sieht am Bildschirm ein bisschen unscharf aus, aber im Ausdruck ergibt sich eine hervorragende Druckqualität. Also besser ausdrucken.

Aufbereitet als PDF-Datei für die



<http://www.c-plusplus.de>

C++

Begleitheft zum Kurs
für
Mathematisch-technische
Assistentinnen und Assistenten



Abteilung Numerik und MATA-Ausbildung

Inhaltsverzeichnis

Vorwort	ix
1 Einleitung	1
1.1 Begriffe	1
1.2 Geschichte der objektorientierten Programmierung	2
1.3 Geschichte von C++	4
I Einführung	5
2 C++ als Erweiterung von C	7
2.1 Formales Aussehen von C++-Programmen	7
2.1.1 Benutzen von “altem“ C-Code	9
2.2 Kommentare in C++-Programmen	10
2.3 Typen	10
2.3.1 Standardtypen	10
2.3.2 Benutzerdefinierte Typen	11
2.3.3 Typumwandlungen	12
2.3.4 Deklarationen von Variablen	13
2.4 Der Gültigkeitsbereichsauflösungsoperator ::	14
2.5 Namensbereiche	15
2.5.1 Grundlagen zu Namensbereichen	15
2.5.2 <code>using</code> -Deklarationen bzw. die <code>using</code> -Direktiven	17
2.5.3 Namespaces und Funktionsargumente	19
2.5.4 Geschachtelte Namensbereiche	20
2.5.5 Unbenannte Namensbereiche	21
2.5.6 Weitere Techniken zu Namensbereichen	21
2.6 Referenzen	22
2.6.1 Referenzen als Funktionsparameter	23
2.6.2 Referenzen als Funktionsergebnisse	24
2.7 Konstante Datenobjekte	26
2.7.1 Zeiger und <code>const</code>	28
2.7.2 Referenzen und <code>const</code>	30
2.7.3 Funktionsparameter und <code>const</code>	30
2.7.4 Funktionsergebnisse und <code>const</code>	34
2.7.5 Typumwandlung mittels <code>const_cast<typ></code>	37
2.8 Standardparameter von Funktionen	38

2.9	<code>inline</code> -Funktionen	40
2.10	Überladen von Funktionen	43
2.10.1	Typumwandlung bei überladenen Funktionen	45
2.10.2	Überladen und Gültigkeitsbereich	47
2.11	Ausnahmebehandlung	47
2.11.1	Geschachtelte <code>try</code> -Blöcke	50
2.11.2	<code>catch</code> (. . .)	50
2.11.3	Ausnahmen "weiterreichen"	50
2.11.4	Unterscheidung von Ausnahmen	51
2.11.5	Nicht abgefangene Ausnahmen	52
2.11.6	Ausnahmen in der Schnittstelle einer Funktion	53
2.12	Die neue Verwaltung des Freispeichers	55
2.12.1	Unerfüllbare Speicheranforderungen	57
2.12.2	Platzieren von Objekten	60
3	Einblick in die Standardbibliothek	63
3.1	Aus-/Eingabe	63
3.1.1	Datentypen zur Ein- und Ausgabe	63
3.1.2	Der Ausgabeoperator <code><<</code>	64
3.1.3	Der Eingabeoperator <code><<</code>	65
3.1.4	Manipulatoren	67
3.1.5	Fehlerzustände in Strömen	68
3.1.6	Dateibehandlung	69
3.2	Die Standard-String-Klasse	70
3.2.1	Erzeugung eines Strings	71
3.2.2	Ein-/Ausgabe von Strings	71
3.2.3	Zugriff auf einzelne Zeichen eines Strings	72
3.2.4	Vergleich von Strings	72
3.2.5	Verkettung von Strings	73
3.2.6	Zuweisen an einen String	74
II	Objektorientierte Techniken	77
4	Klassen	79
4.1	Grundlagen	79
4.1.1	Programmierparadigmen	79
4.2	Klassen/Objekte	86
4.2.1	Grundlagen	86
4.2.2	Zugriffsschutz	86
4.2.3	Konstruktoren	88
4.2.4	Implementierungsmöglichkeiten	89
4.2.5	Klassen als Softwarebausteine	91
4.2.6	Destruktoren	94
4.2.7	Member-Funktionen	97
4.2.8	Der <code>this</code> -Zeiger	98
4.2.9	Konstante Member-Funktionen	99

4.2.10	Verwenden von Klassen	103
4.3	Konstruktoren im Detail	106
4.3.1	Standard-Parameterloser-Konstruktor	107
4.3.2	Standard-Copy-Konstruktor	108
4.3.3	Selbstgeschriebene Konstruktoren	109
4.3.4	Initialisierungslisten	111
4.3.5	Ausnahmen in Konstruktoren	113
4.3.6	Copy-Konstruktor und dynamische Komponenten	115
4.3.7	Konstruktoren und Typumwandlung	118
4.3.8	<code>new</code> bzw. <code>new[]</code> und parameterbehaftete Konstruktoren	119
4.3.9	Adressen von Konstruktoren	120
4.4	Destruktoren im Detail	120
4.5	Ressourcenmanagement	124
4.6	Statische Klassenkomponenten	125
4.6.1	Statische Member-Daten	125
4.6.2	Statische Member-Funktionen	127
4.7	Konstanten oder Referenzen als Komponenten	128
4.7.1	Klassenkonstanten	128
4.8	Komponentenzeiger	129
4.8.1	Zeiger auf Datenkomponenten	130
4.8.2	Zeiger auf Funktionskomponenten	130
4.8.3	Komponentenzeiger und <code>void *</code>	132
4.9	Befreundete Klassen und Funktionen	132
5	Operatoren	135
5.1	Übersicht über die Operatoren	135
5.2	Operatorüberladung	139
5.2.1	Standard-Operatoren für Klassen	140
5.2.2	Grundlagen der Operatorüberladung	142
5.2.3	Operatorüberladung als globale Funktion	143
5.2.4	Operatorüberladung als Member-Funktion	147
5.2.5	Zusammenfassung: Operatorüberladung	150
5.2.6	Keine Defaultparameter für Operatorfunktionen	151
5.2.7	Operatorüberladung bei großen Typen	151
5.2.8	Besonderheiten spezieller Operatoren	153
5.3	Übersicht: Überladungsmöglichkeiten	164
5.4	Konvertierungs-Operatoren	166
5.5	Operatorüberladung in der Standardbibliothek	168
5.6	Konversionen, Konstruktoren und Überladung	169
6	Templates	171
6.1	Template-Funktionen	171
6.1.1	Grundlagen zu Template-Funktionen	171
6.1.2	Typumwandlungen und Template-Funktionen	173
6.1.3	Explizite Instanziierung	174
6.1.4	Template-Parameter	175
6.1.5	Überladen und Spezialisierung von Funktions-Templates	178

6.1.6	Regeln zum Auffinden der “richtigen“ Funktion	179
6.2	Template-Klassen	180
6.2.1	Grundlagen zu Template-Klassen	180
6.2.2	Template-Parameter	182
6.2.3	Spezialisierung von Klassen-Templates	184
6.2.4	Element-Templates	185
6.2.5	Templates und Vererbung	187
6.3	Implementierungsmöglichkeiten	187
7	Vererbung	189
7.1	Grundlagen	189
7.1.1	Einfache Vererbung	190
7.2	Zugriffsschutz	191
7.2.1	<code>public</code> -Vererbung	193
7.2.2	<code>protected</code> -Vererbung	195
7.2.3	<code>private</code> -Vererbung	196
7.2.4	Anwenderschnittstelle und Vererbungsschnittstelle	197
7.2.5	<code>using</code> -Deklaration einzelner Komponenten	198
7.3	Beispiel für (einfache) Vererbung	199
7.3.1	1. Versuch der Ableitung	201
7.3.2	2. Versuch der Ableitung	203
7.3.3	Probleme bei der Ableitung	207
7.3.4	3. Versuch der Ableitung	208
7.4	Neudefinition, Überladung, virtuelle Funktionen	212
7.4.1	Neudefinition mit unterschiedlicher Signatur	213
7.4.2	Neudefinition mit gleicher Signatur	214
7.4.3	Aufruf virtueller Funktionen	215
7.4.4	Virtuelle Funktionen und Defaultargumente	216
7.4.5	Virtuelle Destruktoren	216
7.4.6	Zuweisung virtuell?	218
7.5	Polymorphie	220
7.5.1	Beispiel für die Verwendung der Polymorphie	220
7.5.2	Laufzeittypinformation	224
7.5.3	<code>typeid</code>	226
7.6	Template-Klassen und Vererbung	227
7.6.1	Template von “normaler“ Klasse ableiten	228
7.6.2	Template von Template ableiten	229
7.6.3	“Normale“ Klassen von Template-Klassen ableiten	231
7.7	Konstruktoren und Vererbung	231
7.8	Destruktoren und Vererbung	233
7.9	Vererbung und Klassen mit dynamischen Komponenten	233
7.10	Ausnahmen und Vererbung	234
7.11	Rein virtuelle Funktionen, abstrakte Basisklassen	237
7.12	Mehrfachvererbung	240
7.12.1	Beispiel für Mehrfachvererbung	241
7.12.2	Namenskonflikte	242
7.12.3	Mehrfache Basisklassen	243

7.12.4	Virtuelle Basisklassen	247
7.12.5	Schlussbemerkungen zu mehrfachen und virtuellen Basisklassen	251
7.13	Navigieren in Klassenhierarchien	255
7.14	Abstrakte Dienste	258

III Die Standardbibliothek 265

8	Ein- Ausgabe	267
8.1	Der Template-Typ <code>char_traits<T></code>	267
8.2	Hintergrund zur Ein- Ausgabe in C++	270
8.3	Fähigkeiten unserer Compiler	274
8.4	Ausgabe	275
8.4.1	Einfache Ausgabe	275
8.4.2	Pufferung der Ausgabe, Manipulatoren	276
8.4.3	Formatierung der Ausgabe	277
8.5	Eingabe	285
8.5.1	Elementfunktionen zur Eingabe	285
8.5.2	Formatierung der Eingabe	286
8.6	Fehlerzustände von Strömen	287
8.6.1	Beispiel zum Einlesen eines selbstdefinierten Datentypes . . .	290
8.7	Manipulatoren	293
8.7.1	Manipulatoren ohne Argumente	293
8.7.2	Manipulatoren mit einem Argument	294
8.8	Dateibehandlung	301
8.8.1	Öffnen/Schließen über Konstruktor/Destruktor	301
8.8.2	Öffnen/Schließen über Elementfunktionen	303
8.8.3	Verwenden geöffneter Dateien	304
8.8.4	Dateipositionierung	305
8.9	String-Streams	307
8.10	Sonstiges	307
8.10.1	Verbinden eines Eingabestroms mit einem Ausgabestrom . . .	307
8.10.2	Ströme und Ausnahmen	307
9	Ausnahmen in der Standardbibliothek	309
9.1	Die Headerdatei <code>exception</code>	309
9.2	Die Headerdatei <code>stdexcept</code>	311
9.2.1	<code>logic_error</code> und abgeleitete Fehlerklassen	311
9.2.2	<code>runtime_error</code> und abgeleitete Fehlerklassen	312
9.3	Sonstige Standard-Fehlerklassen	312
9.3.1	Die Fehlerklasse <code>bad_alloc</code>	312
9.3.2	Die Fehlerklassen <code>bad_typeid</code> und <code>bad_cast</code>	313
9.3.3	Die Fehlerklasse <code>ios_base::failure</code>	314
9.4	Übersicht über die Fehlerklassen der Standardbibliothek	315
9.5	Fehlerklassen verwenden	315

10 C++-Strings	317
10.1 String-Iteratoren	318
10.2 Größe und Kapazität eines Strings	321
10.3 Erzeugen/Zerstören von C++-Strings	323
10.4 Zugriff auf einzelne Zeichen eines Strings	325
10.5 Zuweisungen an einen String	326
10.6 C++-Strings wie C-Strings verwenden	328
10.7 Vergleiche von Strings	329
10.7.1 String-Member-Funktionen <code>compare</code>	329
10.7.2 Globale Operator-Funktionen zum Vergleich	330
10.8 Einfügen, Anhängen, Verketteten	331
10.8.1 Einfügen in einen String	331
10.8.2 Anhängen an einen String	333
10.8.3 Strings verketteten	334
10.9 Teilstrings	335
10.9.1 Auf Teilstrings zugreifen	335
10.9.2 Teilstrings löschen	335
10.9.3 Einen Teilstring ersetzen	336
10.10 Suchen in Strings	338
10.10.1 Suchen eines Teilstrings in einem String	338
10.10.2 Suchen nach einzelnen Zeichen	339
10.11 Ein- und Ausgabe von C++-Strings	341
10.12 Vertauschen von Strings	342
10.13 String-Streams	342
11 Die Standard-Template-Library (STL)	345
11.1 Universelle Hilfsmittel der Standardbibliothek	345
11.1.1 Vergleichsoperatoren	345
11.1.2 Die Template-Klasse <code>pair<></code>	346
11.2 Iteratoren	347
11.2.1 Iterator-Kategorien	348
11.2.2 Iterator-Traits	350
11.2.3 Funktionen, welche von der Iterator-Kategorie abhängen	352
11.2.4 Hilfsfunktionen für Iteratoren	353
11.2.5 Iterator-Adapter	354
11.2.6 Stream-Iteratoren	357
11.3 Standardcontainer	359
11.3.1 Gemeinsamkeiten aller Containerklassen	360
11.3.2 Die Containerklasse <code>vector<T></code>	368
11.3.3 Die Spezialisierung <code>vector<bool></code>	378
11.3.4 Die Containerklasse <code>deque<T></code>	379
11.3.5 Die Containerklasse <code>list<T></code>	387
11.3.6 Die Containerklassen <code>set<T></code> und <code>multiset<T></code>	400
11.3.7 Die Containerklassen <code>map<Key,T></code> und <code>multimap<Key,T></code>	413
11.4 Container-Adapter	427
11.4.1 Der Containeradapter <code>queue<T></code>	427
11.4.2 Der Containeradapter <code>priority_queue<T></code>	432

11.4.3	Der Containeradapter <code>stack<T></code>	436
11.5	Bitsets	440
11.5.1	Konstruktion und Größe eines Bitset	440
11.5.2	Operationen für Bitset	442
11.5.3	Der Referenz-Hilfstyp für Bitsets	446
11.6	Funktionsobjekte	448
11.6.1	Basisklassen zu Standard-Funktionsobjekten	450
11.6.2	Standard-Operatoren als Funktionsobjekte	450
11.6.3	Prädikate	451
11.6.4	Binder, Funktionsadapter, Negierer	451
11.7	Algorithmen	456
11.7.1	Übersicht über die Algorithmen der Standardbibliothek	458
11.7.2	Gemeinsame Bezeichnung für alle Algorithmen	461
11.7.3	Nichtmodifizierende Algorithmen für Sequenzen	462
11.7.4	Modifizierende Algorithmen für Sequenzen	470
11.7.5	Algorithmen und Sortierung	481
11.7.6	Algorithmen für Mengen	487
11.7.7	Algorithmen für Heaps	492
11.7.8	Minimum, Maximum und Vergleich	493
11.8	Die numerische Bibliothek	495
11.8.1	Numerische Standardfunktionen	495
11.8.2	Komplexe Zahlen	496
11.8.3	Mathematische Vektoren	499
11.8.4	Slices zu einem <code>valarray</code>	505
11.8.5	Verallgemeinerte Slices	509
11.8.6	Masken für <code>valarray</code> 's	512
11.8.7	Indirekte <code>valarray</code> 's	512
11.8.8	Numerische Algorithmen für Sequenzen	513
IV	Anhang	517
	Literaturverzeichnis	519
	Index	520

Vorwort

Dies ist (wird) die Ausarbeitung des C++-Kurses, den ich seit einigen Jahren im Rechenzentrum der RWTH-Aachen im Rahmen der Ausbildung Mathematisch-technischer Assistentinnen und Assistenten halte.

Zu dem Zeitpunkt, wo unsere Auszubildenden an diesem Kurs teilnehmen, haben sie bereits die Programmiersprache C kennengelernt, so dass für diesen Kurs C-Kenntnisse vorausgesetzt werden können.

Dem Kursinhalt liegt weitgehend der C++-ANSI-Standard zugrunde, wenn auch der im Praktikum verwendete Compiler (`gcc-2.95.2`) noch nicht alle Sprachmittel des Standards beherrscht.

Aachen, im Sommer 2001

Wilhelm Hanrath

Kapitel 1

Einleitung

1.1 Begriffe

Der Begriff *Objektorientierung* (abgekürzt: *OO*) ist seit einigen Jahren sehr populär. Man muss unterscheiden zwischen

- Objektorientierter Programmierung (abgekürzt: *OOP*), das sind Programmier-techniken und –mittel, mit denen man *objektorientierte Programme* schreiben kann.
- Objektorientierter Analysen (abgekürzt: *OOA*), das ist eine auf Objektorientierung zugeschnittene Art der Problemanalyse.
- Objektorientiertem Design (abgekürzt: *OOD*), hierunter versteht man Entwurfs- und Modellierungsmethoden, die bei der Erstellung von objektorientierten Lösungen zu einer Problemstellung hilfreich sein können.

Bei der Erstellung einer objektorientierten Softwarelösung für eine “normale“ oder “größere“ Problemstellung werden objektorientierte Analysemethoden (OOA), objektorientierte Entwurfsmethoden (OOD) und objektorientierte Programmierung (OOP) eingesetzt und die verwendeten Methoden sollten aufeinander abgestimmt sein.

Nach einhelliger Meinung sind nicht Programmiersprachen an sich objektorientiert, sondern allenfalls die Art und Weise, in dieser Programmiersprache zu programmieren. Programmiersprachen unterscheiden sich darin, in wie weit sie Sprachmittel zur Verfügung stellen, mit denen ein objektorientierter Programmieransatz möglich ist.

Es gibt

- Programmiersprachen, die objektorientierte Programmierung nicht unterstützen,
- andere Programmiersprachen, welche Sprachkonstrukte zur Verfügung stellen, mit denen man (auch) objektorientiert programmieren kann, aber nicht muss (*hybride objektorientierte Sprachen*),
- wiederum andere Sprachen, in denen man quasi gezwungen ist, objektorientiert zu programmieren (*puristische objektorientierte Sprachen*).

Hauptgegenstand dieses Kurses ist objektorientierte Programmierung in C++, objektorientierte Problemanalyse und objektorientierter objektorientierter Programm-entwurf (OOA/OOD) wird in diesem Kurs nicht behandelt (hierzu gibt es spezielle Literatur, etwa [Boo 94], und Methoden, etwa **UML** — Unified Modelling Language).

1.2 Geschichte der objektorientierten Programmierung

Der Ursprung der objektorientierten Programmierung lag in der Entwicklung der Sprache SIMULA (~ 1965 – 1967).

Zunächst setzten sich die etwa zur gleichen Zeit entwickelten *prozeduralen Programmiersprachen* (etwa ALGOL 68, FORTRAN, ...) durch.

Bei prozeduraler Programmierung stehen möglichst optimale *Algorithmen* im Vordergrund, die als Funktionen/Prozeduren in der jeweiligen Sprache realisiert werden. Daten werden (auf unterschiedlich mögliche Weise) von Funktion zu Funktion weitergeleitet und weiterverarbeitet.

Mittels *strukturierter Programmierung, Analyse und Entwicklungsmethoden* war (und ist) es möglich, qualitativ hochwertige Software zu erstellen, für welche die damals verfügbare Hardware leistungsfähig genug war. (Für objektorientierte Programmierung reichte diese Leistungsfähigkeit noch nicht aus!)

Steigende **Qualitätsanforderungen** an Software in Bezug auf:

- Korrektheit:
In wie weit erfüllt die Software die in der Spezifikation festgelegten Anforderungen?
- Robustheit:
Wie reagiert die Software auf abnormale Bedingungen (etwa: falsche Eingabe)?
- Effektivität:
Wie groß sind Anforderungen an Hardware-Ressourcen (wie Prozessor-Zeit, Arbeitsspeicher, externer Speicher, Netzwerklast)?
- Benutzerfreundlichkeit:
Wie schnell kann die Bedienung der Software von Anwendern mit unterschiedlichen Vorkenntnissen erlernt werden?
- Funktionalität:
Auf welche Palette von Problemstellungen kann die Software angewendet werden?
- Verfügbarkeit:
Wie lange dauert der Entwicklungsprozess?
- (möglichst) niedrige Entwicklung- und Unterhaltungskosten:
Wie teuer ist die Software in Entwicklung, Anschaffung und Unterhaltung?

- Erweiterbarkeit:
Wie leicht kann die Software an sich ändernde Anforderungen angepasst werden?
- Kompatibilität:
Wie leicht lässt sich die Software an andere bestehende Softwarepakete anbinden?
- Portabilität:
Auf welchen Plattformen ist die Software einsetzbar?
- Wiederverwendbarkeit:
Sind Bestandteile der Software in anderen Applikationen einsetzbar?
- Verstehbarkeit:
Ist der Quellcode für normale Programmierer verstehbar?
- Testbarkeit:
Wie einfach ist es, die Software auf Korrektheit und Robustheit zu prüfen?
- ...

führten ~ 1980 – 90 zur sog. *Softwarekrise*, in deren Verlauf man sich an die OO-Konzepte erinnerte.

Die bis dahin ein Nischendasein fristenden “alten“ objektorientierten Sprachen (SIMULA und dessen “Ableger“ SMALLTALK, ~ 1969) erhielten größere Aufmerksamkeit (zumal die inzwischen erreichte Leistungsfähigkeit der Hardware deren Anforderungen eher genügte) und es wurden eine Reihe neuer “objektorientierter“ Sprachen entwickelt:

- *puristische*, also rein objektorientierte Sprachen, etwa:
 - OBERON ($\sim 1985 - 87$)
 - OBERON 2 (~ 1991)
 - EIFFEL (~ 1986)
 - JAVA (~ 1995)
 - ...
- hybride Sprachen, also Erweiterung prozeduraler Sprachen um objektorientierte Sprachmittel, etwa:
 - C++ (als Erweiterung von C), ~ 1983
 - TURBO-PASCAL ab Version 5.5, $\sim 1988(?)$
 - OBJECT-COBOL, ~ 1990
 - ...

1.3 Geschichte von C++

Zuerst entworfen (und als Präprozessor zu C-Compilern realisiert) wurde C++ 1983 von dem bei den Bell-Labs in den USA arbeitenden Dänen Bjarne Stroustrup.

Zahlreiche andere Softwareentwickler griffen die Ideen auf, passten sie ihren Bedürfnissen an und entwickelten die Sprache weiter.

Um 1985 gab es die ersten “richtigen“ C++-Compiler.

1989 wurde ein ANSI-Komitee zur Standardisierung der Sprache eingesetzt.

1990 wurde das *semioffizielle* Manual herausgegeben:

The Annotated C++-Reference Manual von M. Ellis und B. Stroustrup.

1998 wurde der ISO/IEC C++-Standard (14882, 1998) verabschiedet ([ISO 98]).

Viele Konzepte der Standardbibliothek sind gegenüber den Anfängen von C++ völlig überarbeitet.

Seitdem mühen sich die Compilerhersteller, mit ihren Produkten diesem Standard möglichst nahe zu kommen.

Die auf unseren Praktikumsrechnern eingesetzten Compiler (*Gnu C++-Compiler, Version 2.95.2* und *SUN-Workshop-C++, Version 6.1*) kommen dem Standard schon recht nahe, erreichen ihn aber in einigen Details noch nicht ganz.

Teil I

Einführung

Kapitel 2

C++ als Erweiterung von C

2.1 Formales Aussehen von C++-Programmen

C++ ist eine Erweiterung der Programmiersprache C, d.h. (fast) alle C-Programme lassen sich fehlerfrei auch durch einen C++-Compiler übersetzen.

Insbesondere entspricht das formale Aussehen eines C++-Programms dem eines C-Programmes:

- Nach ein paar `#include`-Direktiven, Definition bzw. Deklaration von globalen Daten bzw. von Funktionen folgt die Definition einer Funktion mit dem Namen `main` (Typ wie in C: `int main()` oder `int main(int argc, char **argv)`) und es kann die Definition weiterer Funktionen erfolgen.
- Es steht ein Präprozessor mit (mindestens) den gleichen Funktionalitäten wie in C zur Verfügung.
Insbesondere sind die Präprozessor-Direktiven `#include <...>`, `#include "..."` zum Einbinden von Headerdateien, `#define ...` zur Definition von symbolischen Konstanten oder auch Makros mit Argumenten, `#if`, `#ifdef`, `#ifndef`, `#elif` und `#endif` zum bedingten Compilieren sowie `#undef ...` zur Aufhebung einer Makro-Definition verfügbar.
- Ein komplexeres Programm kann auf mehrere Quelldateien aufgeteilt sein:
 - die Deklaration gemeinsam verwendeter Dinge (Variablen, Typen, Funktionen,...) kann in entsprechenden Headerdateien (Endung üblicherweise `.h`) geschrieben werden, welche dann an entsprechender Stelle durch `#include` einzubinden sind,
 - die einzeln übersetzbaren Quelltexte (Endung je nach System `.cc` oder `.C` oder `.cpp` oder sonstig) müssen (nach ihrer Übersetzung) zur Erstellung des lauffähigen Programms zusammengebunden werden, hierbei sind ggf. auch Bibliotheken hinzuzubinden.

C++ verfügt über eine eigene, sehr umfangreiche und mächtige Standardbibliothek, welche nach Einbindung entsprechender Headerdateien in den Quelltext — hierbei braucht i. Allg. eine eventuelle Endung nicht angegeben zu werden, etwa:

```
#include <iostream>
```

— und nach (weitgehend automatischem) Hinzubinden der eigentlichen Bibliothek beim Linken verwendet werden kann.

Die Funktionalitäten der (alten) Standard-C-Bibliothek wird durch die neue C++-Bibliothek mit angeboten — den Namen der Headerdateien zur alten C-Bibliothek ist jedoch ein kleines `c` voranzustellen, also etwa:

```
#include <cstdio>
```

anstelle des in C üblichen

```
#include <stdio.h>
```

Funktionen der C++-Standardbibliothek (etwa auch die von C stammende Funktion `printf`) können (nach Einbinden der entsprechenden Headerdatei) dann über explizite Qualifikation mittels `std::`, also etwa:

```
erg = std::printf("hello, world\n");
```

aufgerufen werden.

Diese explizite Qualifikation mit `std::` aller Funktionen der Standardbibliothek ist bei einigen Compilern zwingend vorgeschrieben (etwa beim SUN-Workshop), bei anderen (etwa dem gcc) nicht.

Durch die Anweisung

```
using namespace std;
```

unmittelbar nach Einbinden der Headerdateien kann man man die ständige Angabe der expliziten Qualifikation `std::` vermeiden. (Hierauf wird im Zusammenhang mit *Namensräumen* noch genauer eingegangen.)

Einige Dinge, welche auf gängigen C-Compilern allenfalls eine Warnung hervorrufen, liefern auf C++-Compilern eine Fehlermeldung. Dies betrifft insbesondere die ordnungsgemäße Deklaration einer Funktion. In C++ muss jede Funktion, bevor sie aufgerufen werden kann, ordnungsgemäß deklariert (oder aber im selben Quelltext vorher definiert) sein!

Zur Deklaration einer Funktion gehört:

- Der Rückgabetypp der Funktion.
Anhand des Rückgabetypes entscheidet der Compiler, ob die Funktion innerhalb komplexerer Ausdrücken aufgerufen werden kann.
- Der Name der Funktion.
Dieser dient dazu, die Funktion (weitgehend) zu identifizieren.
- Die Signatur der Funktion.
Die Signatur ist die Anzahl, der Typ und die Reihenfolge der Funktionsparameter. Anhand der Signatur entscheidet der Compiler, ob die Funktion sachgemäß (mit der richtigen Anzahl, den richtigen Typen und der richtigen Reihenfolge der Argumente) aufgerufen wurde.
Eine fehlende Signatur, etwa:

```
... fkt();
```

bedeutet, dass diese Funktion keine Parameter besitzt, ist also gleichwertig zu:

```
... fkt(void);
```

2.1.1 Benutzen von “altem“ C-Code

Üblicherweise werden in C++ Funktionen durch den Linker anders behandelt als in C (andere interne Namen, ...).

Um in einem C++-Programm selbstgeschriebene C-Funktionen (welche man etwa aus einem früheren Projekt in C übernehmen möchte, ohne die Funktionen in C++ neu zu definieren), muss dem Compiler (und dem Linker) mitgeteilt werden, dass die zugehörigen Objektdateien von einem C-Compiler (mit sog. *C-Bindung*) erstellt worden sind.

Die Deklaration eines solchen, aus einer C-Objektdatei stammenden Objektes (Funktion oder Variable) in dem C++-Quelltext muss dann wie folgt aussehen:

```
extern "C" int f(double); /* Deklaration einer C-Funktion mit
                           int-Rueckgabe und double-Argument */
extern "C" double x;      /* Deklaration einer globalen C-Variablen */
```

Derartige extern "C"-Angaben kann man auch wie folgt zusammenfassen:

```
extern "C" {
int f(double); /* Deklaration einer C-Funktion mit int-Rueckgabe
                  und double-Argument */
double x;      /* Definition einer globalen C-Variablen mit
                  impliziter Initialisierung 0 */
}
```

wobei hier jedoch das `double x` (ohne explizite Initialisierung) definiert (und nicht nur deklariert) wird! Will man so die Variable nur deklarieren und nicht definieren, so muss das `extern` nochmals angegeben werden:

```
extern "C" {
int f(double); /* Deklaration einer C-Funktion mit int-Rueckgabe
                  und double-Argument */
extern double x; /* Deklaration einer globalen C-Variablen */
}
```

Man kann auch wie folgt eine “alte“ (aus C stammende) Headerdatei (etwa mit dem Namen `projekt.h`) in einem C++-Programm verwenden:

```
extern "C" {
#include "projekt.h"
}
...
```

und hiermit sind alle in `projekt.h` deklarierten Funktionen und Variablen im C++-Programm verfügbar.

Mit dem Makro `__cplusplus`, welches von i. Allg. von C++-Compilern explizit gesetzt wird, kann man C-Headerdateien für C- und C++-Programme anwendbar machen:

```

#ifdef __cplusplus
extern "C" {
#endif

/* es folgt die eigentliche C-Headerdatei */

    int f(double);    /* Deklaration einer C-Funktion mit int-Rueckgabe
                        und double-Argument                                */
    extern double x; /* Deklaration einer globalen C--Variablen            */

    ...

#ifdef __cplusplus
}
#endif

```

Diese Technik wird i. Allg. auch von den C-Headerdateien der C++-Standardbibliothek (etwa `cstdio.h`, `cstring.h`, ...) verwendet.

2.2 Kommentare in C++-Programmen

Neben den aus C bekannten Kommentaren, welche durch ein `/*` eingeleitet und durch das nächste `*/` beendet werden und sich somit auch über mehrere Zeilen erstrecken können, gibt es in C++ den durch `//` eingeleiteten Zeilenende-Kommentar.

Diese Zeichen `//` und der Rest der Zeile werden als Kommentar betrachtet und sind für den Compiler unerheblich.

Derartige Kommentare bestehen also aus höchstens einer Zeile — will man so mehrere Zeilen als Kommentar haben, so muss man in jede Zeile `//` schreiben!

Kommentiert man standardmäßig zunächst mittels `//`, so hat man die Möglichkeit, mittels `/*` und `*/` einen ganzen Programmteil — inklusive der `//`-Kommentare — auszukommentieren.

2.3 Typen

2.3.1 Standardtypen

In C++ existieren alle Standardtypen aus C:

- die Zeichentypen `signed char` und `unsigned char` (der Typ `char` ist abhängig vom System einer von beiden!),
- die ganzzahligen Typen `signed short` (= `short`), `unsigned short`, `signed int` (= `int`), `unsigned int`, `signed long` (= `long`) und `unsigned long`,
- die Gleitkommatypen `float`, `double` und `long double`.

Neben diesen Typen gibt es neue Typen:

- Für boolsche Werte den Typ `bool`, der die Werte `true` für *wahr* und `false` für *falsch* haben kann. Ähnlich wie in C wird in arithmetischen Kontexten der Wert `true` als ganzzahlige 1 interpretiert und `false` als ganzzahlige 0. Umgekehrt wird ein arithmetischer Wert (Zahlwert) oder auch ein Adresswert in einem boolschen Kontext (etwa als Bedingung) genau dann als `true` interpretiert, falls der Zahlwert bzw. die Adresse von 0 verschieden ist.
- Den systemabhängige Datentyp `wchar_t` zur Darstellung eines länderspezifischen, größeren Zeichensatzes.

Boolsche Werte, Zeichen und die ganzzahligen Typen werden zusammen als *integrale Typen* bezeichnet (man kann mit ihnen ganzzahlig rechnen). Die integralen Typen bilden zusammen mit den Gleitkommatypen die *arithmetischen Typen*.

Aus diesen grundlegenden Typen können (wie in C) weitere Typen “konstruiert” werden:

- Feldtypen, etwa `int feld[100];`
- Zeigertypen, etwa `int *p;`
- Referenztypen (in C nicht vorhanden!), etwa `int i, &ir = i;`
(Mehr zu Referenzen in Abschnitt 2.6.)

Weiterhin gibt es den “Typ” `void`, der eigentlich kein richtiger Typ ist, sondern “Abwesenheit von Information” bedeutet. (Der hieraus konstruierte Typ `void *` ist jedoch ein “richtiger Typ”!)

2.3.2 Benutzerdefinierte Typen

Neben den im vorigen Abschnitt behandelten “eingebauten” Typen kann der Benutzer wie in C mittels `enum` oder `struct` (und in C++ mit dem `struct` ähnelnden `class`, siehe Kapitel 4) eigene Typen definieren:

```
enum Farbe { rot, gruen, blau};
struct Person {
    char name[32];
    char vorname[32];
    ...
};
```

Im Gegensatz zu C werden hierdurch Typen mit den Namen `Farbe` und `Person` definiert (die Typnamen bestehen aus einem Wort, in C hießen diese Typen noch `enum Farbe` bzw. `struct Person` — die Typnamen bestanden also aus zwei Worten!). Man kann in C++ somit Variablen von den entsprechenden Typen einfach wie folgt definieren:

```
Farbe color;           // Typ: Farbe
Farbe *colp;           // Adressvariable vom Typ: Farbe
Person chef;           // Typ: Person
Person mitarbeiter[64]; // Feld vom Typ: Person
```

2.3.3 Typumwandlungen

Standardkonversionen sind die aus C bekannten Umwandlungen des Ergebnisses eines Ausdrucks von einem Standardtyp in einen anderen (etwa von `long` nach `double`). Diese werden ggf. vom System auch implizit (etwa bei Funktionsaufrufen zur Angleichung der tatsächlichen Argumenttypen an die Typen der Parameter) durchgeführt. Mittels *Konstruktoren* (wird später behandelt) und *Konversionsoperatoren* (wird ebenfalls später behandelt) kann man festlegen, wie eine Umwandlung zwischen einem benutzerdefinierten Typen und anderen Typen (benutzerdefiniert oder Standard-) oder umgekehrt zu geschehen hat. Auch hiermit mögliche Typumwandlungen werden ggf. implizit vom System durchgeführt.

Neben diesen impliziten Typumwandlungen gibt es die expliziten, zunächst die aus C bekannte erzwungene Typumwandlung mittels des *cast*-Operators, etwa:

```
double *feld = (double *) malloc ( 1000*sizeof(double));
```

Hier wird das Ergebnis der dynamischen Speicheranforderung (`malloc`), welches vom Typ `void *` (also eine typlose Adresse) ist, mittels `(double *)` in den Typ `double *` (also eine `double`-Adresse) umgewandelt. Man beachte, dass es zwischen `void *` und `double *` keine Standardumwandlung gibt.

Bei der erzwungenen Typumwandlung:

`(typ) ausdruck`

wird der Wert des Ausdrucks (irgendwie) in den in Klammern angegebenen Typen umgewandelt. Wie umgewandelt wird, entzieht sich der Kontrolle des Programmierers und ist maschinenabhängig und derartige Umwandlungen sind häufig Ursache für Portierungsprobleme.

Eine Umwandlung in einen Standard- oder selbstdefinierten Typ (nicht in einen aus solchen Typen "konstruierten" Typen wie etwa einen Feld- oder Adresstyp) kann auch wie folgt explizit erreicht werden (funktionale Schreibweise):

`typ (ausdruck)`

Der angegebene Ausdruck wird ausgewertet und in den angegebenen Typen umgewandelt. (Mein Verständnis dieses Operators ist, dass er tunlichst nur dann angewendet werden sollte, wenn hierdurch eine Umwandlung mittels Standardumwandlung, Konstruktor oder Konversionsoperator durchgeführt und nicht "irgendwie" umgewandelt wird!)

Feinere Abstufungen in der Art und Weise, wie umgewandelt werden soll, bilden die neuen Umwandlungsoperatoren:

- `const_cast<typ> (ausdruck)` (Konstantheit "wegcasten", wird in Abschnitt 2.7.5 behandelt)

- `static_cast<typ> (ausdruck)`
zur Typumwandlung zwischen verwandten Typen (etwa zwischen Zeigertypen oder zwischen arithmetischen Typen). Eine übliche Anwendung wäre somit:

```
double *feld = static_cast<double *> (malloc ( 1000*sizeof(double)));
```

die von `malloc` gelieferte Adresse vom Typ `void *` wird als Adresse vom Typ `double *` interpretiert.

- `dynamic_cast<typ> (ausdruck)` (wird in Abschnitt 7.5.2 behandelt)
- `reinterpret_cast<typ> (ausdruck)`
zur Neuinterpretation eines Speicherbereiches (ist also die am weitesten system-abhängige Art der Umwandlung), etwa:

`double x; char *p = reinterpret_cast<char *> (& x);`

um sich beispielsweise anzusehen, wie ein `double` intern (als Folge von `char`'s) abgespeichert ist.

Der Vorteil dieser Abstufungen der Typumwandlung ist, dass nicht zwangsläufig (wie in C mit casts bzw. wie mit dem `reinterpret_cast<typ>` in C++ immer noch möglich) *irgendwie* umgewandelt wird!

2.3.4 Deklarationen von Variablen

Wie in C muss jede Variable, bevor sie verwendet wird definiert bzw. deklariert werden.

Bezüglich Speicherklassen, Initialisierung, Zugreifbarkeit und Zeitpunkt der Erzeugung und Zerstörung von Variablen gilt in C++ genau dasselbe wie in C.

Im Gegensatz zu C, wo automatische Variablen nur am Anfang einer Verbundanweisung (unmittelbar hinter der öffenden geschweiften Klammer, vor der ersten Anweisung) erfolgen durften, kann in C++ eine Variable an jeder Stelle definiert oder auch deklariert werden, an der auch eine Anweisung stehen dürfte.

Hierdurch hat man die Möglichkeit, eine Variable erst genau dann zu definieren, wenn man diese Variable benötigt (und i. Allg. dann auch einen Wert hat, mit dem man sie vorbesetzen kann!).

Darüberhinaus kann eine Variable auch

- in der Initialisierung einer `for`-Schleife definiert werden:

```
...
for ( int i=0; i< 100; ++i)
{ ...}
...
```

Dieser Schleifenzähler `i` ist nur innerhalb der `for`-Schleife bekannt, würde in C also folgender Konstruktion entsprechen:

```
...
{ int i;
  for ( i = 0; i < 100; ++i)
  {...}
}
...
```

- in der Bedingung einer `if`-Anweisung definiert werden:

```
...
if ( int i = x + y )
{ ...}
else
{ ...}
...
```

Diese Variable `i` ist nur innerhalb der kompletten `if`-Anweisung bekannt (einschließlich des optionalen `else`-Teils).

- in der Bedingung einer `while`-Anweisung definiert werden:

```
...
while ( int i = fkt() )
{ ...}
...
```

Diese Variable `i` ist nur innerhalb der kompletten `while`-Schleife bekannt.

2.4 Der Gültigkeitsbereichsauflösungsoperator ::

Wie in C gilt auch in C++, dass ein lokaler Name (etwa eine Variable) ein übergeordnetes gleichnamiges Objekt “überdeckt“:

```
...
int main()
{ int i;
  ...
  for( ...;...;...)
  {
    double i;
    ...
  }
  ...
}
```

Die Variablen `int i` und `double i` sind zwei unterschiedliche Objekte, verwendet man innerhalb der `for`-Schleife den Namen `i` zu, so greift man auf die `double`-Variable zu, außerhalb der Schleife bezieht sich `i` auf die `int`-Variable.

Gibt es innerhalb einer Verbundanweisung (etwa dem Anweisungsteil einer Funktion) eine lokale Variable mit dem gleichen Namen wie eine (wichtig:) globale Variable, so kann man innerhalb der Verbundanweisung mittels des einfachen Namens auf die lokale Variable und mittels des *Gültigkeitsbereichsauflösungsoperators* `::` auf die globale Variable zugreifen:

```

...
int i;           // globales i

int main()
{ int k,j;
  double i;      // lokales i
  ...
  k = j + i;     // Zugriff auf lokales i
  ...
  k = j + ::i;   // Zugriff auf globales i
  ...
}
...

```

(Dieser Operator `::` hat eine ziemlich hohe Priorität.)

2.5 Namensbereiche

2.5.1 Grundlagen zu Namensbereichen

In C belegen globale Objekte (Funktionen und externe Variablen) einen Namensraum. D.h. es darf keine zwei gleichnamigen globalen Objekte geben — z.B. ist es nicht möglich, dass eine Funktion mit dem Namen `g` und gleichzeitig eine globale `double`-Variable mit demselben Namen existiert!

Mit dem Schlüsselwort `static` hat man in C nur die Möglichkeit, die Bekanntheit eines globalen Namens (Funktion oder Variable) auf einen Quelltext einzuschränken — ein solches statisches globales Objekt darf den gleichen Namen haben wie ein globales Objekt eines anderen Quelltextes!

Dies führt in größeren Projekten, an denen mehrere Entwickler oder mehrere Softwarepakete beteiligt sind, zu Problemen, da alle von unterschiedlichen Entwicklern oder Softwarepaketen stammenden globalen Namen verschieden sein müssen!

C++ bietet durch *Namespace*s die Möglichkeit, zusammengehörige globale Namen (Variablen, Funktionen, Typen, ...) zu einem Namensraum (*Namespace*) zusammenzufassen. Ein solcher Namensraum hat (i.Allg) einen Namen

```

namespace A { // A: Name des Namensraumes

    // Deklaration von Funktionen und Variablen

    int fkt(double);
    extern double x;
    ...
} // Ende des Namensraumes A

```

und der Zugriff auf einen in einem Namensraum eingeführten Namen erfolgt durch explizite Qualifikation:

```
double A::x;    // Definition der Variablen x aus Namensraum A
...
int main()
{ ...
  A::x = 3.1;    // Zugriff auf Variable X aus Namensraum A
  ...
  A::fkt(2.7);  // Aufruf der Funktion fkt aus Namensraum A
  ...
}

int A::fkt1(double y)
{ ... }          // Definition der Funktion fkt aus Namensraum A
```

Bei konsequenter Anwendung von Namensräumen sind gleichnamige globale Variablen oder Funktionen aus unterschiedlichen Namensbereichen (etwa aus mehreren, in einer Anwendung zu integrierenden Softwarepaketen) kein Problem:

1. Deklaration und Definition von Softwarepaket **paket1**:

(a) Deklaration in der Headerdatei **paket1.h**:

```
namespace paket1 {
  ...
  int f(void);
  void g(int);
  void h(void);
  ...
}
```

(b) Definition der Funktionen in einem Quelltext (etwa **paket1.cc**):

```
#include "paket1.h"
...
int paket1::f(void) {...}
void paket1::g(int i) {...}
void paket1::h(void) {...}
...
```

2. Deklaration und Definition von Softwarepaket **paket2**:

(a) Deklaration in der Headerdatei **paket2.h**:

```
namespace paket2 {
  ...
  int f(void);
  void g(int);
  void h(void);
  ...
}
```

(b) Definition der Funktionen in einem Quelltext (etwa **paket2.cc**):

```

#include "paket2.h"
...
int paket2::f(void) {...}
void paket2::g(int i) {...}
void paket2::h(void) {...}
...

```

3. Benutzung von beiden Softwarepaketen in einer Anwendung (beide übersetzten Softwarepakete sind der Anwendung hinzuzubinden!):

```

#include "paket1.h"
#include "paket2.h"

int main()
{ int i;
  ...
  i = paket1::f();    // Funktion f aus paket1
  i = paket2::f();    // Funktion f aus paket2
  ...
  paket1::g(i);       // Funktion g aus paket1
  paket2::g(i);       // Funktion g aus paket2
  ...
  paket1::h();        // Funktion h aus paket1
  paket2::h();        // Funktion h aus paket2
  ...
}

```

2.5.2 using-Deklarationen bzw. die using-Direktiven

Mittels des Schlüsselwortes `using` kann man einen Namen aus einem Namensbereich so zur Verfügung stellen, dass dieser nicht mehr explizit qualifiziert werden muss (*using-Deklaration*):

```

#include "paket1.h"
#include "paket2.h"

using paket1::f;
using paket2::g;

int main()
{ int i;
  ...
  i = f();             // Funktion f aus paket1
  i = paket2::f();     // Funktion f aus paket2
  ...
  paket1::g(i);        // Funktion g aus paket1
  g(i);               // Funktion g aus paket2
}

```

```

...
paket1::h();      // Funktion h aus paket1
paket2::h();      // Funktion h aus paket2
...
}

```

Man kann so auch alle Namen eines ganzen Namensbereich bekannt machen, so dass eine explizite Qualifikation nicht mehr nötig ist (*using-Direktive*):

```

#include "paket1.h"
#include "paket2.h"

using namespace paket1;

int main()
{ int i;
  ...
  i = f();          // Funktion f aus paket1
  i = paket2::f();  // Funktion f aus paket2
  ...
  g(i);            // Funktion g aus paket1
  paket2::g(i);    // Funktion g aus paket2
  ...
  h();            // Funktion h aus paket1
  paket2::h();    // Funktion h aus paket2
  ...
}

```

Eine *using-Direktive* kann auch lokal (etwa im Anweisungsteil einer Funktion) erfolgen:

```

#include "paket1.h"
#include "paket2.h"

int fkt1(void)
{ using namespace paket1;
  ...
  h();          // Funktion h aus paket1
  ...
}

int fkt2(void)
{ using namespace paket2;
  ...
  h();          // Funktion h aus paket2
  ...
}

```

Weiterhin ist es möglich, mit mehreren `using`-Direktiven die (nicht zueinander in Konflikt stehenden) Namen aus mehreren Namensbereichen bekannt zu machen:

```
namespace A {
    void f(void); // auch in B
    void g(void); // nicht in B
}

namespace B {
    void f(void); // auch in A
    void h(void); // nicht in A
}

using namespace A;
using namespace B;

int main()
{ ...
    g(); // ok: A::g
    h(); // ok: B::h
    f(); // Fehler: A::f oder B::f ???
    A::f(); // ok: explizit A::f
    B::f(); // ok: explizit B::f
    ...
}
```

2.5.3 Namespaces und Funktionsargumente

Zur Vereinfachung von Funktionsaufrufen gilt Folgendes:

Ist beim Aufruf einer Funktion `f` ein Argument von einem in einem Namensbereich definierten Typ und ist im aktuellen Kontext dieser Typ bekannt, die Funktion jedoch nicht — weil der Namensbereich der Funktion nicht direkt über explizite Qualifikation oder indirekt über `using` bekannt gemacht ist —, so sucht der Compiler im Namensbereich des Argumentes nach einer passenden Funktion mit dem Namen `f`.

Findet er nur eine passende Funktion `f` in einem der Namensbereich der Typen der Argumente, so nimmt er diese!

Findet er mehrere passende Funktionen, weil etwa ein anderes Argument einen anderen Typ hat, der in einem anderen Namensbereich definiert wurde und es dort auch ein passendes `f` gibt, so ist dies ein Fehler!

Beispiel:

```
namespace A {
    struct Atyp { }; // neuer Typ in A
}

namespace B {
    struct Btyp { }; // neuer Typ in B
}
```

```

using A::Atyp;
using B::Btyp;

namespace A {
    void h(int i) {}
    void f(Atyp a) {}
    void g(Atyp a, Btyp b) {}
}
namespace B {
    void g(Atyp a, Btyp b) {}
}

int main()
{
    Atyp a;
    Btyp b;
    int i;

    h(i);    // Fehler: h nicht bekannt
    f(a);    // ok: a hat Typ Atyp aus A und
             // in A gibt es passendes f
    g(a,b);  // Fehler: in A und B ist jeweils
             // ein passendes g
    ...
}

```

An diesem Beispiel sieht man auch, dass Namensbereiche *offen* sind, d.h. man kann immer noch neue Namen zu einem Namensbereich hinzufügen!

2.5.4 Geschachtelte Namensbereiche

Man kann Namensbereiche wie folgt schachteln:

```

namespace A {
    ...
    namespace B {
        ...
        void h(void);
        ...
    }
    namespace C {
        ...
        void h(void);
        ...
    }
    ...
}

```


und man kann mit geschachtelter Qualifikation auf die entsprechenden Namen zugreifen:

```
A::B::h(); /* h aus dem Unternamensbereich B
           des Namensbereiches A          */
A::C::h(); /* h aus dem Unternamensbereich C
           des Namensbereiches A          */
```

Entsprechende `using`-Deklarationen:

```
using A::B::h;
```

oder `using`-Direktiven:

```
using namespace A::B;
```

funktionieren auch!

2.5.5 Unbenannte Namensbereiche

Ein *unbenannter Namensbereich* ist ein in einem Quelltext wie folgt definierter Namensbereich

```
...
namespace {
    void h(void) { ...}
    double x = 3.4;
    ...
}
```

Diese Definition weist zwei Besonderheiten auf:

1. Es ist kein Name für den Namensbereich angegeben
2. Alle Funktionen und Variablen müssen innerhalb dieses Namensbereiches komplett definiert (und nicht nur deklariert) werden.

Die Bedeutung eines solchen Namensbereiches ist: Alle in diesem Bereich aufgeführten Funktionen, Variablen, oder auch Typen sind nur in diesem Quelltext — und ohne dass eine explizite Qualifikation notwendig (oder möglich) wäre — verfügbar. Aus anderen Quelltexten heraus kann nicht auf sie zugegriffen werden. Somit ersetzt ein unbenannter Namensbereich die Technik aus C, statische globale Funktionen und Variablen (Schlüsselwort `static`) zu definieren.

2.5.6 Weitere Techniken zu Namensbereichen

Alias-Namen für Namensbereiche

Bei Namensbereichen entsteht das Problem, vernünftige Namen für solche Namensbereiche zu wählen!

Lange Namen sind aussagekräftig aber unhandlich, wenn man diese Namen bei der Erstellung einer Anwendung immer und immer wieder eintippen muss!

Hier hilft folgender Trick:

- bei Erstellung einer universell einsetzbaren Bibliothek sollte man ruhig lange, aussagekräftige Namen für Namensbereiche wählen, etwa:

```
namespace Microsoft_Foundation_Classes_5_01 { ... }
```

- In einer Anwendung kann man dem Namensbereich mit dem langen Namen einen kurzen Alisanamen geben:

```
namespace MFC = Microsoft_Foundation_Classes_5_01;
```

und im Folgenden nur noch mit dieser Abkürzung arbeiten:

```
using MFC::dieses;
using MFC::jenes;
...
...MFC::fkt(...);
...
```

Schnittstellen zusammensetzen

Stehen Funktionalitäten aus mehreren Namensbereichen zur Auswahl, so kann man sich seine favorisierte Auswahl der angebotenen Funktionalitäten selbst in einem neuen Namensbereich zusammenstellen:

```
namespace meine_schnittstelle {
    using namespace A;      // namespace A wird ganz bekannt gemacht
    using B::Lin_Liste...; // verwende lineare Liste aus B
    using C::sort...;       // verwende Sortierfunktion aus C
}
```

Durch

```
using namespace meine_schnittstelle;
```

hat man dann seine Schnittstelle “eingeschaltet“.

2.6 Referenzen

Eine Referenz ist ein weiterer Name für eine bereits vorhandene Variable.

Definiert wird eine Referenz etwa wie folgt:

```
int a, &b=a;
```

a ist eine gewöhnliche `int`-Variable und **b** ist eine `int`-Referenz (kenntlich gemacht durch das vorangestellte `&`). Eine Referenz muss bei ihrer Definition sofort initialisiert werden mit einer Variablen (oder einem anderen Ausdruck, der einen Speicherbereich vom entsprechenden Typ identifiziert!) und die Referenz ist ein weiterer Name für die bei der Initialisierung angegebene Variable (bzw. den angegebenen Speicherbereich!). In obigem Beispiel sind **a** und **b** zwei Namen für ein- und dieselbe Variable (wobei **a** die eigentliche Variable und **b** nur ein anderer Name für diese ist!). Diese Referenz kann wie die Variable verwendet werden:

```

b = 5;           // a bekommt den Wert 5
printf("%d\n",b); // Wert von a wird ausgegeben
scanf("%d",&b);  // a wird eingelesen:

```

Der Ausdruck `&b` bedeutet, wie oben gesehen: Adresse der Variablen `a` — `b` ist ja nur ein anderer Name für `a`. Referenzen selber haben keine Adresse, sondern nur das Objekt, für welches die Referenz ein weiterer Name ist!

Durch entsprechende Referenzen kann man noch mehr Namen für ein- und dieselbe Variable erzeugen:

```

int a;
int &b = a, &c = a;
int &d = b, &e = c;
...

```

Jeder der Namen `b`, `c`, `d` und `e` beziehen sich auf die Variable `a`.

Im Gegensatz zu Zeigern (wo etwa `char ***p` ja Zeiger auf Zeiger auf Zeiger auf `char` bedeutet und somit mehrere `*`'ne bei einer entsprechenden Definition direkt hintereinander stehen können) ist der Referenzmechanismus nur einschichtig, d.h. die Definition

```
int a, &b=a, &&c=b; // &&c ist Fehler
```

ist nicht zulässig (es gibt keine Referenzen auf Referenzen!).

Ebenso unzulässig sind Adress-Variablen auf Referenzen:

```
int &* a; // Fehler: Adresse auf Referenz nicht moeglich
```

Eine Referenz auf eine Adress-Variable hingegen ist möglich:

```
int *p, *&q = p; // q ist zweiter Name fuer Adressvariable p
```

Angewendet werden Referenzen hauptsächlich im Zusammenhang mit Funktionen!

2.6.1 Referenzen als Funktionsparameter

Wird eine Funktion wie folgt definiert:

```

void swap ( int &a, int &b)
{ int tmp;
  tmp = a;
  a = b;
  b = tmp;
}

```

und wie folgt aufgerufen:

```

int main()
{ int i,j;
  ...
  swap(i,j);
  ...
}

```

so sind die Funktionsparameter Referenzen — also nur andere Namen für die Variablen, mit denen sie initialisiert werden. Die Initialisierung der Parameter erfolgt aber beim Aufruf der Funktion anhand der tatsächlichen Funktionsargumente, d.h. die Funktionsparameter (vom Referenztyp) sind andere Namen für die beim Funktionsaufruf stehenden Argumente. Folglich kann dann innerhalb der Funktion auf die als Argument angegebenen Variablen des aufrufenden Programnteils zugegriffen werden — dies ist in C bekanntlich nur über Zeiger und Adressen möglich!

Beim Aufruf `swap(i, j)`; hier im Beispiel in `main` ist somit innerhalb von `swap` der Referenzparameter `a` nur ein anderer Name für die Variable `i` und der Referenzparameter `b` ein anderer Name für die Variable `j` und innerhalb von `swap` wird über den Namen `a` auf die Variablen `i` aus `main` und über `b` auf die `main`-Variable `j` zugegriffen und diese Variablen aus `main` werden in der Tat vertauscht.

Da über den Referenzmechanismus Objekte (mit Speicherbereich) übergeben werden, dürfen beim Aufruf einer derartigen Funktion als Argumente auch nur Ausdrücke stehen, die einen Speicherbereich identifizieren — denen man somit auch etwas zuweisen könnte (*l-value*).

Dies ist für beliebige Ausdrücke nicht der Fall, z.B. ist der Ausdruck `i+j` zwar vom Typ `int`, bezeichnet aber kein Objekt (ist kein *l-value*). Somit ist folgender Aufruf der `swap`-Funktion nicht zulässig:

```
swap( i+j, j);
```

(Wie man den Compiler dazu bringt, hier ein temporäres Objekt mit dem Wert von `i+j` zu erzeugen und die Funktion mit diesem als Argument aufzurufen, wird in Abschnitt 2.7.3 erläutert!)

In C++ hat man somit zwei Möglichkeiten, einer Funktion ein Argument so zu übergeben, dass dieses Argument von der Funktion abgeändert werden kann:

1. über Adressen (wie in C):

Deklaration: `void fkt(int *);`

Aufruf: `fkt(&i);`

2. über Referenzen:

Deklaration: `void fkt(int &);`

Aufruf: `fkt(i);`

Leider ist für den Compiler der Aufruf einer Funktion mit Referenzparametern nicht von einem Aufruf einer Funktion mit gewöhnlichen Argumenten (*Call by Value*) zu unterscheiden:

Deklaration: `void fkt(int);`

Aufruf: `fkt(i);`

Aus diesem Grund sollten Funktionen mit Referenzparametern nur mit Bedacht und gut dokumentiert eingesetzt werden!

2.6.2 Referenzen als Funktionsergebnisse

Man kann auch Funktionsergebnisse vom Referenztyp vereinbaren (bei Definition und Deklaration übereinstimmend anzugeben!), etwa:

```
Deklaration:  int & fkt( ...);  
Definition:  int & fkt(...)  
            {  
                ...  
                return ausdruck;  
            }
```

Hier wird das Funktionsergebnis (**ausdruck**) als Referenz zurückgegeben, also im aufrufenden Programmteil kommt ein **int**-Objekt (i. Allg. also eine **int**-Variable) an (und nicht wie sonst: ein Wert vom Typ **int**) und im aufrufenden Programmteil kann dann auf dieses Objekt zugegriffen werden!

Man muss bei der Definition der Funktion darauf achten, dass das Objekt, welches als Referenz zurückgegeben wird, nach dem Ende der Funktion noch existiert! (Sonst würde im aufrufenden Programmteil auf ein Objekt zugegriffen, welches gar nicht mehr vorhanden ist! Vernünftige Compiler melden die Rückgabe lokaler Objekte mittels Referenz als einen Fehler!)

Vernünftige Anwendungen sind:

1. Das Objekt, welches als Referenz zurückgegeben wird, wird vorher (als Referenz oder über Adresse) beim Funktionsaufruf der Funktion übergeben:

```
int & fkt ( int &a, ...)  
{  
    ...  
    return a;  
}
```

oder

```
int & fkt ( int *a, ...)  
{  
    ...  
    return *a;  
}
```

2. Das Objekt, welches als Referenz zurückgegeben wird, wird innerhalb der Funktion dynamisch erzeugt:

```
int & fkt ( ...)  
{  
    int *p = (int *) malloc(sizeof(int));  
    ...  
    return *p;  
}
```

(Problematisch hierbei ist die spätere Freigabe des dynamisch reservierten Speicherbereiches!)

Gibt eine Funktion eine Referenz (also ein Objekt und keinen Wert) zurück, so kann der Funktionsaufruf auch an den Stellen erfolgen, wo ein Objekt und nicht nur ein Wert benötigt wird, etwa

- in Zusammenhang mit In-/Dekrementierung:

```
++fkt(...) oder fkt(...)++
```

das Funktionsergebnis wird inkrementiert!

- in Zusammenhang mit Adressen:

```
int * p = &fkt(...);
```

die Adresse des Funktionsergebnisses wird einer Adress-Variablen zugewiesen!

- in Zusammenhang mit einer Zuweisung:

```
fkt(...) = ...;
```

Die Funktion liefert ein Objekt (Variable) und diesem wird etwas zugewiesen!

- ...

2.7 Konstante Datenobjekte

Das Schlüsselwort `const` ist bereits in den ANSI-Standard von C aufgenommen worden.

Hiermit hat man die Möglichkeit, konstante Objekte mit einem echten (auch benutzerdefinierten) Typen und mit einem wie bei Variablen üblichen Gültigkeitsbereich zu definieren, etwa:

- strukturierte Konstante:

```
struct complex {
    double realteil;
    double imaginaerteil;
};
complex const i = { 0,1};
```

Die Konstante `i` hat den Typen `complex`.

- Gültigkeitsbereich:

```
int main()
{ ...
    for(...;...;...)
    { double const pi = 3.1415926;
        ...
    }
    ...
}
```

Die Konstante `pi` ist vom Typ `double` und existiert nur innerhalb des Anwendungsteils der `for`-Schleife.

Bei der Verwendung von `const`-Objekten sind einige Regeln zu beachten:

- Ein `const`-Objekt muss bei seiner Definition explizit initialisiert werden:

```
double const pi;    // Fehler: Initialisierung fehlt!
```

Der initialisierende Wert braucht erst bei der Erzeugung der Konstanten festzuliegen:

```
void fkt(int n)
{ int const m = 2*n;    // ok
  ...
}
```

Hier wird die Konstante `m` mittels des variablen Funktionsargumentes initialisiert, welches erst beim Aufruf der Funktion festliegt — aber die Konstante wird ja auch erst beim Funktionsaufruf erzeugt!

Sogar Folgendes ist möglich:

```
int f(int);          // Deklaration einer Funktion f
void fkt( int n)
{ int const m = f(n); // Konstante wird mit dem Funktions-
                      // ergebnis von f initialisiert
  ...
}
```

- Ein `const`-Objekt kann während seiner Lebenszeit nicht (vielmehr: kaum, siehe Abschnitt 2.7.5) verändert werden:

```
double const pi = 3.1415926;
...
p = 3.14;        // Fehler: Konstante kann nicht veraendert werden!
```

- Das Schlüsselwort `const` kann vor oder hinter dem Typnamen stehen und bezieht sich auf alle folgenden Bezeichner:

```
double const pi = 3.1415926, e = 2.7182818; // pi und e sind const
```

Folgendes ist äquivalent:

```
const double pi = 3.1415926, e = 2.7182818; // pi und e sind const
```

- Eine in einem Quelltext global (außerhalb einer Verbundanweisung) so definierte Konstante ist nur in diesem Quelltext bekannt und ist ein Objekt, welches nicht an den Linker weitergemeldet wird, so dass von anderen Quelltexten hierauf nicht Bezug genommen werden kann.

Folglich ist es möglich (und gängige Praxis), eine derartige Konstante in einer Headerdatei zu definieren und diese Headerdatei in allen Quelltexten, wo die Konstante benötigt wird, einfach zu includen!

- Möchte man, dass eine global definierte Konstante an den Linker weitergemeldet wird, so muss man der Definition das Schlüsselwort **extern** voranstellen:

```
extern double const pi = 3.1415926;
```

Auf diese Konstante kann in anderen Quelltexten zugegriffen werden, wenn sie in diesen Quelltexten nochmals (etwa wie folgt, ggf. auch in einer einzubindenden Headerdatei) deklariert wird:

```
extern double const pi; // keine Initialisierung => nur Deklaration
```

(Globale) Konstante Objekte (deren Werte stehen ja schon bei der Übersetzung fest) können auch an allen Stellen stehen, an denen C (und C++) einen konstanten Wert erwartet, etwa als Feldlänge bei der Definition eines Feldes:

```
const int n = 1000;
double feld[n];      // ok, n ist const
```

(Bei vielen Compilern funktioniert das auch mit lokalen konstanten Objekten!)

2.7.1 Zeiger und const

Bei Zeigern ist zu unterscheiden zwischen

- *Zeigern auf etwas Konstantes.*
Das sind (variable!) Zeiger, wobei das, auf was sie zeigen, konstant ist!
(Der Zeiger darf geändert werden — aber das, worauf gezeigt wird, darf nicht geändert werden!)
Sie werden etwa wie folgt definiert:

```
char const *p;
```

p ist eine Adress-Variable vom Typ **char const** — zeigt also auf konstante Zeichen — und braucht, da p selbst keine Konstante ist, auch nicht direkt initialisiert zu werden!

Ein Beispiel für korrekte und fehlerhafte Verwendung ist:

```
char const *p;          // p ist Zeiger auf char const
char w[100];           // Feld vom Typ char
...
p = "hello";           // ok, p bekommt Adresse von "hello"
p[0] = 'H';             // Fehler: das, worauf p zeigt, kann
                        //          nicht geändert werden
...
p = w;                 // auch ok! w ist zwar nicht const,
                        // aber Zuweisung geht trotzdem
w[0] = 'H';             // ok!
p[0] = 'h';             // Fehler: das, worauf p zeigt, kann
                        //          nicht geändert werden
```


In der letzten Anweisung des obigen Beispiels sieht man, dass der an sich variable Speicherbereich von `w[0]` über den Zeiger `char const *p` nicht abgeändert werden kann! (Hier gibt der Compiler eine Fehlermeldung aus!)

– *Konstanten Zeigern.*

Das sind Zeiger, die selber konstant sind (also immer auf dasselbe zeigen) — das, worauf gezeigt wird, kann aber durchaus verändert werden!

Derartige Zeiger können wie folgt definiert werden:

```
char w[100], * const p = w;
```

und sie müssen bei ihrer Definition explizit mit einem Wert initialisiert werden (und diesen Wert behalten sie bis an ihr Lebensende!). Nach den Sprachregeln bezieht sich ein derartiges `const` hinter einem `*` und vor einem Namen nur auf diesen Namen! Etwa in:

```
char a, * const b = &a, * c;
```

ist `a` ein gewöhnliches `char`, `b` ein konstanter Zeiger auf ein `char`, der aber (ein für allemal) auf `a` zeigt, und `c` ein gewöhnlicher (nicht initialisierter) Zeiger auf `char`.

Das, worauf ein konstanter Zeiger zeigt, kann abgeändert werden — so ist beispielsweise nach der Definition:

```
char w[100], * const p = w;
```

der Zugriff:

```
p[0] = 'H';
```

zulässig und das Feldelement `w[0]` erhält hiermit über `p` einen neuen Wert.

Der Versuch, den Wert von `p` selbst zu ändern, etwa:

```
p = "hallo"; // Fehler: p ist const
```

wird vom Compiler unterbunden (Fehlermeldung)!

Ein konstanter Zeiger darf nicht mit der Adresse eines konstanten Objektes initialisiert werden:

```
char * const p = "hallo"; // Fehler!
```

denn das, worauf `p` zeigen soll, ist `char` — aber der Typ der konstanten Zeichenkette `"hello"` ist `char const *`!

Natürlich gibt es auch *konstante Zeiger auf konstante Objekte*, die etwa wie folgt zu definieren sind:

```
char const * const p = "hello";
```

Hier ist der Zeiger selbst konstant (`* const`) und das, worauf gezeigt wird, ist ebenfalls konstant (`char const`). Ein derartiger Zeiger muss wiederum bei seiner Definition explizit initialisiert werden, wobei das, womit initialisiert wird, wiederum nicht unbedingt konstant sein muss — es kann über den Zeiger nur nicht verändert werden:

```

char w[100], v[100];           // w nicht const
char const * const p = w;      // ok!
w[0] = 'H';                    // w[0] bekommt neuen Wert
p[0] = 'h';                     // Fehler: p zeigt auf char const
p = v;                          // Fehler: p ist const

```

2.7.2 Referenzen und const

Eine Referenz an sich ist immer konstant — sie ist ja ein— für allemal ein alternativer Name für das Objekt, mit welchem sie initialisiert wird.

Aber wie bei Zeigern erhebt sich auch hier die Frage, ob das Objekt, für welches die Referenz ein weiterer Name ist, als konstant anzusehen (und somit über die Referenz nicht veränderbar) ist oder nicht.

Man betrachte folgende Beispiele:

```

int a;                          // int-Variable
int const b = 4;                // int-Konstante
int &c = a;                      // ok: c ist anderer Name fuer a
int &d = b;                      // Fehler: b ist const, d aber
                                // Referenz auf (variables) int
int const &e = b;               // ok: e ist Referenz auf int const
int const &f = a;               // auch ok: f ist Referenz auf int const
                                // a ist zwar nicht const, kann aber ueber
                                // Referenz nicht geaendert werden!
f = 7;                          // Fehler: siehe oben
a = 7;                          // ok

```

2.7.3 Funktionsparameter und const

Ob ein gewöhnlicher Funktionsparameter `const` ist oder nicht, spielt bei der Anwendung der Funktion keine Rolle (und bei der Funktionsdefinition nur eine sehr geringe). So können beide wie folgt deklarierte Funktionen

```

void fkt1( int param);
void fkt2( int const param);

```

mit einem beliebigen Argument vom Typ `int`, gleichgültig ob `const` oder nicht, aufgerufen werden:

```

int i;
...fkt1(i);      // ok, Argument ist variabel
...fkt1(7);      // ok, Argument ist const
...fkt1( i + 7); // ok, Argument ist Ausdruck
...fkt2(i);      // ok, Argument ist variabel
...fkt2(7);      // ok, Argument ist const
...fkt2( i + 7); // ok, Argument ist Ausdruck

```

In allen Fällen wird dem funktionseigenen Parameter `param` bei seiner Erzeugung (also beim Funktionsaufruf) ein Wert zugewiesen, nämlich der Wert des Funktionsargumentes. (Ob der Wert nun von einer Variablen, einer Konstanten oder einem sonstigen Ausdruck stammt, ist hierbei unerheblich!)

Der einzige Unterschied zwischen beiden Funktionen ist der, dass der Parameter `param` der Funktion `fkt1` eine Variable ist, also innerhalb der Funktion `fkt1` verändert werden kann, und dass der Parameter `param` der Funktion `fkt2` eine Konstante ist und innerhalb von `fkt2` nicht mehr abgeändert werden kann — ein ziemlich geringfügiger und zur Funktion lokaler Unterschied!

Anders sieht es bei Zeigern oder Referenzen als Funktionsparametern aus! Hier ist schon von Bedeutung, ob das, worauf gezeigt wird bzw. für das die Referenz ein weiterer Name ist, konstant ist oder nicht!

So sind folgende Aufrufe folgender Funktionen fehlerhaft:

```
void fkt1 ( int * param);
void fkt2 ( int & param);
int i,j;
const int dim=10;
...
fkt1(&dim);      // Fehler: Adresse eines const int,
                  // keine Adresse eines int!
fkt2(7);         // Fehler: Typ von 7 ist int const
fkt2(i+j);       // Fehler: Argument ist kein "Objekt",
                  // sondern nur ein Wert vom Typ int
```

Sollen die beiden Funktionen so aufgerufen werden können, so muss deren Deklaration wie folgt aussehen:

```
void fkt1 ( int const * param);
void fkt2 ( int const & param);
```

und die Funktionen müssen auch entsprechend definiert werden! Der Parametertyp von `fkt1` ist jetzt Zeiger auf `const int` und der von `fkt2` ist Referenz auf `const int`.

Aus historischen Gründen wird bei Adressen vom Typ `char` als Funktionsparametern nicht zwischen den Typen `char *` und `const char *` unterschieden.

Die wie folgt deklarierte (und definierte) Funktion:

```
void fkt(char *);
```

könnte (ohne Fehlermeldung) wie folgt aufgerufen werden:

```
char w[100], *p;
fkt("hallo"); // ok
fkt(w);       // ok:
fkt(p);       // ok:
```

Die Funktion sollte aber besser wie folgt deklariert (und definiert) werden:

```
void fkt(const char *);
```

Bei Funktionen mit `const`-Adressen oder `const`-Referenzen als Parametern kann man beim Aufruf trotzdem Variablen als Argumente angeben, etwa für die durch:

```
void fkt(const char *);
```

deklarierte Funktion sind ebenfalls die Aufrufe:

```
char w[100], *p;
fkt(w);          // ok:
fkt(p);          // ok:
```

möglich! Hier wird für den Funktionsaufruf einfach “vergessen“, dass `w` und `p` eigentlich Adressen von variablen (und keinen konstanten) `char`’s sind — die über `w` bzw. `p` vermittelten Zeichen werden innerhalb der Funktion eben als `const` betrachtet und innerhalb der Funktion auch nicht abgeändert!

Ebenso verhält es sich mit der obigen Funktion `fkt2`:

```
void fkt2 ( int const & param);
```

Durch den Parametertyp `int const &` wird ermöglicht, dass als Argument ein konstantes `int` (oder auch ein variables, welches dann aber innerhalb der Funktion als konstant angesehen wird) angegeben werden kann.

Als Argument kann jetzt auch ein beliebiger Ausdruck vom Typ `int` angegeben werden, auch ein Ausdruck, der kein eigentliches “Objekt“ im Arbeitsspeicher repräsentiert, für welches die Referenz ein weiterer Name sein könnte. Hier wird vom Compiler ein temporäres Objekt mit dem Wert des Ausdrucks erzeugt und innerhalb der Funktion kann mittels des Parameters `param` auf dieses (temporäre) Objekt zugegriffen, dieses aber aufgrund der Konstantheit von `param` nicht verändert werden!

Somit sind folgende Aufrufe der Funktion völlig legitim:

```
int i,j;
int const k;
fkt2(k);    // ok
fkt2(i);    // ok: Variabilität von i wird vergessen
fkt2(7);    // ok: temp. Objekt mit Wert 7
fkt2(i+j);  // ok: temp. Objekt mit Wert von i+j
```

Konstante Referenzen als Funktionsparameter werden häufig verwendet, wenn einer Funktion “große“ Datenobjekte (etwa einige Kilobyte große Strukturen) eigentlich vom Wert her übergeben werden müssen — das hierbei notwendige Umkopieren großer Datenmengen aber vermieden werden soll:

- Ohne Referenzparameter:

```
struct A {
    ...
};          // einige Kilobyte grosse Struktur

void fkt(A); // Deklaration einer Funktion mit A-Parameter
```

```

int main()
{ A a;          // Variable vom Typ A, einige Kilobyte gross
  ...
  fkt(a);       // Aufruf von fkt mit Argument a
  ...
}

void fkt ( A b) // Definition der Funktion
{             // Parameter b ist einige Kilobyte gross
  ...
}

```

Hier wird (wie in C) Call by Value realisiert, d.h. beim Aufruf der Funktion `fkt` wird eine zur Funktion lokale Variable `b` vom Typ `A` erzeugt und diese mit dem Wert des Argumentes `a` initialisiert.

Während des Funktionsablaufes gibt es somit 2 Variablen vom Typ `A`, nämlich `a` aus dem Hauptprogramm und `b` in der Funktion. Beim Funktionsaufruf werden zudem alle Daten von `a` nach `b` kopiert!

Änderungen des Parameters `b` haben keine Auswirkungen auf das Argument `a`.

- Mit Referenzparameter:

```

struct A {
  ...
};          // einige Kilobyte grosse Struktur

void fkt(A const &); // Dekl. Funktion mit A-Referenz-Parameter

int main()
{ A a;          // Variable vom Typ A, einige Kilobyte gross
  ...
  fkt(a);       // Aufruf von fkt mit Argument a
  ...
}

void fkt ( A const &b) // Definition der Funktion
{
  ...
}

```

Hier wird das Argument beim Funktionsaufruf per Referenz übergeben, d.h. innerhalb der Funktion ist der Name `b` nur ein anderer Name für die Variable `a` des Hauptprogramms. Es ist somit nur einmal der für eine `A`-Variable notwendige Speicherbereich reserviert und zudem entfällt das Kopieren von einem Speicherbereich in einen anderen.

Innerhalb der Funktion kann die Variable `a` nicht abgeändert werden, da sie hier nur unter dem Namen `b` bekannt ist und `b` eine Referenz auf ein konstantes `A` ist!

Aus Anwendersicht sind also beide Realisierungen gleichwertig, die zweite bringt aber Speicherplatzersparnis und geringere Rechenzeit (Umkopieren entfällt!).

2.7.4 Funktionsergebnisse und `const`

Funktionsergebnisse sind (wenn es keine Referenzen sind) immer konstant, d.h. man kann sie zwar in komplexeren Ausdrücken weiterverwenden — man kann ihnen z.B. aber nichts zuweisen!

Bei Funktionen, welche eine Adresse als Funktionsergebnis liefern, muss man unterscheiden, ob das, worauf die Adresse verweist, konstant sein soll oder nicht (die Adresse als Funktionsergebnis selber ist immer konstant!):

```
int * fkt1(...);          // Fkt.-Resultat ist Adresse eines variablen int
int const * fkt2(...);    // Fkt.-Resultat ist Adresse eines konstanten int
```

Wendet man den Verweisoperator auf das Funktionsergebnis an (geht, da das Ergebnis in jedem Fall eine Adresse ist), kann man etwa dem über `fkt1` gelieferten Objekt etwas zuweisen, dem über `fkt2` gelieferten Objekt aber nicht:

```
*fkt1(...) = ...;        // ok
*fkt2(...) = ...;        // Fehler: *fkt2(...) ist int const
```

Funktionsergebnisse vom Referenztyp können auch als `const` deklariert (und entsprechend definiert) werden:

```
int const & fkt( ... );
```

Diese Funktion gibt in der Tat ein konstantes Objekt vom Typ `int` zurück und der Funktionsaufruf kann überall dort stattfinden, wo ein `int const`-Objekt verwendet werden dürfte — diesem Funktionsergebnis dürfte somit nichts zugewiesen werden (`fkt(...) = ...;`) und dürfte auch nicht inkrementiert werden (`fkt(...)`++) — eine Adresse hat das Funktionsergebnis gleichwohl und man kann einer entsprechenden Zeigervariablen die Adresse des Funktionsergebnisses zuweisen:

```
int const * p = &fkt(...); // p muss Zeiger auf int const sein!
```

Dies wäre bei einer gewöhnlichen Funktion mit `int`-Resultat beispielsweise nicht möglich!

Bei der Definition einer solchen Funktion mit einer konstanten Referenz als Funktionsresultat muss man (wie in Abschnitt 2.6 für Funktionen mit allgemeinen Referenzen als Ergebnis erläutert) sicherstellen, dass das per Referenz zurückgegebene Objekt nach der Beendigung der Funktion im Arbeitsspeicher noch vorhanden ist!

Betrachtet man folgende beiden Funktionsdefinitionen (`A` sei irgendein Typ):

```
A fkt1( A &a, ... ) { ...; return a; }
```

```
A const & fkt2( A &a, ... ) { ...; return a; }
```

so liefern beide ein konstantes `A` als Funktionsergebnis, `fkt1` nur als Wert (ohne Adresse) und `fkt2` als (konstantes) Objekt (mit Adresse).

Neben dieser für Anwender geringfügigen Differenz besteht für das System jedoch eine bedeutsamer Unterschied (insbesondere dann, wenn der Datentyp **A** ziemlich groß ist, etwa einige Kilobyte)!

Hierzu wollen wir uns veranschaulichen, was das System beim Aufruf und der Beendigung der Funktion **fkt1** im Arbeitsspeicher so alles anstellen muss:

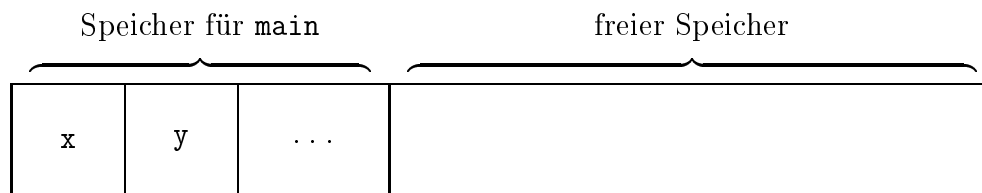
```
struct A {
    ...
};          // ziemlich gross

A fkt1( A &a, ...)    // Definition der Funktion
{ ...
    return a;
}

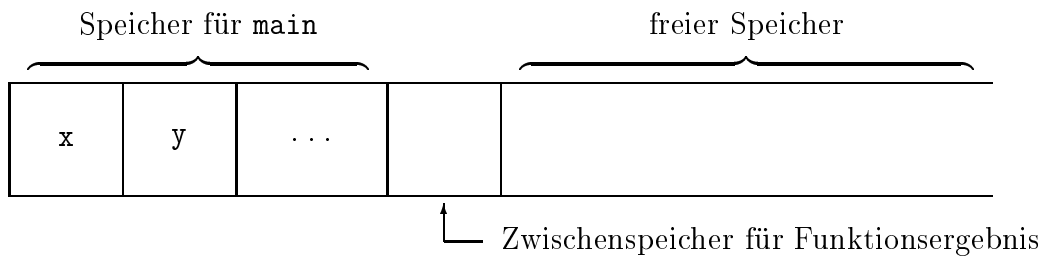
int main()
{ A x, y;
    ...
    y = fkt1(x,...);  // Funktionsaufruf
    ...
}
```

Den zeitlichen Ablauf wollen wir uns an obigem Beispiel ansehen:

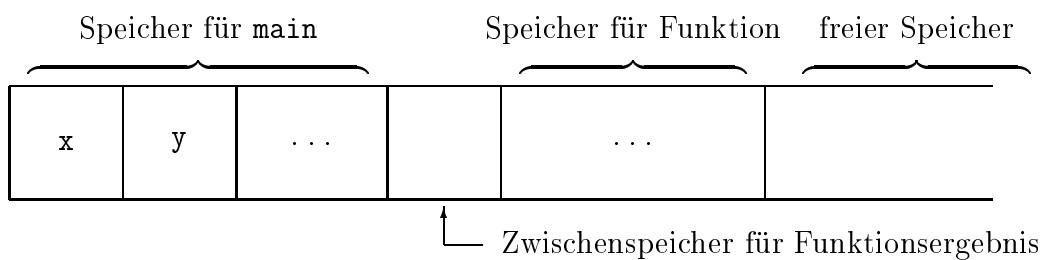
1. Vor dem Aufruf der Funktion **fkt1** hat das System bereits für das Hauptprogramm Speicherbereich reserviert, in dem u.a. auch die beiden (großen) Variablen **x** und **y** vom Typ **A** untergebracht sind:



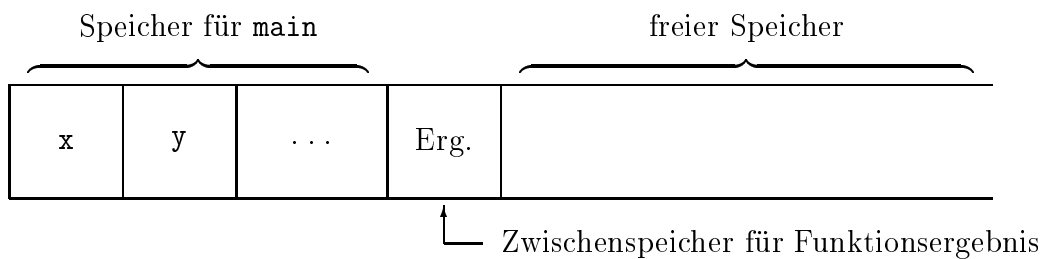
2. Beim Aufruf der Funktion **fkt1** sieht der Compiler anhand der Deklaration der Funktion, dass diese ein Ergebnis vom Typ **A** zurückgibt. Damit das Funktionsergebnis nach Beendigung der Funktion im Hauptprogramm weiterverwendet werden kann, legt das System, vor dem eigentlichen Funktionsaufruf, einen temporären Speicher an, in dem das Funktionsergebnis, nachdem es in der Funktion berechnet wurde, zwischengespeichert werden kann:



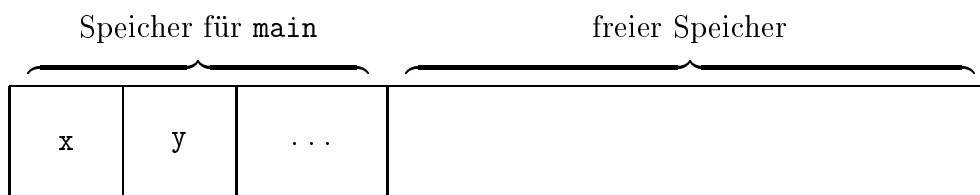
3. Dann wird der Speicherbereich für die Funktion reserviert (glücklicherweise ist der Parameter **a** eine Referenz — somit braucht kein neues **A**-Objekt angelegt zu werden, in welches der Wert des Funktionsargumentes **x** kopiert werden müsste!):



4. Die Funktion läuft ab und es wird die **return**-Anweisung erreicht! Der Wert des Ausdrucks hinter dem **return** wird in den dafür vorgesehenen Zwischenspeicher für das Funktionsergebnis (Erg.) kopiert, anschließend ist die Funktion beendet und der Speicherbereich der Funktion wird freigegeben:



5. Nachdem das Funktionsergebnis im aufrufenden Programmteil verwendet (hier der Variablen **y** zugewiesen) wurde, kann der temporäre Zwischenspeicher wieder freigegeben werden:



und die Situation ist wie vor dem Funktionsaufruf, nur dass die Variable `y` (und — da Referenzparameter — möglicherweise auch `x`) durch die Funktion einen neuen Wert erhalten hat!

Festzuhalten ist, dass es für gewöhnliche Funktionsergebnisse einen Zwischenspeicher gibt,

- der erzeugt wird, bevor die eigentliche Funktion aufgerufen wird,
- in den beim `return` das Funktionsergebnis kopiert wird (viel Kopierarbeit, wenn das Funktionsergebnis einen großen Typen hat!),
- der ein wenig länger “lebt“ als der Speicherbereich der Funktion,
- der, nachdem dessen Inhalt (also das Funktionsergebnis) im aufrufenden Programmteil verwendet wurde, freigegeben wird!

Das Funktionsergebnis gibt es somit mehrfach, einmal innerhalb der Funktion (als Ausdruck hinter dem `return`) und einmal im temporären Zwischenspeicher und das Funktionsergebnis muss (hier) zweimal kopiert werden, einmal von der Funktion in den Zwischenspeicher und einmal im Hauptprogramm bei der Zuweisung von diesem Zwischenspeicher in die Variable `x`.

Ganz schön viel Arbeit bei einer Funktion mit gewöhnlichem Ergebnis!

Auch in dieser Hinsicht sind Referenzen (konstant oder nicht) als Funktionsargumente ergonomischer. Es handelt sich hierbei ja nur um Namen für anderweitig abgespeicherte Objekte. Hier sind also allenfalls Namen (intern natürlich Adressen) zu verwalten und zurückzugeben — auch bei großen Datentypen!

2.7.5 Typumwandlung mittels `const_cast<typ>`

In den vorherigen Abschnitten haben wir gesehen, dass man mittels Referenzen auf `const` oder mittels Zeiger auf `const` auf Objekte (Variablen) zugreifen kann, die gar nicht konstant sind — sondern sich beim Zugriff über die Referenz bzw. den Zeiger wie `const` verhalten:

Ist `T` ein Typ, `obj` eine Variable von diesem Typ (`T obj;`), `cref` eine durch `obj` initialisierte Referenz auf `const` dieses Types (`T const &ref = obj;`) und `cptr` ein mit der Adresse von `obj` initialisierter Zeiger auf `T const` (also: `T const *cptr = &obj;`), so sind Referenz und Zeiger `const`, das eigentliche Objekt aber, auf welches Referenz und Zeiger verweisen, variabel.

```
T obj;                // obj ist variables Objekt vom Typ T
T const &cref = obj;    // cref ist Referenz auf T const,
                        // mit obj initialisiert
T const *cptr = &obj;  // cptr ist Zeiger auf T const,
                        // mit Adresse von obj initialisiert
cref = ... ;           // Fehler: cref ist Referenz auf T const
*cptr = ...;           // Fehler: cptr ist Zeiger auf T const
```

In diesem Fall kann man mittels des Operators `const_cast<Typ>(Operand)` bei Referenz oder Zeiger die Konstantheit “weg casten“, so dass die Konstantheit “vergessen“ wird und das tatsächliche Objekt über Referenz oder Zeiger auf `T const` doch noch abgeändert werden kann (hierbei muss `Typ` der entsprechende, nicht konstante Referenz- oder Zeigertyp sein!):

```
T &vref = const_cast<T&> (cref); // vref ist Referenz auf was Variables
                                // initialisiert mit Referenz auf const,
                                // der das const weggecastet worden ist!
vref = ...;                    // ok, vref ist Referenz auf was Variables

T * vptr;                      // vptr ist Zeiger auf was Variables
vptr = const_cast<T*> (cptr); // vptr mit Zeiger auf const initialisiert,
                                // dem das const weggecastet worden ist!
*vptr = ...;                   // ok, vptr ist Zeiger auf was Variables
```

Das Resultat dieses “Wegcastens“ der Konstantheit ist undefiniert, wenn das Objekt, auf welches die Referenz auf `const` bzw. der Zeiger auf `const` verweist, tatsächlich konstant ist! (In diesem Fall dürfte das “Wegcasten“ noch funktionieren — aber beim Versuch, das entsprechende Objekt — also eine Konstante — abzuändern, dürfte das Programm abstürzen!)

2.8 Standardparameter von Funktionen

Bei der Deklaration einer Funktion können die letzten Funktionsparametern mit einem Defaultwert (jeweils eine zur Compilierzeit vom Wert her feststehende Konstante) vorbesetzt werden (also von hinten beginnend, möglicherweise bis zum ersten Parameter der Funktion — es ist nicht möglich, einen Parameter mit einem Defaultwert zu versehen und für einen folgenden Parameter keinen Defaultwert anzugeben):

```
void fkt( int a1, double a2 = 2.7, int a3 = 4, double a4 = 3.1);
```

In diesem Beispiel hat der letzte Parameter (`a4` vom Typ `double`) den Defaultwert 3.1, der vorletzte Parameter (`a3` vom Typ `int`) den Defaultwert 4 und der drittletzte Parameter (`a2` vom Typ `double`) den Defaultwert 2.7.! (Da die Namen der Parameter bei der Deklaration nicht angegeben werden brauchen, könnte die Deklaration gleichwertig auch so:

```
void fkt( int, double = 2.7, int = 4, double = 3.1);
```

lauten!)

Beim Aufruf der Funktion können dann die letzten Funktionsargumente (und dann auch die trennenden Kommata) fehlen (wiederum von hinten beginnend und keins zwischendurch überspringend — es ist beispielsweise also nicht möglich, das drittletzte Argument auszulassen und das vorletzte oder letzte nicht auszulassen!).

Die wie oben deklarierte Funktion könnte somit mit unterschiedlicher Argumentzahl aufgerufen werden:

```
fkt(i,x,j,y); // 4 Argumente, a1 bekommt Wert von i,
              //                a2 bekommt Wert von x,
```

```

//          a3 bekommt Wert von j,
//          a4 bekommt Wert von y.
fkt(i,x,j); // 3 Argumente, a1 bekommt Wert von i,
//          a2 bekommt Wert von x,
//          a3 bekommt Wert von j,
//          a4 bekommt Defaultwert 3.1
fkt(i,x);   // 2 Argumente, a1 bekommt Wert von i,
//          a2 bekommt Wert von x,
//          a3 bekommt Defaultwert 4,
//          a4 bekommt Defaultwert 3.1
fkt(i);     // 1 Argument, a1 bekommt Wert von i,
//          a2 bekommt Defaultwert 2.7
//          a3 bekommt Defaultwert 4,
//          a4 bekommt Defaultwert 3.1

```

Ohne Argumente kann diese Funktion nicht aufgerufen werden (erster Parameter hat keinen Defaultwert!).

Haben alle Parameter einer Funktion einen Defaultwert, so kann die Funktion dann ganz ohne Argumente aufgerufen werden!

Hauptanwendung solcher Standardwerte sind Funktionen, die eine Reihe gewöhnlicher Parameter besitzen (sollten dann die ersten Parameter in der Parameterliste sein) und deren Ablauf durch gewisse Steuerparameter beeinflusst werden kann (letzten Parameter in der Parameterliste). Diesen Steuerparametern gibt man dann i. Allg. Defaultwerte, in denen sich der defaultmäßige Ablauf der Funktion widerspiegelt.

Beispiel: Umwandlung einer Zeichenkette (genauer: die am Anfang der Zeichenkette stehenden Ziffern) in einen ganzzahligen Wert:

```
int itoa( char const *string, int base = 10);
```

Eine “umzuwandelnde” Zeichenkette ist immer (als erstes Argument) anzugeben — mit dem zweiten Argument kann man festlegen, ob die Zeichenkette dezimal (standardmäßig) oder hexadezimal oder oktal oder dual ... aufzufassen ist:

```

char str[] = "110";
... itoa(str);           // dezimal, entspricht itoa(str,10)
... itoa(str,2);         // dual
... itoa(str,8);         // oktal
... itoa(str,16);        // hexadezimal
...

```

(Funktionsdefinition als Übung!)

Wie bereits erwähnt, müssen derartige Standardparameter bei der Funktionsdeklaration (und nicht bei der Funktionsdefinition) stehen — üblicherweise werden sie bei der Funktionsdeklaration in einer Headerdatei angegeben, so dass in allen Quelltexten die gleichen Standardparameter vereinbart sind.

Defaultwerte kann man zu Parametern von beliebigem Typ, auch von Referenz- oder Adresstypen — const oder auch nicht, angeben:

```
void fkt( char const *a= "hello", int &b= i);
```

doch Vorsicht: lässt man (wie üblich) die Namen der Parameter einfach fort:

```
void fkt( char const *= "hello", int &= i);
```

so erhält man hier Fehlermeldungen, da *= und &= jeweils als ein Token (Operatoren für multiplikative Zuweisung und Bit-Und-Zuweisung) aufgefasst werden!

Richtig müsste die Deklaration lauten:

```
void fkt( char const * = "hello", int & = i);
```

(Leerzeichen zwischen * bzw. & und = beachten!)

2.9 inline-Funktionen

Üblicherweise wird, wenn eine Funktion aufgerufen wird, der Programmfluss im aufrufenden Programmteil unterbrochen, zur Funktion “gesprungen“, diese ausgeführt und anschließend wird im aufrufenden Programmteil hinter dem Funktionsaufruf fortgefahren.

Vor dem “Sprung“ zur Funktion (siehe Abschnitt 2.7.4) müssen gewisse Informationen (Befehlszähler, Rücksprungadresse,...) gesichert, Platz fürs Funktionsergebnis geschaffen und Speicherbereich für die Funktion (Parameter- und sonstige lokale Variablen) geschaffen werden. Bei Beendigung der Funktion muss das in der Funktion berechnete Funktionsergebnis in den Zwischenspeicher kopiert, der Speicherbereich der Funktion freigegeben und der Zustand vor dem Funktionsaufruf (Befehlszähler, ...) wiederhergestellt werden.

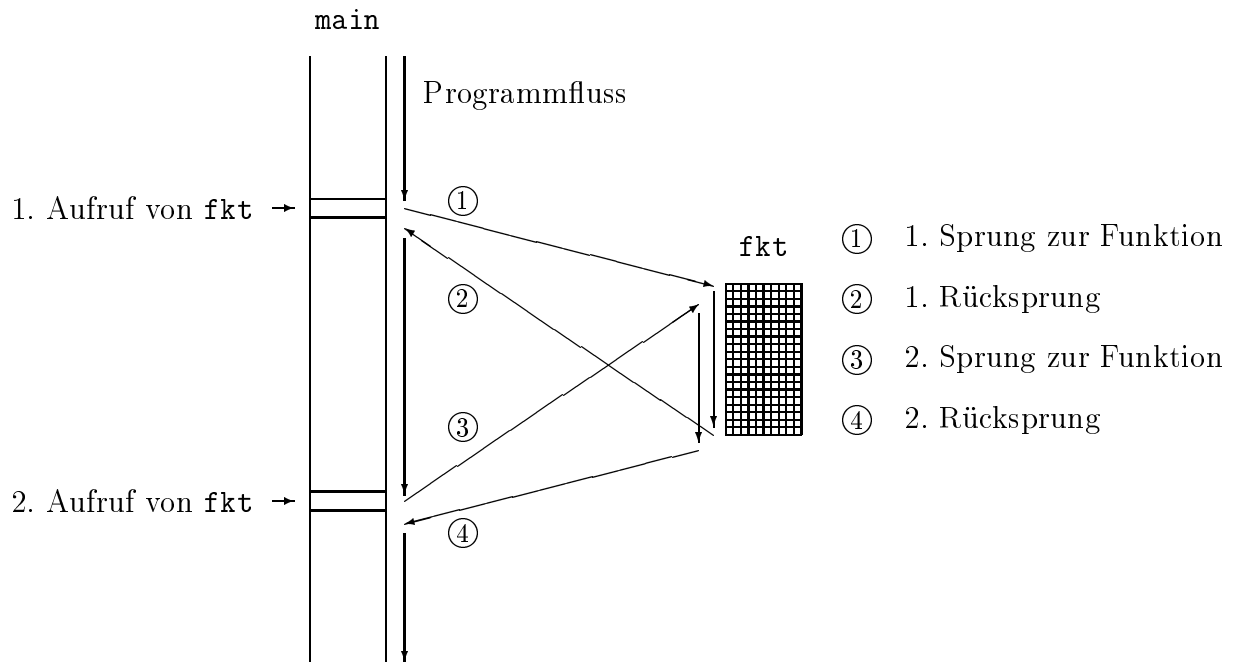
Dies geschieht bei jedem Aufruf der Funktion, wobei die Befehle, welche während der Funktion ausgeführt werden müssen, nur einmal im ausführbaren Programm vorhanden sind.

Zur Verdeutlichung:

```
void fkt(...)
{...}
```

```
int main()
{
    ...
    fkt(...);
    ...
    fkt(...);
    ...
}
```

in obigem Beispiel wird ein- und dieselbe Funktion **fkt** zweimal im Hauptprogramm aufgerufen — jedesmal ist ein “Funktionssprung“ notwendig:

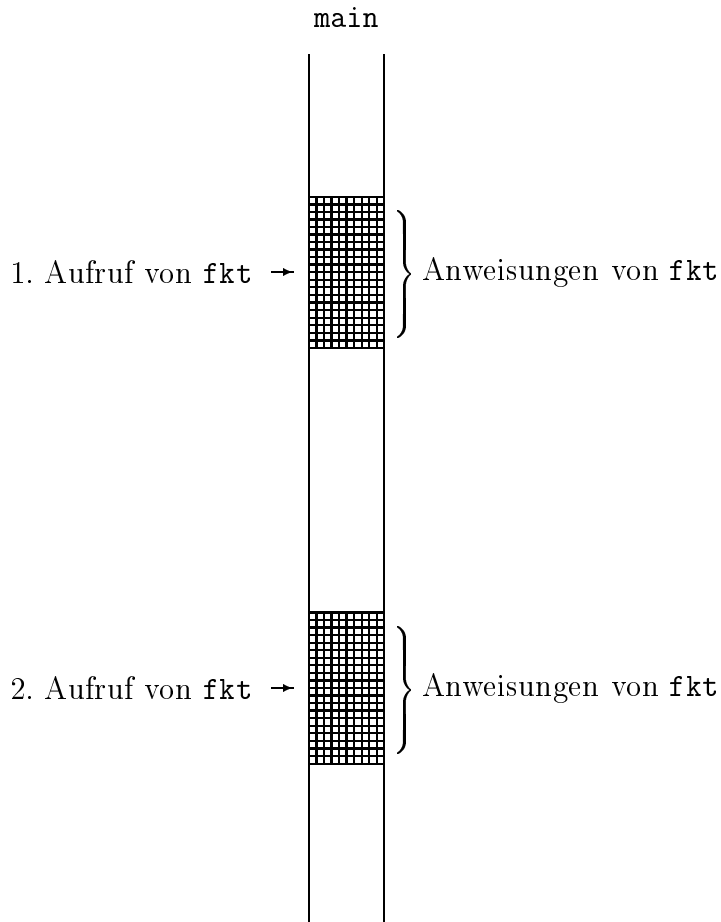


Schreibt man vor eine Funktionsdefinition das Schlüsselwort `inline`, so wird der Compiler gebeten, diese Funktion nicht über einen wie oben beschriebenen Funktionssprung zu realisieren, sondern die für die Funktion notwendigen Anweisungen direkt in den aufrufenden Programmteil "einzubauen":

```
inline void fkt(...)
{...}

int main()
{
    ...
    fkt(...);
    ...
    fkt(...);
    ...
}
```

Hier wird der Compiler gebeten, die Funktionsaufrufe wie folgt umzusetzen:



Wie man sieht, wird hierdurch — falls der Compiler der Bitte nachkommt — das ausführbare Programm größer, da für jeden Funktionsaufruf die zur Funktion gehörenden Anweisungen jeweils an Ort und Stelle eingebaut werden.

Dafür hat das System zur Laufzeit beim Aufruf der Funktion weniger Verwaltungsaufwand, so dass das Programm eventuell schneller läuft.

Sinnvoll angewendet wird diese Technik meist nur bei “kleinen” Funktionen, für die man in C vielleicht Makros mit Argumenten verwendet hätte:

– C-Makro:

```
#define MAX(A,B) ( (A) > (B) ? (A) : (B) )
```

– C++-Inline-Funktion:

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

Der Vorteil solcher `inline`-Funktionen ist, dass es sich um echte Funktionen handelt, d.h.

- der Compiler führt eine Typüberprüfung der Argumente durch und er veranlasst ggf. Typumwandlungen,
- solche Funktionen haben genau die selben Nebeneffekte wie richtige Funktionen (man betrachte beispielsweise folgenden Aufruf des Makros: `MAX(i++,j++)`)

Zu beachten ist

- dass, wenn man eine Funktion bei ihrer Definition als `inline` vereinbart (unabhängig davon, ob der Compiler sie als `inline` umsetzt oder nicht), vor einer Verwendung der Funktion die komplette Funktionsdefinition im selben Quelltext wie der Aufruf stehen muss (eine Deklaration der Funktion ist nicht möglich!),
- dass auf eine in einem Quelltext definierte `inline`-Funktion aus anderen Quelltexten heraus nicht zugegriffen werden kann (der Compiler muss ja die komplette Funktion und nicht nur deren Typ kennen!),
- dass ein- und dieselbe `inline`-Funktion in jedem zu einem Projekt gehörenden Quelltext aufs neue (wörtlich übereinstimmend mit gleicher Bedeutung) erneut definiert werden kann.

Somit ist es legitim und üblich, die komplette Definition einer `inline`-Funktion in eine Headerdatei zu schreiben und diese überall wo notwendig einzubinden. (Man muss nur sicherstellen, dass die Definition der `inline`-Funktion nicht mehrfach in ein- und demselben Quelltext steht — also mehrfaches Einbinden der Headerdatei verhindern!)

2.10 Überladen von Funktionen

Eine Funktion wird durch ihren Namen, ihren Rückgabetypen und ihre Signatur eindeutig beschrieben.

Der Rückgabetypp der Funktion legt fest, in welchem Zusammenhang (komplexerer Ausdruck) die Funktion aufgerufen werden kann:

```
int fkt1(...);
double fkt2(...);
int i, a[100];

i = a [fkt1(...)];    // ok: Ergebnis von fkt1 ist int
i = a [fkt2(...)];    // Fehler: Ergebnis von fkt2 ist double
```

Die Signatur einer Funktion ist die Anzahl, der Typ und die Reihenfolge der Argumente. Anhand der Signatur entscheidet der Compiler, ob die Funktion ordnungsgemäß aufgerufen wird (richtige Anzahl und Typ der Argumente) — ggf. werden die Argumente, falls möglich, in die Typen der Parameter umgewandelt:

```
void fkt(double); // Signatur: ein double-Parameter
int i;
double x;
...
fkt(x,i);          // Fehler: falsche Argumentzahl
fkt("hallo");      // Fehler: Typ des Argumentes falsch
fkt(x);            // ok
fkt(i);            // ok: Wert von i wird in double umgewandelt
```

Man kann in C++ mehrere Funktionen mit dem gleichen Namen definieren (und deklarieren), wenn sie sich nur in der Signatur (wesentlich) unterscheiden:

```
void swap ( int &a, int &b)
{ int tmp = a;
  a = b;
  b = tmp;
}

void swap ( double &a, double &b)
{ double tmp = a;
  a = b;
  b = tmp;
}
```

Hier sind nun zwei Funktionen definiert — beide mit dem Namen `swap` — die eine dient zur Vertauschung des Inhalts zweier `int`-Variablen und die andere zur Vertauschung von `double`'s.

Der Compiler sucht beim Aufruf der Funktion anhand der tatsächlichen Argumente die passende Funktion (mit diesem Namen):

```
int i,j;
double x,y;

swap (i,j);    // ok: swap (int&, int&)
swap (x,y);    // ok: swap(double,double)
swap (x,i);    // Fehler
```

Findet er keine genau passende Funktion, so versucht er, nach gewissen Regeln (siehe Abschnitt 2.10.1) die Typen der Argumente so umzuwandeln, dass sie auf die Parametertypen einer ihm bekannten Funktion passen.

Bei der Überladung muss die Signatur schon "wesentlich" verschieden sein, zwischen einem Typen `T` und einer Referenz auf `T`, also `Typ T&`, kann der Compiler beim Aufruf der Funktion nicht unterscheiden:

```
void fkt( int );      // Parameter: int
void fkt( int&);      // Parameter: Referenz auf int
int i;
fkt(i);              // Fehler: fkt(int) oder fkt(int&) ???
```

Eine solche Funktionsüberladung ist ein Fehler!

Wie in Abschnitt 2.6 bereits erläutert, kann einer Funktion, welche eine Referenz auf `const` (oder ein Zeiger auf `const`) als Parameter hat, durchaus etwas Variables als Argument übergeben werden — dieses Variable wird innerhalb der Funktion als konstant angesehen!

Formal gehört eine derartige Konstantheit aber zu den Unterscheidungsmerkmalen eines Parametertypes und hat somit Einfluss auf die Signatur. Somit ist auch folgende Funktionsüberladung möglich:


```
void fkt( char *);           // Parameter vom Typ char *
void fkt( char const *);    // Parameter vom Typ char const *
```

Anhand des tatsächlichen Funktionsargumentes (konstant oder nicht) wird vom Compiler die passende Funktion gewählt:

```
char w[100];
fkt(w);           // 1. Version: fkt(char *);
fkt("hallo");     // 2. Version: fkt(char const *);
```

2.10.1 Typumwandlung bei überladenen Funktionen

Passen bei einer Funktion die tatsächlichen Argumente nicht genau auf die bei den Deklarationen angegebenen Parametertypen, so versucht der Compiler, die “am besten passende” Funktion zu ermitteln und die Typen der Argumente des Aufrufes in die Parametertypen der Funktion umzuwandeln.

Gibt es keine “passende” Funktion, ist der Funktionsaufruf ein Fehler.

Gibt es mehrere unterschiedliche “gleich gut passende” Funktionen, so ist der Funktionsaufruf ebenfalls ein Fehler!

Das Ermitteln der zu einem konkreten Aufruf “am besten passenden Funktion” geschieht in mehreren Schritten:

1. Zunächst werden alle irgendwie zum Aufruf “passenden” Funktionen ermittelt. Das sind alle (zum Aufruf gleichnamigen) Funktionen, deren Parameterliste so sind, dass die tatsächlichen Argumente in die Typen der Parameter umgewandelt werden können.
2. Zu jeder “passenden” Funktion und jedem Argument des Aufrufes wird hierbei ermittelt, wie dieses Argument in den Typ des Parameters dieser passenden umgewandelt werden kann. Umgewandelt werden kann ein Argument in den Typ des Parameters durch:

- (a) Eine Standardumwandlungssequenz.

Eine Standardumwandlungssequenz kann (in dieser Reihenfolge) aus folgenden vier einzelnen Umwandlungen bestehen, wobei jede einzelne Umwandlung auch fehlen darf:

- *Lvalue-Transformation*, etwa eine Umwandlung eines Feldnamens in eine Adresse, Umwandlung eines Funktionsnamens in die Adresse einer Funktion.

Eine solche *Lvalue-Transformation* wird als trivialer Umwandlungsschritt aufgefasst und als *exact match* (exakte Übereinstimmung) bewertet.

- *Promotion*, also Umwandlung von `char`, `bool` und `short` nach `int` (ggf. auch für `unsigned`), sowie Umwandlung von `float` nach `double` bzw. von `double` nach `long double`.

Eine derartige Umwandlung wird als *promotion* (Aufwertung) bewertet.

- *Conversion*, also eine eigentliche Umwandlung wie etwa ganzzahlig nach `double` oder umgekehrt, Umwandlung zwischen ganzzahligen Typen, Umwandlung von Adressen, Umwandlung eines Objektes einer abgeleiteten Klasse in eine Basisklasse etc. .
Eine derartige Umwandlung wird als *conversion*, also eine eigentliche Umwandlung bewertet.
- *Qualifiction Adjustment*, also das Hinzufügen einer Qualifikation (etwa Umwandlung von `int *` nach `const int *`). Ein *qualifiction adjustment* wird auch als *exact match* (exakte Übereinstimmung) bewertet.

Ist bei einer solchen Standardumwandlungssequenz eine *Conversion* beteiligt, so wird die ganze Sequenz als *conversion* bewertet, ist ansonsten eine *Promotion* beteiligt, so wird die ganze Sequenz als *promotion* bewertet, ansonsten wird die gesamte Sequenz als *exact match* bewertet.

Eine mit *exact match* bewertete Standardumwandlungssequenz ist “besser” als eine mit *promotion* bewertete und eine mit *promotion* bewertete Sequenz ist besser als eine mit *conversion* bewertete.

(b) Eine benutzerdefinierte Umwandlungssequenz.

Eine benutzerdefinierte Umwandlungssequenz besteht aus einer (optionalen) anfänglichen Standardumwandlungssequenz, einer anschließenden benutzerdefinierten Umwandlung (eine benutzerdefinierte Umwandlung ist die Anwendung eines Konstruktors oder eines Konversionsoperators — wird später behandelt) und einer (optionalen) anschließenden Standardumwandlungssequenz.

Die anfängliche Standardumwandlungssequenz dient zur Umwandlung des Argumentes des Funktionsaufrufes in den Parametertyp der benutzerdefinierten Umwandlung und die anschließende Standardumwandlungssequenz zur Umwandlung des Ergebnisses der benutzerdefinierten Umwandlung in den Parametertypen der “passenden” Funktion.

Eine benutzerdefinierte Umwandlungssequenz *Sequenz1* ist genau dann besser als eine andere benutzerdefinierte Umwandlungssequenz *Sequenz2*, wenn an beiden Sequenzen genau dieselbe benutzerdefinierte Umwandlung beteiligt ist und die zur *Sequenz1* gehörende, anschließende Standardumwandlungssequenz im Sinne von 2a besser ist, als die zur *Sequenz2* gehörende — ansonsten sind sie gleichwertig.

(c) Einer . . .-Umwandlung.

Dies bedeutet, dass das Argument des Aufrufes in die variable Argumentliste . . . der “passenden” Funktion fällt (falls diese eine solche hat!).

Eine Standardumwandlungssequenz ist besser als jede benutzerdefinierte Umwandlungssequenz und jede . . .-Umwandlung, eine benutzerdefinierte Umwandlungssequenz ist besser als jede . . .-Umwandlung.

3. Unter den “passenden” Funktionen wird die “am besten passende” ausgewählt — das ist diejenige, bei der die Umwandlung eines jeden Argumentes des Aufrufs in den zugehörigen Parameter der Funktion im Sinne von 2 nicht schlechter ist

als bei allen anderen passenden Funktionen und wo es mindestens ein Argument gibt, dessen Umwandlung besser ist als bei jeder anderen Funktion.

(Im Zusammenhang mit Templates und in einigen Sonderfällen sind diese Regeln noch viel subtiler! Interessenten werden hier auf die Ausführungen des ANSI-Standardisierungs-Komitees verwiesen!)

2.10.2 Überladen und Gültigkeitsbereich

Funktionen können nur im gleichen Gültigkeitsbereich überladen werden, ansonsten spricht man von “Überdeckung” und die überdeckte Funktion kann nicht ohne weiteres aufgerufen werden, Beispiel:

```
...
void f(int);      // global
...
void g()
{
    void f(double); // lokal, globales f(int) hat anderen
                    // Gültigkeitsbereich

    f(1);          // lokales f(double) wird aufgerufen, hierbei wird
                    // das Argument von int nach double umgewandelt

    ::f(1);        // Gültigkeitsbereichsaufloesung, globales f(int)
                    // wird aufgerufen
    ...
}
...
```

2.11 Ausnahmebehandlung

Größere Softwarepakete haben die Eigenschaft, dass darin unterschiedliche Funktionen aus unterschiedlichen, getrennt entwickelten Modulen zusammenarbeiten müssen. Häufig ist es so, dass in einem Teil einer Anwendung auf eine Funktionalität (Funktion) aus einem (Bibliotheks-)Modul zugegriffen werden muss — die Anwendung ruft die entsprechende Funktion auf und diese Funktion arbeitet dann irgendwie mit den Daten (Objekten oder klassischen Daten) der Anwendung.

Stellt nun die Funktion während ihres Ablaufes eine “unübliche Situation” (Ausnahme, *Exception* — etwa Speichermangel, falsche Daten, ...) fest, so dass die Funktion nicht mehr in der Lage ist, ordnungsgemäß abzulaufen und ihre eigentliche Bestimmung zu erfüllen, so kann die Funktion auf unterschiedliche Art- und Weise auf diese fehlerhafte Situation reagieren:

1. Fehlermeldung ausgeben und das Programm abbrechen.

Diese Reaktion der Funktion ist in vielen Fällen zu drastisch. Das Dilemma ist, dass die Funktion den Fehler nur feststellen, ihn aber i. Allg. selbst nicht

beheben kann (die Funktion weiß ja nicht, wie etwa die fehlerhaften Daten zustandegekommen sind!). Der Aufrufer der Funktion wäre zwar in der Lage, auf den Fehler zu reagieren, kann aber, da er ja den Funktionsablauf nicht so genau kennt, im Voraus den Fehler nicht erkennen (sonst hätte er wahrscheinlich die Funktion auch gar nicht so aufgerufen) und muss die Fehlererkennung der Funktion überlassen — mit dem Effekt, dass er hier nicht mehr zur Behebung des Fehlers kommt, da das Programm von der Funktion bereits abgebrochen wurde!

2. Einen Fehlerstatus an den Aufrufer als Funktionsergebnis zurückgeben oder in einer globalen Fehlervariablen setzen.

Gut an dieser Lösung ist, dass die hier eine Trennung von Fehlerfeststellung (in der Funktion) und Behandlung des Fehlers (beim Aufrufer) erfolgt.

In diesem Fall muss der Aufrufer dieser Funktion aber nach jedem Aufruf das Funktionsergebnis bzw. die globale Fehlervariable abtesten, so dass der eigentliche Algorithmus, dessen Lösung durch das Programm geliefert werden soll, ständig von Fehlerabfragen unterbrochen wird. (Der Programmierer kann sich also nicht auf die eigentliche Lösung seines Problems konzentrieren, sondern muss ständig auch an die Behandlung von allen erdenklichen Fehlern denken.)

C++ bietet hier ein neues Konzept:

- Ein Teil des Lösungsalgorithmus (inklusive von Funktionsaufrufen) wird in geschweifte Klammern eingeschlossen und dieser Block wird mit dem Schlüsselwort **try** versehen (sog. **try**-Block):

```
try {
    /* Loesungsalgorithmus
       ggf. mit
       Funktionsaufrufen */
    ...
}
```

- Wird innerhalb dieses geklammerten Teils eine Ausnahmesituation festgestellt, so wird mit einer **throw**-Anweisung ein Objekt (**ausdruck**: Variable, Konstante eines gewissen Types, *Ausnahmeobjekt* oder *Ausnahme* genannt) “ausgeworfen“:

```
if ( sonderfall) throw ausdruck;
```

Das System bricht hierauf “geordnet“ die Abarbeitung des ganzen **try**-Blockes ab, wobei alle innerhalb dieses Blockes — auch bei zwischenzeitlich erfolgten Funktionsaufrufen — erzeugten automatischen Variablen wieder zerstört werden (der Variablenstack wird auf den Zustand zu Beginn des **try**-Blockes “zurückgespult“), der Programmfluss wird unmittelbar hinter dem **try**-Block fortgesetzt und das einzige Objekt, was noch vom **try**-Block übrig ist, ist das mittels **throw** ausgeworfene!

- Hinter dem **try**-Block steht (mindestens) eine **catch**-Anweisung (i. Allg. aber mehrere):

```
try {  
    ...  
}  
catch ( typ1 name)  
{  
    ...  
}  
catch ( typ2 name)  
{  
    ...  
}  
...
```

Eine derartige `catch (typ name)`-Anweisung “fängt” ein ausgeworfenes Fehlerobjekt “ab” — aber nur, wenn es vom beim `catch` angegebenen Typ ist — und der zum `catch` gehörende Anweisungsteil `{ ... }` wird ausgeführt.

Eine Folge von solchen `catch`-Anweisungen ähnelt in ihrer Funktion einer `switch`-Anweisung: Der Typ des ausgeworfenen “Fehlerobjektes” wird der Reihe nach mit allen bei den `catch`-Anweisungen angegebenen Typen verglichen und bei der ersten Übereinstimmung wird das entsprechende `catch` “ausgewählt” und die zu diesem `catch` gehörenden, in den geschweiften Klammern stehenden Anweisungen (und nur diese) durchgeführt. Anschließend geht es mit dem Programmfluss hinter der letzten `catch`-Anweisung weiter.

Die Schreibweise und Funktionalität eines einzelnen `catch (typ name) { ... }` ähnelt der Definition einer Funktion mit einem Parameter `name` vom angegebenen Typ `typ`. Sollte dieses `catch` “ausgewählt” werden, so bekommt die “Parameter-Variable” `name` den Wert des ausgeworfenen Fehlerobjektes (passt ja vom Typ her!) und der Anweisungsteil des `catch` wird ausgeführt, wobei auf die Parameter-Variable `name` zugegriffen werden kann!

Wurde der ganze `try`-Block ordnungsgemäß durchlaufen und dabei kein Fehler ausgeworfen, so sind sämtliche, anschließende `catch`-Anweisungen wirkungslos, d.h. der Programmfluss ignoriert diese und es geht unmittelbar hinter der letzten `catch`-Anweisung im Programm weiter.

Diese Technik bietet dem Entwickler einer Bibliothek und deren Anwender eine neue Art der Fehlerbehandlung:

- Der Entwickler einer Bibliotheksfunktion sorgt in seiner Funktion dafür, dass, wenn eine Ausnahmesituation (welche die Funktion selber nicht beheben kann) erkannt wird, ein Fehlrojekt mittels `throw` ausgeworfen wird.
- Der Anwendungsentwickler kann, indem er ihn in einen `try`-Block schreibt, seinen Algorithmus als ganzes direkt entwickeln, ohne immer wieder an die Behandlung von Fehlern denken zu müssen.

Im Anschluss an den `try`-Block muss der Anwendungsentwickler sich dann um die während des Algorithmus möglicherweise aufgetretenen Fehler kümmern.

2.11.1 Geschachtelte try-Blöcke

Man kann mehrere `try`-Blöcke schachteln:

```
...
try {    // Anfang try-Block1
    ...
    try {    // Anfang try-Block2
        ...
    }        // Ende try-Block2
    catch (int i) { ... }    // Abfangen von Fehlern aus Block2
    catch (char c) { ... }
    ...
}        // Ende try-Block1
catch ( float f) { ... }    // Abfangen von Fehlern aus Block1
catch ( double *dp) { ... }
...
```

Hiermit ist also eine mehrschichtige Fehlerbehandlung möglich!

Wird ein Ausnahmeobjekt am Ende eines `try`-Blockes nicht abgefangen, so wird das Objekt ggf. an den übergeordneten `try`-Block weitergeleitet, d.h. auch der übergeordnete `try`-Block wird sofort verlassen und die dahinter befindlichen `catch`-Anweisungen auf passenden Typ überprüft und ggf. durchgeführt.

Wird ein Ausnahmeobjekt von keinem `catch` “abgefangen“, so wird i. Allg. das Programm beendet (vgl. Abschnitt 2.11.5).

2.11.2 catch (...)

Die `catch`-Anweisung:

```
catch ( ... ) { /* irgendwas */ }
```

(hier müssen innerhalb der runden Klammern zu `catch` wirklich drei Punkte hintereinander stehen) ist als “*default*“-`catch` zu verstehen, d.h. dieses `catch` fängt jedes erdenkliche Fehlerobjekt von jedem beliebigen Typen ab, wobei bei der Bearbeitung dieser Ausnahme nicht auf das konkrete Fehlerobjekt zugegriffen werden kann.

Wenn überhaupt, steht ein solches `catch` am Ende der `catch`-Anweisungen im Anschluss an einen `try`-Block!

2.11.3 Ausnahmen “weiterreichen“

Bei geschachtelten `try`-Blöcken kann man ein Ausnahmeobjekt des inneren `try`-Blockes durch ein `catch` abfangen, im Anweisungsteil des `catch` gewisse Aktionen durchführen und mittels `throw`; (ohne Argument) dasselbe Fehlerobjekt erneut auswerfen (gewöhnlich wird durch das “Abfangen“ eines `catch` das Fehlerobjekt ganz “entfernt“).

Hierdurch hat man die Möglichkeit, auf ein- und denselben Fehler an mehreren Stellen zu reagieren:

```

...
try {    // Anfang try-Block1
    ...
    try {    // Anfang try-Block2
        ...
        if (sonderfall) throw 7;    // int-Fehler auswerfen
        ...
    }        // Ende try-Block2
    catch (int i)    // Abfangen von Fehlern aus Block2
    { ...        // gewisse Aufräumarbeiten bei int-Fehler durchführen
        throw;    // Fehlerobjekt erneut auswerfen, weiterreichen
    }
    ...
}        // Ende try-Block1
catch ( int f) // Abfangen von Fehlern aus Block1
{
    ... // weitere Aufräumarbeiten fuer
        // gleichen Fehler durchführen
}
...

```

2.11.4 Unterscheidung von Ausnahmen

Natürlich kann man unterschiedliche Fehler durch etwa unterschiedliche Werte des Fehlerobjektes (von einem gewissen Typ) darstellen:

```

...
try {
    ...
    if ( dies ) throw 7;
    ...
    if ( jenes ) throw 12;
    ...
    if ( sonstwas ) throw 25;
    ...
}
catch (int i)
{ switch (i)
  { ...
    case 7: ...; break;    // dies
    ...
    case 12: ...; break;    // jenes
    ...
    case 25: ...; break;    // sonstwas
    ...
  }
}

```

Üblicherweise werden jedoch unterschiedliche Fehler durch unterschiedliche (selbst-definierte) Datentypen von Fehlerobjekten dargestellt. (Im Fehlerobjekt kann dann genauere Information untergebracht werden, was nun zu diesem speziellen Fehler geführt hat.)

Beispielfragment:

```
...
// Fehlertyp, der einen Fehler beim Lesen anzeigt!
struct Lesefehler {
    ...
};

// Fehlertyp, der Speichermangel anzeigt!
struct Speichermangel {
    ...
};

...

try {
    int zahl, *intfeld;
    ...
    if ( scanf("%d", &zahl) != 1 )
    {
        Lesefehler lesefehler;
        throw lesefehler;
    }
    ...
    if ( (intfeld = (int *) malloc ( zahl * sizeof(int) )) == NULL)
    {
        Speichermangel speichermangel;
        throw speichermangel;
    }
    ...
}
catch ( Lesefehler err)
{ ... }
catch ( Speichermangel err)
{ ... }
...
```

Mit Mitteln der OOP (insbes. Vererbung, siehe Abschnitt 7.10) kann man natürlich von Fehlertypen neue Fehlertypen ableiten und somit auch Hierarchien von Fehlertypen aufbauen.

2.11.5 Nicht abgefangene Ausnahmen

Wird eine Ausnahme geworfen, aber nicht von einem `catch` abgefangen, so wird die Bibliotheksfunktion `void std::terminate();` aufgerufen, welche (auf UNIX-

Systemen mit einem Speicherabzug) das Programm abbricht.

Man kann selbst die Reaktion auf nicht abgefangene Ausnahmen einstellen. Hierzu muss man selbst eine Funktion des gleichen Typen wie `terminate` schreiben, also etwa:

```
void nicht_abgefangen() { ... }
```

Dem System muss dann noch mitgeteilt werden, dass diese Funktion beim Auftreten einer nicht abgefangenen Ausnahme aufgerufen werden soll. Dies geschieht durch folgenden Aufruf der Bibliotheksfunktion:

```
set_terminate(nicht_abgefangen);
```

wobei als Argument der Name der selbstgeschriebenen Funktion zur Behandlung nicht abgefangener Ausnahmen angegeben werden muss. Die Funktion `set_terminate` ist in der Headerdatei `<exception>` deklariert.

Dort ist auch mittels `typedef` folgender Funktionstyp vereinbart:

```
typedef void (*terminate_handler)();
```

(Der Name `terminate_handler` bezeichnet somit folgenden Typen: Zeiger auf eine Funktion mit keinem Argument und `void` Rückgabe.) Die Funktion `set_terminate` ist dann wie folgt deklariert:

```
terminate_handler set_terminate ( terminate_handler);
```

Diese Funktion erhält nicht nur die neue Funktion zur Behandlung nicht abgefangener Ausnahmen als Argument, sondern liefert auch die Adresse der bisher eingestellten Funktion zur Behandlung nicht abgefangener Ausnahmen als Funktionsergebnis. Dieses Ergebnis könnte einem entsprechenden Funktionszeiger zugewiesen werden.

2.11.6 Ausnahmen in der Schnittstelle einer Funktion

Die Typen der Ausnahmeobjekte, welche durch eine Funktion möglicherweise ausgeworfen werden, muss strenggenommen dem Anwender der Funktion bekannt sein, damit er den Funktionsaufruf in einen entsprechenden `try`-Block schreiben kann, hinter dem er die möglicherweise aufgetretenen Ausnahmen abfängt.

Man kann bei der Deklaration und dann auch bei der Definition einer Funktion hinter der die Parameterliste abschließenden runden Klammern das Schlüsselwort `throw` und dann (ebenfalls in runden Klammern) eine Liste der durch diese Funktion (oder durch von dieser Funktion aufgerufene weitere Funktionen) möglicherweise ausgeworfenen Ausnahmen angeben:

```
int fkt(double, int *) throw ( Lesefehler, Speichermangel);
```

hiermit ist eine Funktion mit Namen `fkt`, `int`-Rückgabe, einem `double`- und einem `int`-Adress-Wert als Argument deklariert, welche möglicherweise Ausnahmen vom Typ `Lesefehler` oder `Speichermangel`, aber keine anderen Ausnahmen, auswirft. Bei einer solchen (hoffentlich auch dokumentierten) Deklaration (und entsprechenden Definition) der Funktion kann der Anwender dann Maßnahmen ergreifen, um ggf. derartige Fehler abzufangen.

Eine Deklaration (und Definition):

```
int fkt(double, int*) throw();
```

bedeutet, dass diese Funktion keine Ausnahmen auswirft!

Bei der (klassischen) Deklaration (und Definition):

```
int fkt(double, int*);
```

ist nichts zu möglichen Ausnahmen gesagt, d.h. diese Funktion könnte möglicherweise irgendwelche Ausnahmen auswerfen oder auch nicht!

Ist zu einer Funktion (bei Definition und Deklaration) eine Liste von möglicherweise ausgeworfenen Ausnahmen angegeben und wird während des Ablaufs der Funktion eine andere, nicht bei der Funktionsdeklaration aufgeführte Ausnahme ausgeworfen, so ist dies ein Fehler, der standardmäßig (auf UNIX-Systemen mit Speicherabzug) zum Programmabbruch führt!

Dieser Programmabbruch kann vermieden werden, indem man der Funktionsdeklaration und Definition eine möglicherweise ausgeworfene Ausnahme vom in der Standardbibliothek definierten Typ `std::bad_exception` hinzufügt.

```
// Deklaration
int fkt() throw(..., std::bad_exception);
...
// Definition
int fkt() throw(..., std::bad_exception) { ... }
...
```

In diesem Fall wirft das System immer dann, wenn von der Funktion eine “unerwartete” (bei Deklaration und Definition nicht angegebene) Ausnahme auswirft, anstelle dieser “unerwarteten” eine von dem Typ `std::bad_exception` (welche man dann natürlich abfangen sollte).

Noch feiner kann das Verhalten bei “unerwarteten” Ausnahmen analog zur Behandlung nicht abgefangener Ausnahmen selbst eingestellt werden:

In der Headerdatei `<exception>` ist wiederum ein Funktionstyp:

```
typedef void (*unexpected_handler) ();
```

definiert und die Bibliotheksfunktion:

```
unexpected_handler set_unexpected (unexpected_handler);
```

deklariert, mit der eine selbstgeschriebene Funktion:

```
void unerwartet(){...}
```

durch:

```
set_unexpected ( unerwartet );
```

so installiert werden kann, so dass im Folgenden bei jeder Funktion, welche eine nicht bei ihrer Deklaration hinter `throw` angegebene Ausnahme auswirft, diese selbstgeschriebene Funktion `unerwartet` aufgerufen wird!

Das Funktionsergebnis von `set_unexpected` ist die Adresse der bislang für diesen Fall zuständigen Funktion.

Üblicherweise sollte innerhalb der selbstgeschriebenen Funktion `unerwartet` wiederum eine Ausnahme ausgeworfen werden, welche zur Schnittstellendefinition der beteiligten Funktionen passt! Ein völliges Ignorieren von unerwarteten Ausnahmen scheint nicht möglich zu sein!

2.12 Die neue Verwaltung des Freispeichers

Alternativ zu der aus C bekannten, beispielsweise wie folgt angewendeter Freispeicherverwaltung mittels `malloc` und `free`:

```
...
double * dp1, *dp2;
...
/* Reservierung eines double: */
dp1 = (double *) malloc( sizeof( double) );
...
/* Reservierung eines double-Feldes: */
dp2 = (double *) malloc ( 100*sizeof(double) );
...
/* Freigabe: */
free(dp1);
free(dp2);
...
```

(diese Verwaltung ist auch in C++ noch verfügbar, Headerdatei `<cstdlib>` includen!)
gibt es in C++ eine neue Verwaltung des Freispeichers.

Hierzu sind die folgende Operatoren definiert:

- **new** zur Reservierung von Speicher für ein Objekt (und ggf. anschließende Initialisierung),
- **new[]** zur Reservierung von Speicher für ein Feld (Array) von Objekten (und ggf. anschließende Initialisierung),
- **delete** zur Freigabe des Speichers (und Zerstörung) eines mittels **new** erzeugten Objektes und
- **delete[]** zur Freigabe des Speichers (und Zerstörung) eines mittels **new[]** erzeugten Feldes von Objekten.

Die Reservierung und Freigabe des Speichers für ein einzelnes Objekt sieht etwa wie folgt aus:

- Reservierung:

- Reservierung ohne explizite Initialisierung:

```
double *dp = new double;
```

Der Zeiger `dp` zeigt (wenn die Reservierung geklappt hat) auf ein (nicht explizit initialisiertes) dynamisch angelegtes `double`-Objekt. (Der Standard bietet mehrere Möglichkeiten, nicht erfüllte Speicheranforderungen zu erkennen und auf diese zu reagieren, siehe Abschnitt 2.12.1!)

- Reservierung mit expliziter Initialisierung:

```
double *dp = new double ( wert );
```

Der Zeiger `dp` zeigt (wenn die Reservierung geklappt hat) auf ein dynamisch angelegtes `double`-Objekt, welches bei seiner Erzeugung den (ggf. in den Typ `double` umgewandelten) angegebenen Wert `wert` erhält.

Funktionsergebnis ist jeweils vom Typ `double*`.

Ohne explizite Initialisierung hat das Objekt seinen "Standardwert", bei Standardtypen also einen zufälligen. (Bei selbstdefinierten Datentypen wird deren Standardkonstruktor aufgerufen!)

- Freigabe:

```
delete dp;
```

Hinter dem `delete` ist die Adresse eines vorher mittels `new` reservierten Speicherbereiches (oder die Adresse 0) anzugeben. Der Speicherbereich wird freigegeben (beim Argument 0 passiert nichts schädliches!).

Die Reservierung und Freigabe des Speichers für ein Feld von Objekten sieht dann wie folgt aus:

- Reservierung:

- Reservierung ohne explizite Initialisierung:

```
double *dp = new double[100];
```

Der Zeiger `dp` zeigt (wenn die Reservierung geklappt hat) auf ein (nicht explizit initialisiertes) dynamisch angelegtes Feld von 100 `double`-Objekten.

- Reservierung mit expliziter Initialisierung (mit dem angegebenen `wert`):

```
double *dp = new double[100] ( wert );
```

Hinter der Typangabe ist hierbei in eckigen Klammern die Länge des Feldes anzugeben. Der Zeiger `dp` zeigt (wenn die Reservierung geklappt hat) auf ein dynamisch angelegtes Feld von 100 `double`-Objekten, welche alle bei ihrer Erzeugung den (ggf. in den Typ `double` umgewandelten) angegebenen Wert `wert` erhalten.

- Freigabe:

```
delete[] dp;
```

Ist wiederum der Adress-Wert 0 das Argument von `delete[]`, so passiert nichts schädliches.

Die Vorteile dieser neuen Operatoren sind:

- Es sind eingebaute Operatoren und keine Bibliotheksfunktionen wie in C.
- Der Typ des anzulegenden Objektes kann direkt angegeben werden, der Umweg über `sizeof` ist nicht mehr erforderlich.

- Ergebnis der Operatoren ist jeweils Adresse vom richtigen Typ, eine erzwungene Typumwandlung ist ebenfalls nicht mehr erforderlich!
- Der reservierte Speicherbereich kann direkt initialisiert werden.
- Diese Operatoren können auch für selbstdefinierte Datentypen neu definiert und den Anforderungen des neuen Types angepasst werden.
- Die neuen Operatoren bieten vielfältigere Möglichkeiten, unerfüllte Speicheranforderungen zu erkennen und hierauf zu reagieren (siehe Abschnitt 2.12.1).

Vorsicht ist jedoch geboten:

Ein mit `new` reservierter Speicherbereich darf nur mit `delete` und ein mit `new[]` reservierter Speicherbereich darf nur mit `delete[]` wieder freigegeben werden! Verwechslungen sind hier unbedingt zu vermeiden, da sie zu Laufzeitfehlern des Programms führen!

Gleichzeitig zur “neuen” Speicherverwaltung (`new`, `new[]`, `delete`, `delete[]`) kann auch die “alte” (`malloc`, `free`) eingesetzt werden — doch auch hier darf die Verwendung nicht vermischt werden, ein mit `new` reservierter Speicherbereich darf etwa nicht mit `free` wieder freigegeben werden!

2.12.1 Unerfüllbare Speicheranforderungen

Standardeinstellung

Standardmäßig wird bei einer unerfüllbaren Speicheranforderung mittels `new` oder `new[]` eine Ausnahme vom Typ `bad_alloc` ausgeworfen. Dieser (zum Namensraum `std` gehörende) Typ ist in der Headerdatei `<new>` definiert!

Dies bedeutet, dass man nicht (wie bei `malloc`) bei jeder Speicheranforderung selbst überprüfen muss, ob die Reservierung geklappt hat oder nicht!

Eine Anwendung kann beispielsweise wie folgt aussehen:

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

struct A { ... }; // selbstdefinierter Typ
...
try
{ ...
    // Speicher reservieren, ohne Ueberpruefung,
    // ob es geklappt hat:
    double *dp = new double[1000];
    char *cp = new char[10000];
    A *ap = new A;
    ...
}
catch( bad_alloc fehler)
{ ... // auf Speichermangel reagieren
}
```

Vergisst man, um die `new`-Aufrufe einen `try`-Block zu schreiben oder Ausnahmen von diesem Typ `bad_alloc` mit einem `catch` abzufangen, so reagiert das Programm wie üblich bei nicht abgefangenen Ausnahmen — also standardmäßig mit einem Programmabbruch (siehe Abschnitt 2.11.5).

Keine Ausnahmen bei `new` und `new[]`

Möchte man, dass bei `new` bzw. `new[]` und Speichermangel keine Ausnahme `bad_alloc` ausgeworfen wird, so kann man zum `new` bzw. `new[]` das zusätzliche (ebenfalls zum Namensraum `std` gehörende und in `new` definierte) Argument `nothrow` angeben. In diesem Fall geben diese Operatoren bei Speichermangel die Adresse 0 zurück, an welcher die fehlgeschlagene Speicherreservierung abgelesen werden kann.

Eine vernünftige Anwendung sieht dann (ähnlich wie bei `malloc` üblich) so aus:

```
...
using namespace std;
double *dp1, *dp2;

if ( (dp1 = new(nothrow) double) == 0)
{ // nicht geklappt
}

...
if ( (dp2 = new(nothrow) double[100]) == 0)
{ // nicht geklappt
}

...
```

Einen `new`-Handler installieren

Alternativ kann man eine Funktion ohne Argumente und Rückgabe definieren, etwa

```
void speichermangel()
{ ... }
```

und durch folgenden Aufruf

```
set_new_handler( speichermangel);
```

der (ebenfalls in `<new>` deklarierten) Bibliotheksfunktion `set_new_handler` dem System mitteilen, dass bei Speichermangel im Zusammenhang mit `new` oder `new[]` keine Ausnahme auszuwerfen ist, sondern genau diese selbstdefinierte und beim Aufruf von `set_new_handler` als Argument angegebene Funktion aufzurufen ist.

Da die selbstgeschriebene Funktion bei akutem Speichermangel aufgerufen wird, sollte in dieser Funktion alles unterlassen werden, was (auch implizit) weiteren Speicher erfordert (etwa Standard- oder Fehlerausgabe).

Aus diesem Grund sollten ggf. nur "*Low-Level*"-Ausgabefunktionen, etwa auf UNIX-Systemen der `write`-Systemaufruf, und zum Programmabbruch der Systemaufruf `_exit(int)` verwendet werden!

Für den Typen dieser selbstgeschriebenen Funktion `speichermangel` ist in `new` der Typname:

```
typedef (*new_handler)();
```

definiert und die genaue Deklaration von `set_new_handler` sieht so aus:

```
new_handler set_new_handler( new_handler);
```

d.h. die Bibliotheksfunktion `set_new_handler` erhält nicht nur als Argument eine Funktion (vom Typ `new_handler`), sondern gibt auch als Funktionsergebnis die Adresse einer solchen Funktion zurück — nämlich die Adresse der bisher für Speichermangel zuständigen Funktion.

Somit kann man für einen gewissen Programmabschnitt einen separaten New-Handler installieren und nachträglich die Einstellung wieder auf den vorherigen New-Handler — was immer der war — zurückstellen:

```
...
new_handler *fkt_ptr; // Funktionszeiger
void speichermangel()
{ ... } // selbstdefinierte Funktion
...
/* selbstgeschriebene Funktion als New-Handler installieren,
   dabei die Adresse des bislang gueltigen Handlers in fkt_ptr
   sichern */
fkt_ptr = set_new_handler(speichermangel);
.
.
.
// vorherigen Zustand wieder restaurieren:
set_new_handler( fkt_ptr);
...
```

Mittels des Aufrufs

```
set_new_handler(0);
```

wird wiederum die Standardeinstellung für die Reaktion auf Speichermangel eingestellt, d.h. es ist kein New-Handler installiert und es wird mittels Ausnahmen auf Speichermangel reagiert.

Reservespeicher anlegen

Die meisten Programme können bei eintretendem Speichermangel nicht mehr ordnungsgemäß, ihrer Bestimmung entsprechend ablaufen — sie müssen in der Regel irgendwie abgebrochen werden.

Bei vielen Anwendungen ist es jedoch erforderlich, dass vor dem Programmabbruch noch ein paar wichtige Dinge erledigt werden müssen — etwa die bisher bearbeiteten Daten noch abspeichern oder bislang ermittelte Ergebnisse noch irgendwie ausgeben. Zur Ermöglichung von solchen Aufräumarbeiten, welche ggf. in geringem Umfang selbst auch noch weiteren Speicher erfordern, kann folgender “Trick“ verwendet werden:

- Zu Beginn des Programms wird dynamisch ein moderat großer, für Aufräumarbeiten ausreichender “Reservespeicher“ reserviert, auf dessen Anfang eine globale Adress-Variable verweist.

- Es wird ein New-Handler selbst geschrieben und installiert, an dessen Anfang zunächst die Freigabe des Reservespeichers steht, anschließend Aufräumarbeiten (Datensicherungen) vorgenommen werden und schließlich das Programm beendet wird.

Das Szenario könnte also etwa wie folgt aussehen:

```
...
#include <new>
char *reserve;           // Zeiger auf Reservespeicher
void speichermangel(void); // Dekl. des eigenen New-Handlers
...
int main()
{
    reserve = new char[10000]; // Reservespeicher anlegen
    set_new_handler( speichermangel); // eigenen New-Handler installieren
    ...
}

// Definition des New-Handlers:
void speichermangel()
{
    delete[] reserve; // zunaechst mal Reservespeicher freigeben
    ...               // Aufraeumungsarbeiten durchfuehren
    exit(1);          // Programm beenden
}
...
```

2.12.2 Platzieren von Objekten

Die Operatoren `new` und `new[]` gibt es standardmäßig noch in der Variante, dass ein zusätzliches Argument vom Typ `void *` (also eine beliebige Adresse) angegeben werden kann. Die Speicheranforderung bedient sich dann dieser Adresse, d.h. die angegebene Adresse wird vom Operator zurückgegeben, aber vom richtigen Typ. D.h. es wird gar kein wirklich neuer Speicherbereich reserviert, sondern der vorher bereits vorhandene, ggf. aber in einem anderen Typ, verwendet.

Auf diese Art und Weise hat man die Möglichkeit, ein Objekt oder ein Feld von Objekten eines bestimmten (i. Allg. dann selbstdefinierten) Types an eine gewisse (vorher im Programm bereits bekannte) Adresse im Arbeitsspeicher zu platzieren, Beispiel:

```
...
struct A { ... }; // selbstdefinierter Datentyp
...
// void-Adresse beschaffen:
void *buffer = reinterpret_cast<void *>( 0xFFFF);
...
// Platzieren eines A-Objektes an diese Adresse:
```



```
A *ap = new(buffer) A;  
...
```

Bei einer derartigen Plazierung von Objekten wird durch `new` bzw. `new[]` keine Ausnahme `bad_alloc` ausgeworfen.

Kapitel 3

Einblick in die Standardbibliothek

Die Standardbibliothek von C++ ist im Zuge der Standardisierung völlig neu überdacht worden.

In ihr werden die in C++ ermöglichten Konzepte der Objektorientierung (Klassen und Vererbung, Polymorphie) und generischen Programmierung (Templates) eingesetzt — aber man kann die Standardbibliothek auch verwenden, ohne selbst objektorientiert programmieren zu müssen.

Die Standardbibliothek besteht aus drei wesentlichen Bestandteilen:

- Standard Ein/Ausgabe,
- einem neuen Datentyp `string` zur komfortablen Verarbeitung von Zeichenketten,
- der *Standard-Template-Library* (kurz: STL) — eine Ansammlung von nützlichen Templates (Klassen und Funktionen), mit der direkt komplexe Strukturen (wie etwa lineare Listen) und zugehörige Algorithmen für konkrete, auch selbstdefinierte, abstrakte Datentypen verfügbar sind.

In diesem Kapitel wird kurz auf die Standard Ein/Ausgabe und den Datentyp `string` eingegangen (Details in Kapitel 8 und 10).

Die STL wird hier nicht behandelt, da hier die zum Verständnis notwendigen Mittel der generischen Programmierung und Objektorientierung noch fehlen. Die STL wird ausführlich in Kapitel 11 vorgestellt.

3.1 Aus–/Eingabe

3.1.1 Datentypen zur Ein– und Ausgabe

Zur Ein–/Ausgabe bietet C++ einen völlig neuen Zugang gegenüber C (obwohl C–Ein–/Ausgabe immer noch möglich ist, vgl. Abschnitt 2.1)!

Die Möglichkeiten der C++–Ein–/Ausgabe sind in der Headerdatei `<iostream>` deklariert, welche somit mittels

```
#include <iostream>
```

einzubinden ist. Die im Standard definierten Datentypen, Funktionen und (globalen) Variablen gehören zum Namensbereich `std`, so dass ggf. mittels expliziter Qualifikation `std::` auf sie zugegriffen werden muss.

Der Einfachheit halber, und da es auf vielen (auch unserem) Compiler auch ohne funktioniert, wird im Folgenden diese Qualifikation `std::` fortgelassen!

In `<iostream>` sind zunächst die Datentypen `ostream` (Ausgabestrom) und `istream` (Eingabestrom) definiert. Auf einen `ostream` kann "geschrieben" und von einem `istream` kann gelesen werden.

Von diesem Typ `ostream` sind bereits die globalen Variablen `cout` (Standardausgabe, entspricht dem `stdout` in C) und `cerr` (Standardfehlerausgabe, entspricht dem `stderr` in C) und vom Typ `istream` die globale Variable `cin` (Standardeingabe, entspricht dem `stdin` in C) definiert.

3.1.2 Der Ausgabeoperator <<

Für die Ausgabe auf einen Strom spielt der "Ausgabeoperator" << eine große Rolle. (Es handelt sich hierbei um den von C stammenden Links-Shift-Operator, welcher für alle Standardtypen überladen ist und der auch für eigene Typen überladen werden kann. Mehr zu Operatorüberladung im Abschnitt 5.2.)

Die Ausgabe einer Zeichenkette auf die Standardausgabe geschieht etwa wie folgt:

```
cout << "hello, world\n";
```

Die Funktionalität dieses Operators richtet sich nach dem, was auszugeben ist, d.h. dieser Ausgabeoperator ist nicht nur für Zeichenketten, sondern auch für `int`-Werte, `double`-Werte, `char`-Werte, ... verwendbar:

```
...
int i;
double x;
char c, wort[] = "Hello";
...
cout << "Hallo";    // Ausgabe eines String-Literals
cout << i;          // Ausgabe eines int-Wertes
cout << 7;          // Ausgabe einer int-Konstanten
cout << x;          // Ausgabe eines double-Wertes
cout << 3.141;      // Ausgabe einer double-Konstanten
cout << c;          // Ausgabe eines Zeichens
cout << "\n";       // Ausgabe eines konstanten Zeichens,
                    // Zeilenvorschubzeichen
cout << wort;       // Ausgabe einer Zeichenkette
...
cerr << "Kein Speicher mehr!\n"; // Ausgabe einer Meldung
                    // auf die Standardfehlerausgabe
...
```

Dieser Operator << ist i. Allg. so definiert, dass der auch geschachtelt in folgender Form aufgerufen werden kann (man beachte die unterschiedlichen Typen der auszugebenden Objekte!):

```

...
int i;
double x;
...
cout << i << " * " << x << " = " << i*x << '\n';
...

```

Diese Ausgabe entspräche folgender C-printf-Anweisung:

```
printf("%d * %g = %g\n", i, x, i*x);
```

Der Operator << hat bei der Ausgabe eine zentrale Bedeutung — will man bei einer Ausgabe wirklich einen Bit-Shift-Operator aufrufen, muss man explizit entsprechend klammern:

```
cout << (i << j) ;
```

Bit-Shift

Die Ausgabe mittels des <<-Operators erfolgt unformatiert (bzw. in einem voreingestellten Standardformat), d.h. es werden immer genauso viele Zeichen ausgegeben, wie zur Darstellung des auszugebenden Wertes erforderlich — nicht mehr und nicht weniger.

Wie in C kann auch in C++ eine Feldbreite, Ausrichtung innerhalb der Feldbreite (rechts- oder linksbündig) und vieles mehr eingestellt werden, was das Aussehen der Ausgabe irgendwie beeinflusst. Hierauf wird in Abschnitt 8.4.3 eingegangen.

Desweiteren gibt es eine Reihe weiterer Funktionen zur Ausgabe auf einen Ausgabestrom, auf die hier ebenfalls nicht weiter eingegangen wird (siehe Abschnitt 8.4.1).

3.1.3 Der Eingabeoperator <<

Analog zum Ausgabeoperator << ist für die Eingabe der Eingabeoperator >> definiert, der in natürlicher Weise für alle erdenklichen Standardtypen verwendet werden kann:

```

int    i;
char   c, w[100];
short  s;
long   l;
double x;
float  f;
...
cin >> i;    // Einlesen eines int-Wertes, Abspeichern in Variable i
cin >> c;    // Einlesen eines Zeichens, Abspeichern in Variable c
cin >> w;    // Einlesen eines Strings, Abspeichern in char-Feld w
cin >> s;    // Einlesen eines short-Wertes, Abspeichern in Variable s
cin >> l;    // Einlesen eines long-Wertes, Abspeichern in Variable l
cin >> x;    // Einlesen eines double-Wertes, Abspeichern in Variable x
cin >> f;    // Einlesen eines float-Wertes, Abspeichern in Variable f
...

```

Der Operator >> ist wieder so definiert, dass das System an dem, was hinter dem Operator steht, erkennt, was wohl einzulesen ist — ist das Argument vom Typ int, so

werden alle folgenden Ziffern gelesen (und als `int`-Wert zugewiesen), ist das Argument vom Typ `char`, so wird nur ein Zeichen gelesen (und als ein Zeichen zugewiesen) usw.. Beim Einlesen mittels `>>` werden Referenzen (siehe Abschnitt 2.6) verwendet und hierdurch ist das Einlesen in C++ viel benutzerfreundlicher als in C, es brauchen nur Variablen (und nicht wie in C Adressen von Variablen) angegeben zu werden:

```
...
int i;
...
// C-Einlesen von i:
scanf("%d",&i);    // &i: Adresse von i
...
// C++-Einlesen von i:
cin >> i;           // i reicht
...
```

Der Operator `>>` ist wiederum so definiert, dass er geschachtelt aufgerufen werden kann:

```
int    i;
char   c, w[100];
short  s;
long   l;
double x;
float  f;
...
// Alles auf einmal einlesen
cin >> i >> c >> w >> s >> l >> x >> f;
...
```

Standardmäßig werden beim Einlesen mittels des Operators `>>` Zwischenraumzeichen (Leerzeichen, Tabulatoren, Zeilenvorschübe) überlesen. Somit wird in:

```
char c;
cin >> c;
```

das nächste, Nichtzwischenraumzeichen der `char`-Variablen `c` zugewiesen. Nach dem Überlesen des Zwischenraums werden alle folgenden Zeichen gelesen (und “gesammelt“), welche noch zu einem Wert vom Typ des Argumentes gehören können, und das erste Zeichen, welches nicht mehr zum Argumenttyp “passt“ (etwa ein Buchstabe `z` beim Einlesen eines `int`-Wertes), beendet den Lesevorgang (und “wird nicht gelesen“). Der Wert, den die “gesammelten“ Zeichen repräsentieren, wird dem Argument zugewiesen.

Beim Lesen von Zeichenketten

```
char w[100];
cin >> w;
```

ist das nach führendem Zwischenraum und folgendem Nichtzwischenraumzeichen nächste Zwischenraumzeichen das erste, dann nicht mehr zur Zeichenkette “passende“ Zeichen. Alle gelesenen Zeichen werden der Reihe nach in das als Argument angegebene Zeichenfeld `w` geschrieben und es wird ein `'\0'`-Zeichen angehängt (Überlaufgefahr!). Bei der Eingabe gilt hier wie in C: passt das nach Zwischenraum stehende erste Zeichen nicht zum Typ des Argumentes (etwa ein Buchstabe `z` beim Einlesen eines `int`-Wertes), so wird dieses Zeichen nicht gelesen und es wird auch nichts zugewiesen!

Ein solcher fehlgeschlagener Leseversuch oder auch das Ende der Eingabe (`EOF` wie in C) wird im “Zustand“ des Eingabestroms (hier also `cin`) abgespeichert und mit geeigneten Mitteln kann man diesen Zustand abfragen und entsprechend reagieren (siehe Abschnitte 3.1.5 und 8.6).

Das Lesen mittels `>>` ist standardmäßig unformatiert — es können jedoch wiederum Formatierungen eingestellt werden, siehe Abschnitt 8.5.2.

Neben dem Eingabe-Operator `>>` gibt es eine Reihe weiterer Funktionen zum Lesen von einem Eingabestrom (siehe Abschnitt 8.5.1).

3.1.4 Manipulatoren

Manipulatoren sind wie gewöhnliche Variablen aussehende “Dinge“, welche (wie Variablen) auf einen Ausgabestrom “ausgegeben“ (*Ausgabemanipulatoren*) bzw. von einem Eingabestrom “gelesen“ (*Eingabemanipulatoren*) werden können.

Bei der Ausgabe bzw. Eingabe eines Manipulators muss jedoch nicht wirklich etwas neues ausgegeben bzw. gelesen werden, sondern der Aus- bzw. Eingabestrom wird in irgendeiner Weise beeinflusst (*manipuliert*).

Es gibt (u.A.) folgende Ausgabe-Manipulatoren:

- **flush:**

bewirkt das Leeren des Ausgabepuffers.

Ausgabe ist im Allgemeinen gefuffert, alles, was ausgegeben wird, erscheint nicht unmittelbar auf dem Bildschirm, sondern wird zunächst in einen Ausgabepuffer geschrieben. Erst, wenn dieser Puffer voll ist, wird er “geleert“, d.h. der Inhalt wird jetzt erst tatsächlich auf dem Bildschirm ausgegeben.

Die “Ausgabe“ des Manipulators **flush**:

```
cout << flush;
```

bewirkt ein sofortiges “Leeren“ (d.h. Ausgabe auf den Bildschirm) des zu `cout` gehörenden Ausgabepuffers.

- **endl:**

bewirkt die Ausgabe eines Zeilenvorschubzeichens `'\n'` und anschließende “Leerung“ des Ausgabepuffers.

- **ends:**

bewirkt die Ausgabe eines Stringendezeichens `'\0'` und anschließende “Leerung“ des Ausgabepuffers.

In “einer“ Ausgabeoperation können mehrere Manipulatoren (auch gleiche mehrfach) “ausgegeben“ werden:

```
cout << i << endl << j << endl << k << flush;
```

Es gibt ebenfalls Eingabe-Manipulatoren, von diesen sei hier nur

– **ws**:

“überliest“ Zwischenraumzeichen, diese werden gelesen und nicht abgespeichert. Der Eingabestrom “steht“ jetzt auf dem nächsten Zeichen, welches kein Zwischenraumzeichen ist.

erwähnt. Mehr zu Manipulatoren ist in Abschnitt 8 zu finden.

3.1.5 Fehlerzustände in Strömen

Der Zustand eines Datenstroms ist im Datenstrom selbst abgespeichert und der Name des Datenstroms kann als Bedingung verwendet werden, die genau dann **true** liefert, wenn der Datenstrom *in Ordnung*, also fehlerfrei ist, und ansonsten **false** (insbesondere beim Ende der Eingabe!):

```
...
int i;
...
cin >> i;
if ( cin )
{ // Lesen von i hat geklappt!
    ...
}
...
double x;
...
cin >> x;
if ( !cin )
{ // Lesen hat nicht geklappt!
    ...
}
...
```

Da der Eingabeoperator `>>` den Eingabestrom (nach dem Lesen) selbst als Ergebnis zurückliefert, kann er wie folgt verwendet werden:

```
...
int i;
...
while ( cin >> i ) // solange Lesen eines int's klappt
{ // mach was mit i
    ...
}
...
```


3.1.6 Dateibehandlung

Natürlich kann in C++ auch von Dateien gelesen und auf Dateien geschrieben werden. Hierzu gibt es die Klassen:

- `ifstream` (*input file stream*) für Dateien, von denen gelesen werden soll (gleiche Funktionalität wie `istream`),
- `ofstream` (*output file stream*) für Dateien, auf die geschrieben werden soll (gleiche Funktionalität wie `ostream`).

Objekte (Variablen) dieser Klassen werden (i. Allg.) bei ihrer Erzeugung mit einer Datei des Systems verknüpft — bei der Erzeugung muss dann ein Dateiname angegeben werden:

```
...
ofstream ausgabe("Ausgabe.txt");
/* ausgabe ist eine Variable vom Typ ofstream und ist mit
   der Datei Ausgabe.txt des Systems verknuepft.
   Diese Datei wird zum Schreiben geoeffnet          */
...
ifstream eingabe("Eingabe.txt");
/* eingabe ist eine Variable vom Typ ifstream und ist mit
   der Datei Eingabe.txt des Systems verknuepft.
   Diese Datei wird zum Lesen geoeffnet              */
...
```

Ob das Öffnen einer Datei geklappt hat, ist wiederum im Zustand des Stromes vermerkt und dieser Zustand sollte im Allgemeinen abgefragt werden, bevor man Ein- bzw. Ausgabeoperationen durchführt:

```
...
ofstream ausgabe("Ausgabe.txt");
if ( !ausgabe )
{
    // Oeffnen hat nicht geklappt!
    ... // reagiere darauf.
}
...
ifstream eingabe("Eingabe.txt");
if ( !eingabe )
{
    // Oeffnen hat nicht geklappt!
    ... // reagiere darauf.
}
...
```

Wie bereits erwähnt hat der Typ `ofstream` die gleiche Funktionalität wie `ostream` und der Typ `ifstream` wie `istream`, d.h. die entsprechenden Operatoren und Techniken (insbes. Manipulatoren) sind mit gleicher Bedeutung definiert und verfügbar.

Die Ausgabe auf die wie oben geöffnete und mit `ofstream` `ausgabe` verknüpfte Datei `Ausgabe.txt` kann also wie folgt geschehen:

```
ausgabe << "Ergebnis: " << i << endl;
```

und das Lesen von der wie oben geöffneten und mit `ifstream` `eingabe` verknüpften Datei `Eingabe.txt` kann wie folgt geschehen:

```
...
int i;
double x;
...
eingabe >> i;           // Einlesen eines int
if ( !eingabe ) { ... } // Lesen hat nicht geklappt
...
eingabe >> x;           // Einlesen eines double
if ( !eingabe ) { ... } // Lesen hat nicht geklappt
...
```

Geöffnete Dateien brauchen im Allgemeinen nicht explizit geschlossen zu werden, dies übernimmt das System, wenn die Variable vom Typ `ofstream` bzw. `ifstream` an ihrem “Lebensende” angekommen ist:

```
...
int f(void)
{
    ofstream ausgabe("Ausgabe.txt");    // Lebensanfang, Variable ausgabe
        // wird erzeugt und mit der hierbei geöffneten Datei verknüpft
    ...
    return 0;
} // Lebensende der automatischen Variablen ausgabe,
    // Datei wird hierbei vom System geschlossen!
...
```

3.2 Die Standard-String-Klasse

Die C++-Standardbibliothek stellt den Datentyp (Klasse) `string` mit einer ganzen Reihe von Funktionen und Operatoren zur einfachen Verarbeitung von Zeichenketten zur Verfügung — dieser Datentyp kann weitestgehend die aus C bekannten und in C++ ebenfalls möglichen, ‘\0’-terminierten Felder vom Typ `char` ersetzen.

(Der Datentyp `string` ist ein durch die Standardbibliothek “selbstdefinierter” Datentyp und **kein** in C++ “eingebauter” Typ!)

Zur Verwendung dieses Datentypes muss die Headerdatei `<string>` includet werden. Im Folgenden bezeichne ich Objekte dieser Klasse `string` als *Strings* oder *C++-String* und ‘\0’-terminierte `char`-Felder als *C-Strings*.

In diesem Abschnitt werden nur einige Funktionalitäten dieser Klasse vorgestellt, eine detailliertere Beschreibung findet man in Abschnitt 10.

3.2.1 Erzeugung eines Strings

Bei der Erzeugung eines Strings kann dieser auf unterschiedliche Weise initialisiert, d.h. mit einer Zeichenkette vorbesetzt werden:

```
#include <string>
...
string s1;           // erzeugt leeren String
...
string s2("hallo");  // Initialisierung mit konst. Zeichenkette
string s3 = "hallo"; // Initialisierung mit konst. Zeichenkette
...
char w[100] = ...;   // char-Feld, muss C-String enthalten
string s4(w);        // Initialisierung mit char-Feld
string s5 = w;       // Initialisierung mit char-Feld
...
char *p=...;         // char-Zeiger, muss auf C-String zeigen
string s6(p);        // Initialisierung mit char-Zeiger
string s7 = p;       // Initialisierung mit char-Zeiger
...
string s8(s2);       // Initialisierung mit C++-String
string s9 = s2;      // Initialisierung mit C++-String
...
```

Der Inhalt des Strings ist dann jeweils eine Kopie der bei der Initialisierung angegebenen Zeichenkette.

Natürlich kann man auch konstante C++-Strings definieren, etwa:

```
const string s = ...;   bzw.
const string s(...);
```

(Konstante Strings sollten bei ihrer Erzeugung initialisiert werden, da ihr “Wert“ (Inhalt) später nicht mehr geändert werden kann!)

Die Initialisierung von Strings kann auch mit einem `const char w[] = ...` oder einem Zeiger auf `char const` erfolgen oder auch mit einem `const string` erfolgen.

3.2.2 Ein-/Ausgabe von Strings

Zur Ausgabe eines Strings steht wiederum der Ausgabeoperator `<<` zur Verfügung (in folgenden Beispielen wird immer `cout` als Ausgabe- bzw. `cin` als Eingabestrom angegeben, es könnte aber jeder andere Datenstrom verwendet werden!):

```
string s;
...
cout << s << endl; // gibt String s und Zeilenvorschub aus
...
```

Auch der Eingabeoperator `>>` ist für (nicht konstante) Strings definiert:

```
string s;
...
cin >> s;
...
```

hierbei wird zunächst führender Zwischenraum (Leerzeichen, Tabulatoren, Zeilenvorschübe) überlesen (nicht im String abgelegt), alle folgenden Zeichen bis (ausschließlich) zum nächsten Zwischenraumzeichen werden gelesen und der Reihe nach in den String `s` geschrieben. Der vorherige Inhalt des Strings geht hierbei verloren und der String wird bei Bedarf dynamisch (im Rahmen der Kapazitäten der Implementierung) vergrößert. (Wird beim Einlesen eines Strings die Kapazitätsgrenze des Systemes erreicht, wird dies im Eingabestrom als Fehler vermerkt!)

Zum Einlesen eines Strings stehen alternativ folgende (globale) Funktionen zur Verfügung (vgl. Abschnitt 8.5.1):

```
getline(cin, s);
```

liest die nächste Eingabezeile (alle Zeichen bis einschließlich des nächsten Zeilenvorschubs `'\n'`) und weist die gelesenen Zeichen (ausschließlich des Zeilenvorschubzeichens) dem String `s` zu. Funktionsergebnis ist der Eingabestrom nach dem Lesevorgang.

Ist `c` ein Zeichen, so übernimmt im Aufruf:

```
getline(cin, s, c);
```

dieses Zeichen `c` die Rolle des Zeilenvorschubs im obigen Aufruf `getline(cin, s);`, d.h. es werden alle Zeichen bis (einschließlich) des nächsten mit `c` übereinstimmenden gelesen und die gelesenen Zeichen (ausschließlich des `c`) im String `s` abgespeichert.

3.2.3 Zugriff auf einzelne Zeichen eines Strings

Natürlich ist ein String intern “irgendwie” als (dynamisches) Feld vom Typ `char` abgespeichert und man kann wie in C mittels Indizierung (Operator `[]`) auf die einzelnen Zeichen des Strings zugreifen:

```
string s = "hallo";
s[0] = 'H';
cout << s << endl;    /// Ausgabe: Hallo
```

Bei einem derartigen Zugriff auf einzelne Zeichen eines Strings (`s`) muss man selber darauf achten, dass der Index im erlaubten Bereich liegt. Greift man mit negativem oder zu großem Index zu, so führt das (wie in C) zu merkwürdigen Laufzeitfehlern. Greift man mit Indizierung auf einen konstanten String zu, so kann der gelieferte Buchstabe **nicht** abgeändert werden!

3.2.4 Vergleich von Strings

Für Strings stehen alle Vergleichsoperatoren zur Verfügung — es wird anhand des Maschinenzzeichensatzes verglichen:

<code>==</code>	Test auf Gleichheit
<code>!=</code>	Test auf Ungleichheit
<code><</code>	Test auf lexikographisch kleiner
<code>></code>	Test auf lexikographisch größer
<code><=</code>	Test auf lexikographisch kleiner oder gleich
<code>>=</code>	Test auf lexikographisch größer oder gleich

Beispiele:

```
string s1 = ..., s2 = ...;
...
if ( s1 == s2 ) // Inhalte gleich?
{ ... }
...
if ( s1 < s2 ) // s1 lexikographisch kleiner als s2?
{ ... }
...
```

Der Vergleich von Strings ist somit in C++ viel intuitiver als in C mittels `strcmp`. Die Vergleichsoperatoren für Strings sind so definiert, dass ein C++-String auch mit einem C-String verglichen werden kann — eine der beim Vergleich beteiligten Zeichenketten muss jedoch ein C++-String sein:

```
string s = ...;
char w[100] = ...; // muss C-String enthalten
char *p = ...;     // muss auf C-String zeigen
...
if ( s == "hallo" ) // Vergleich mit Zeichenkettenliteral
{ ... }
if ( "hallo" == s ) // auch so herum moeglich
{ ... }
if ( w < s )        // Vergleich mit char-Feld
{ ... }
if ( s > w )        // auch so herum moeglich
{ ... }
if ( s >= p )       // Vergleich mit char-Zeiger
{ ... }
if ( p <= s )       // auch so herum moeglich
{ ... }
```

3.2.5 Verketteten von Strings

Strings können mit dem Additionsoperator `+` verkettet werden:

```
string s1 = ..., s2 = ...;
...
... s1 + s2 ...
...
```

Ergebnis dieser Operation `s1 + s2` ist ein temporärer String, in dem zunächst (am Anfang) der Inhalt von `s1` und unmittelbar dahinter der von `s2` steht. Die Stringlänge dieses temporären Strings ist somit die Summe der Stringlängen von `s1` und `s2`.

(Wird hierbei der temporäre String zu groß für das System, wird eine Ausnahme vom in der Standardbibliothek definierten Typ `length_error` ausgelöst!)

Dieser `+`-Operator ist wiederum so definiert, dass einer der beteiligten Operanden (erster oder zweiter) auch ein C-String — sogar ein einzelnes Zeichen sein kann — der andere Operand muss jedoch ein C++-String sein:

```
string s1 = ..., s2 = ...;
char w[100] = ...;      // muss C-String enthalten
char *p = ...;          // muss auf C-String zeigen
...

// Verkettungen
s1 + s2;                // String mit String
s1 + w;                 // String mit char-Feld
w + s1;                 // char-Feld mit String
s1 + p;                 // String mit char-Zeiger
p + s1;                 // char-Zeiger mit String
s1 + "hallo";           // String mit Zeichenkettenliteral
"hallo" + s1;           // Zeichenkettenliteral mit String
s1 + 'c';               // String mit Zeichen, entspricht: s1 + "c";
'c' + s1;               // Zeichen mit String, entspricht: "c" + s1;
```

3.2.6 Zuweisen an einen String

Der Zuweisungsoperator `=` ist ebenfalls für Strings definiert, wobei auf der rechten Seite ein C++-String, ein C-String oder auch ein einzelnes Zeichen stehen darf — links des Gleichheitszeichens muss ein C++-String stehen:

```
string s1 = ..., s2 = ...;
char w[100] = ...;      // muss C-String enthalten
char *p = ...;          // muss auf C-String zeigen
...

// Zuweisungen
s1 = s2;                // String an String
s1 = w;                 // char-Feld an String
s1 = p;                 // char-Zeiger an String
s1 = "hallo";           // Zeichenkettenliteral an String
s1 = 'c';               // Zeichen an String, entspricht: s1 = "c";
...
```

Der bisherige Inhalt (und Länge) von `s1` geht hierbei verloren.

Im Zusammenhang mit Verkettungen ist auch der Operator `+=` definiert, wobei auch hier der linke Operand ein C++-String sein muss:

```
string s1 = ..., s2 = ...;
char w[100] = ...;          // muss C-String enthalten
char *p = ...;              // muss auf C-String zeigen
...
...                          // Anhaengen
s1 += s2;                   // String s2 an String s1
s1 += w;                    // char-Feld w an String s1
s1 += p;                    // char-Zeiger an String s1
s1 += "hallo";              // Zeichenkettenliteral an String s1
s1 += 'c';                  // Zeichen an String s1, entspricht: s1 += "c";
...
```

Beim Anhängen an einen String kann natürlich wieder eine implementationsabhängige Kapazitätsgrenze überschritten werden — in diesem Fall wird wiederum eine Ausnahme vom Typ `length_error` ausgelöst!

Teil II

Objektorientierte Techniken

Kapitel 4

Klassen

4.1 Grundlagen

Objektorientierung ist eine Art und Weise zu Programmieren (*Programmierparadigma*).

Es folgt zunächst eine kurze Vorstellung der in diesem Zusammenhang wichtigsten Programmierparadigmen.

4.1.1 Programmierparadigmen

Prozedurale Programmierung

Das mit der Entwicklung der prozeduralen Programmiersprachen historisch erste Programmierparadigma ist die *prozedurale Programmierung*.

Die wichtigsten Merkmale:

- Trennung von Daten und Prozeduren (Funktionen),
- Hauptaugenmerk ist auf die Prozeduren gerichtet, welche tunlichst “optimale Algorithmen“ realisieren sollen,
- Daten werden von Prozedur an Prozedur weitergegeben, es gibt unterschiedliche Arten und Weisen, Daten weiterzugeben bzw. gemeinsam zu nutzen:
 - globale Daten.
 - Funktionsparameter/–Argumente (*Call by Value*, *Call by Reference*),
 - Funktionsergebnis, Rückgabeparameter

Hierbei auftauchende Probleme:

- Bei globalen Daten hat jeder Programmteil Zugriff auf diese Daten, dies kann zu unerwünschten Seiteneffekten führen, wenn ein Programmteil nicht korrekt auf solche Daten zugreift!
- Große Parameterlisten für Funktionen/Prozeduren machen diese unhandlich.

Abhilfe:

- ordentliche Organisation der Daten
- Zugriffsschutz (*Information-Hiding*)

Dies führt zum Programmierparadigma:

Modulare Programmierung

Merkmale:

- Menge von zusammengehörenden Daten und Prozeduren werden zu einem *Modul* zusammengefasst,
- der Modul wird mit einer Anwenderschnittstelle versehen,
- der Anwender kann nur auf diese Schnittstelle zugreifen, auf interne Dinge (interne Daten und Prozeduren) kann er nicht zugreifen (*Information-Hiding*, *Datenkapselung*).

Auch in C (und C++) kann man modular programmieren:

- Anwenderschnittstelle in Headerdatei deklarieren (etwa: `modul.h`),
- Implementierung in separater Implementations-Datei (etwa: `modul.c`), hierbei Techniken zur *Information-Hiding* verwenden,
- dem Anwender die Headerdatei und die übersetzte Implementierung (etwa: `modul.o`) zur Verfügung stellen.

Beispiel eines Moduls in C:

Keller-Speicher, ganzzahlige Werte sollen nach dem Lifo-Prinzip (*Last in, first out*) abgespeichert werden:

1. Headerdatei mit Anwenderschnittstelle (Name etwa: `Stack.h`):

```
#ifndef _keller_h
#define _keller_h
/* Schnittstelle zum Kellerspeicher */

/* Einkellern eines Wertes: */
void push (int);
/* Auskellern des zuletzt gespeicherten Wertes */
int pop (void);

#endif
```

- durch `#ifndef ...` wird sichergestellt, dass die Headerdatei nicht mehrfach in einer Übersetzungseinheit eingebunden wird,

- es wird nur die Schnittstelle — also die Funktionen `push` und `pop` — deklariert

2. Implementationsdatei (Name etwa: `Stack.c`):

```
/* Kellerspeicher, realisiert ueber ein int-Feld */
#include <stdio.h>    /* Ausgabe von Fehlermeldungen */
#include <stdlib.h>   /* Programmabbruch mit exit    */
#define MAX 100      /* Feldlaenge */

/* Keller      */
static int keller[MAX];

/* "stack-pointer", Index des ersten
   freien Elementes, mit 0 vorbesetzt */
static int sp = 0;

void push(int wert)
{ if ( sp >= MAX)    /* Keller voll? */
  { fprintf(stderr,"Keller voll!\n");
    exit(-1);
  }
  else
    keller[sp++] = wert;
}

int pop(void)
{ if ( sp == 0)      /* Keller leer? */
  { fprintf(stderr,"Keller leer!\n");
    exit(-1);
  }
  else
    return keller[ --sp];
}
```

- durch die Definition der Datenkomponenten `keller` und `sp` als `static` ist die Bekanntheit dieser Komponenten auf diesen Quelltext (`Stack.c`) beschränkt. Da dem Anwender (neben der Headerdatei `Stack.h`) nur die übersetzte Implementationsdatei (`Stack.o`) zur Verfügung gestellt wird, kann er gar nicht auf diese Komponenten zugreifen!
- Durch die Vorbesetzung von `sp` mit 0 wird sichergestellt, dass der Stack vernünftig initialisiert ist.

Nachteil dieser Modul-Realisierung des Kellerspeichers ist, dass nur ein solcher Kellerspeicher zur Verfügung steht und dass man Kellerspeicher nicht wie eingebaute Typen (etwa `int`) verwenden kann, d.h. folgende wünschenswerte Eigenschaften:

- mehrere Stacks verwenden: `Stack a, b, c;`
- Felder von Stacks: `Stack Stackfeld[100];`
- Stacks an Funktionen als Argument übergeben: `void fkt(Stack arg);`
- Stacks als Funktionsergebnisse erhalten: `Stack fkt(void);`

fehlen.

Das nächste Programmierparadigma macht auch dies möglich:

Programmieren mit Abstrakten Datentypen

- Ein abstrakter Datentyp ist eine Einheit aus Daten und zugehörigen Funktionen,
- gewisse dieser Funktionen sind für den Anwender zugänglich, andere nicht (Anwenderschnittstelle, *Information-Hiding*),
- man kann *Objekte* (Variablen) von diesem Typ wie Variablen eingebauter Typen verwenden.

Realisierung in C++:

Es wird ein neuer Datentyp `Stack` definiert.

Variablen von diesem Typ "sind" Kellerspeicher, d.h. man kann in ihnen `int`-Werte ablegen und später wieder herausholen.

1. Headerdatei:

```
#ifndef _Stack_h
#define _Stack_h

struct Stack { // neuer Datentyp hat Namen: Stack

    protected:    // Impl.-Details, nicht öffentlich
        int keller[100];
        int sp;

    public:        // öffentliche Schnittstelle

        Stack();    // benötigt, um Stack zu erzeugen
        void push(int); // Funktion zum Einkellern
        int pop(void); // Funktion zum Auskellern
};

#endif
```

- Die Deklaration dieses Types erfolgt analog der einer C-Struktur (Schlüsselwort `struct`).

Diese Struktur hat neben den Datenkomponenten (ganzzahlige Variable `int sp`; und ganzzahliges Feld `int keller[100]`;) zusätzlich auch Funktionen als Komponenten (die Funktion `void push(int)`; und die Funktion `int pop(void)`; und `Stack()`;). Als Komponenten sind hier die Deklarationen der entsprechenden Funktionen aufgeführt!

- Es tauchen die neuen Schlüsselworte `protected` und `public` (zusätzlich: `private`, kommt hier noch nicht vor!) auf, die die Zugreifbarkeit auf die Komponenten regeln.

Die zum *Zugriffsabschnitt* `protected` (und `private`) gehörenden Komponenten (Daten oder Funktionen) stellen die Implementierungsdetails dar, die der Anwender gar nicht zu kennen braucht — auf die entsprechenden Komponenten (Daten oder Funktionen) kann der Anwender gar nicht zugreifen (Information-Hiding).

Der `public`-Zugriffsabschnitt stellt die Anwenderschnittstelle zum Datentyp dar, auf die hier aufgeführten Komponenten (Daten und Funktionen) kann der Anwender (wie bei Strukturen üblich mittels der Operatoren `.` bzw. `->`) zugreifen (siehe die Beispielanwendung).

- Die für Kellerspeicher üblichen Funktionen `void push(int)`; zum “Einkellern“ eines Elementes und `int pop(void)`; zum “Auszellern“ eines Elementes sind wie gewohnt deklariert.
- Zusätzlich gibt es eine weitere Funktion mit dem gleichen Namen `Stack` wie der Datentyp, ohne Ergebnistyp (auch nicht `void`) und hier ohne Parameter. Es handelt sich um den sogenannten *Konstruktor*, der bei jeder Erzeugung einer Variablen von diesem Typ automatisch aufgerufen wird und der dafür sorgt, dass die Variable ordnungsgemäß initialisiert wird (hier: der Stack-Pointer muss mit 0 vorbesetzt werden, s.u.).

Die Komponenten der Struktur heißen *Member*, die Datenkomponenten auch *Member-Daten* und die Funktionskomponenten *Member-Funktionen* oder auch *Methoden*.

Die Member-Funktionen (zum Datentyp gehörende Funktionen) sind bislang nur deklariert, sie müssen natürlich auch irgendwo definiert werden. Dies erfolgt üblicherweise in der

2. Implementationsdatei:

```
// Stack.cc: Implementierung des Datentypes Stack
// Eigene Headerdatei einbinden
#include "Stack.h"
#include <iostream>    // wegen Ein-/Ausgabe
#include <cstdlib>     // wegen exit()

using namespace std;
```

```

/* Definition der zum Datentyp Stack
   gehoerenden Funktion push:          */
void Stack::push(int wert)
{ if ( sp >= 100)    // Keller voll?
  { cerr << "Keller voll" << endl;
    exit(-1);
  }
  else
    keller[sp++] = wert;
}

/* Definition der zum Datentyp Stack
   gehoerenden Funktion pop:          */
int Stack::pop(void)
{ if ( sp == 0)     // Keller leer?
  { cerr << "Keller leer" << endl;
    exit(-1);
  }
  else
    return keller[ --sp];
}

/* Definition des zum Datentyp Stack gehoerenden
   Konstruktors Stack:
   notwendig, damit bei der Erzeugung eines Stacks
   der Stack-Pointer sp den Wert 0 erhaelt! */
Stack::Stack()
{ sp = 0;
}

```

- Die eigene Headerdatei, in der die Deklaration des Datentypes steht, muss eingebunden werden, zusätzlich ggf. weitere notwendige Headerdateien.
- Die Definition der Member-Funktionen erfolgt unter Bezugnahme auf den Datentyp, die Schreibweise:

```
void Stack::push(int wert) { ... }
```

 bedeutet:
 Definition der zum Datentyp `Stack` gehörenden Funktion `void push(int)`.
 Die Definition der Funktionen `push` und `pop` ist wie üblich.
- Der Konstruktor `Stack::Stack()`; muss auch definiert werden, er wird automatisch bei jeder Erzeugung einer Variablen vom Typ `Stack` aufgerufen und muss dafür sorgen, dass die erzeugte Variable konsistent erzeugt wird, hier also dafür, dass der Stack-Pointer (Index des ersten freien Elementes im Feld) mit 0 vorbesetzt wird.

Eine Anwendung dieses Datentypes könnte wie folgt aussehen:


```
#include "Stack.h"

int main()
{ int i,j;    /* zwei int-Variablen */
  Stack a, b; /* Erzeuge zwei Stacks, der eine
               heisst a, der andere b          */
  ...
  a.push(7); /* Einkellern des Wertes 7 in den
              Stack a, es wird die Member-Funktion
              push fuer den Stack a aufgerufen */
  b.push(i); /* Einkellern des Wertes von i in Stack b */
  ...
  j = i+b.pop(); /* b.pop(): hole oberstes Element des
                  Stacks b heraus, mit diesem wird der
                  Ausdruck i+... ausgewertet und das
                  Ergebnis der Variablen j zugewiesen! */
  ...
}
```

- Member-Funktionen werden (wie bei Komponenten einer Struktur üblich) mittels des Operators `.` (bei Zeigern ggf. mittels `->`) aufgerufen.
`a.push(7)` bedeutet: rufe für den Kellerspeicher `a` die Funktion `push` mit dem Argument 7 auf. Entsprechend bedeutet `b.pop()`: rufe für den Kellerspeicher `b` die Funktion `pop` (natürlich ohne Argumente) auf.

- Bei der Erzeugung der Kellerspeicher:

```
Stack a, b;
```

wird jeweils der Konstruktor aufgerufen, d.h. für den Kellerspeicher `a` wird der zugehörige Stack-Pointer `a.sp` auf 0 gesetzt und für den Kellerspeicher `b` entsprechend (jeder Kellerspeicher hat eine eigene Komponente `sp`!).

- Da die Member-Funktionen `push` und `pop` im öffentlichen Zugriffsabschnitt (`public`) stehen, kann die Anwendung auf diese Komponenten zugreifen.
 Die (Daten-)Komponenten `int sp;` und `int keller[100];` stehen nicht im öffentlichen Zugriffsabschnitt, deswegen kann die Anwendung nicht auf diese Komponenten zugreifen — der entsprechende Versuch führt zu einer Compiler-Fehlermeldung:

```
#include "Stack.h"
...
Stack a, b;
...
a.sp = 10000;          // FEHLER: sp ist nicht public!!!
...
cout << b.keller[0];  // FEHLER: keller ist nicht public!!!
...
```

4.2 Klassen/Objekte

4.2.1 Grundlagen

Eine Klasse ist ein abstrakter Datentyp, eine Zusammenfassung von Daten und Funktionen:

```
struct Stack {
    protected:
        int keller[100]; // Datenkomponenten
        int sp;
    private:
        Stack();          // Funktionskomponenten
        void push(int);
        int pop(void);
};
```

} Datentyp, Klasse

Innerhalb der geschweiften Klammern können also Daten definiert und Funktionen deklariert werden. Das durch diese geschweifte Klammern Eingeschlossene heißt auch *Klassenrumpf*. Obwohl durch diesen Klassenrumpf kein einziges Bit im Speicher verbraucht wird, spricht man trotzdem von *Klassendefinition*. (Ich unterscheide nicht zwischen den Begriffen *Klassendeklaration* und *Klassendefinition*.)

Die im Klassenrumpf deklarierten Funktionen (Member-Funktionen, Methoden) müssen natürlich irgendwo (unter Bezugnahme auf die Klasse) definiert sein.

Objekte sind “Variablen“ von einem solchen Abstrakten Datentyp (Klasse):

```
Stack a, b, c;                                } Variablen, Objekte
```

Wie bei Strukturen wird bei einem Objekt (Variable vom Struktur-Typ) mittels der Operatoren `.` oder `->` auf die Komponenten zugegriffen:

```
Stack a, b, *p = &b;
...
a.push(7);    // rufe fuer Objekt a die Methode push auf
p->pop();      // rufe fuer das Objekt, auf welches p zeigt,
               // die Methode pop auf
...
```

4.2.2 Zugriffsschutz

Information-Hiding wird durch Zugriffsabschnitte realisiert.

- Der **public**-Zugriffsabschnitt stellt die Anwenderschnittstelle dar. Auf die hier aufgeführten Komponenten (Daten oder Funktionen) kann der Anwender zugreifen.
- Im **protected**- und **private**-Zugriffsabschnitt sind Implementierungsdetails der Klasse untergebracht. Auf die hier aufgeführten Komponenten kann der

Anwender nicht zugreifen. (Der Unterschied zwischen `protected` und `private` kommt erst im Zusammenhang mit Vererbung zum Tragen.)

Bei der Klassendeklaration kann ein Zugriffsabschnitt mehrfach genannt werden:

```
struct A {
    public:
        ...    // oeffentliche Komponenten
    private:
        ...    // private Implementationsdetails
    public:
        ...    // wiederum oeffentlich
    ...
};
```

Das in C++ neue Schlüsselwort `class` ist wie `struct` zu verwenden, der (von Vererbung abgesehen) einzige Unterschied ist der, dass bei `struct` Komponenten, die nicht explizit einem Zugriffsabschnitt zugeordnet sind, implizit `public`, also öffentlich sind — und bei `class` sind sie `private`:

```
struct A {
    int f(void);    // implizit public
    char b;        // implizit public
private:
    int sp;        // explizit private
    void g(int);   // explizit private
public:
    void h(int);   // explizit public
    ...
};

...
class B {
    int f(void);    // implizit private
    char b;        // implizit private
private:
    int sp;        // explizit private
    void g(int);   // explizit private
public:
    void h(int);   // explizit public
    ...
};
```

Es gibt unterschiedliche Stile bei der Verwendung von `class` oder `struct` und der Reihenfolge der Zugriffsabschnitte.

Der von mir bevorzugte Stil:

- standardmäßige Verwendung von `class`,

- nur die Komponenten, die ich wirklich `public` haben möchte, schreibe ich in den `public`-Zugriffsabschnitt,
- den `public`-Teil schreibe ich an den Anfang der Klassendeklaration (damit ein Anwender der Klasse beim Durchlesen der Deklaration zunächst die für ihn wichtige öffentliche Schnittstelle sieht, Implementationsdetails sieht er dann erst später).
- in die öffentliche Schnittstelle schreibe ich im Allgemeinen ausschließlich Funktionskomponenten und keine Datenkomponenten. (Zugriff auf interne Datenkomponenten kann durch geeignete Funktionen gewährleistet werden!)

4.2.3 Konstruktoren

Konstruktoren haben den gleichen Namen wie die Klasse und keinen Rückgabetypen, können aber Parameter aufweisen.

Konstruktoren werden automatisch immer bei der Erzeugung eines Objektes der Klasse aufgerufen und müssen ggf. dafür sorgen, dass das Objekt konsistent initialisiert wird.

Im Stack-Beispiel:

```
...
#include "Stack.h"
...
Stack a, b, c;           // 3 Konstruktoraufrufe, je einer
...                     // fuer a, b und c
Stack stackfeld[100];    // 100 Konstruktoraufrufe, je einer
...                     // fuer jedes Feldelement
Stack *p;               // KEIN Konstruktoraufruf (Zeiger)
...
p = new Stack;           // ein Konstruktoraufruf fuer dynamisch
...                     // erzeugten Stack
p = new Stack[100];      // 100 Konstruktoraufrufe, je einer fuer
...                     // dynamisch erzeugtes Feldelement
...
// VORSICHT: KEIN Konstruktoraufruf bei malloc etc.!!!
p = ( Stack *) malloc( sizeof( Stack) );
p = ( Stack *) malloc( 100 * sizeof( Stack) );
...
```

Ist kein eigener Konstruktor definiert, wird vom System der sogenannte *Standardkonstruktor* zur Verfügung gestellt, der nur dafür sorgt, dass für die Datenkomponenten ausreichend Speicher reserviert wird. Dieser Speicher wird nicht explizit initialisiert. (Bei automatischen Objekten ist der Speicherinhalt somit zufällig, bei statischen und globalen Variablen ist der Speicher mit 0-Bits belegt!)

4.2.4 Implementierungsmöglichkeiten

Üblicherweise schreibt man die Klassendeklaration in eine Headerdatei (etwa: `Stack.h`) und die Implementierung der zugehörigen Member-Funktionen in eine Implementierungsdatei (etwa: `Stack.cc`) und stellt dem Anwender die Headerdatei (`Stack.h`) und die übersetzte Implementierungsdatei (`Stack.o`) zur Verfügung.

Hierzu gibt es Alternativen:

- Die Member-Funktionen werden in der Headerdatei im Klassenrumpf gleich vollständig definiert (nicht nur deklariert):

```
/* Definition der Member-Funktionen in Headerdatei
   im Klassenrumpf: implizit inline */
#ifndef _Stack_h
#define _Stack_h

#include <iostream>
#include <cstdlib>

struct Stack { // neuer Datentyp hat Namen: Stack

    protected:    // Impl.-Details, nicht oeffentlich
        int keller[100];
        int sp;

    public:        // oeffentliche Schnittstelle

        Stack()    // benoetigt, um Stack zu erzeugen
        { ... }    // gleich definiert
        void push(int i) // Funktion zum Einkellern
        { ... }    // gleich definiert
        int pop(void) // Funktion zum Auskellern
        { ... }    // gleich definiert
    };

#endif
```

Die so definierten Member-Funktionen werden implizit vom System als `inline` aufgefasst, so dass diese Headerdatei in unterschiedlichen, zu einem Projekt gehörenden Übersetzungseinheiten eingebunden werden kann. (Mehrfaches Einbinden in eine Übersetzungseinheit ist zu verhindern — `#ifndef ...!`)

Eine separate Implementierung der Funktionen ist dann nicht mehr notwendig, der Anwender benötigt nur diese Headerdatei.

- Die Member-Funktionen werden im Klassenrumpf (in der Headerdatei) deklariert, aber zusätzlich in der Headerdatei auch, und zwar explizit als `inline`, definiert (wobei man bei der Definition wiederum auf die Klasse Bezug nehmen muss):

```

/* Definition der Member-Funktionen in Headerdatei
   ausserhalb des Klassenrumpfes: explizit inline */
#ifndef _Stack_h
#define _Stack_h

#include <iostream>
#include <cstdlib>

struct Stack { // neuer Datentyp hat Namen:  Stack

    protected:    // Impl.-Details, nicht oeffentlich
        int keller[100];
        int sp;

    public:        // oeffentliche Schnittstelle
        // Deklaration der Member-Funktionen
        Stack();    // benoetigt, um Stack zu erzeugen
        void push(int); // Funktion zum Einkellern
        int pop(void); // Funktion zum Auskellern
};

// Definition der Member-Funktionen
inline Stack::Stack() { ... }

inline void Stack::push(int i) { ... }

inline int Stack::pop(void) { ... }
#endif

```

Durch die explizite `inline`-Definition der Member-Funktionen wird wiederum das Einbinden der Headerdatei in unterschiedliche zu einem Projekt gehörenden Übersetzungseinheiten möglich. (Mehrfaches Einbinden in eine Übersetzungseinheit ist zu verhindern — `#ifndef ...!`)

Auch hier braucht dem Anwender nur diese Headerdatei zur Verfügung gestellt zu werden.

- In der Praxis sind natürlich auch Mischformen möglich, d.h.
 - In der Headerdatei sind einige (“kleine“) Member-Funktionen gleich definiert, entweder im Klassenrumpf und somit implizit `inline` oder außerhalb des Klassenrumpfes, dann aber explizit `inline`.
 - Die restlichen (“großen“) Member-Funktionen sind in einer separaten Implementationsdatei definiert.

Der Anwender benötigt in diesem Fall natürlich wieder die Headerdatei und die (übersetzte) Implementationsdatei.

4.2.5 Klassen als Softwarebausteine

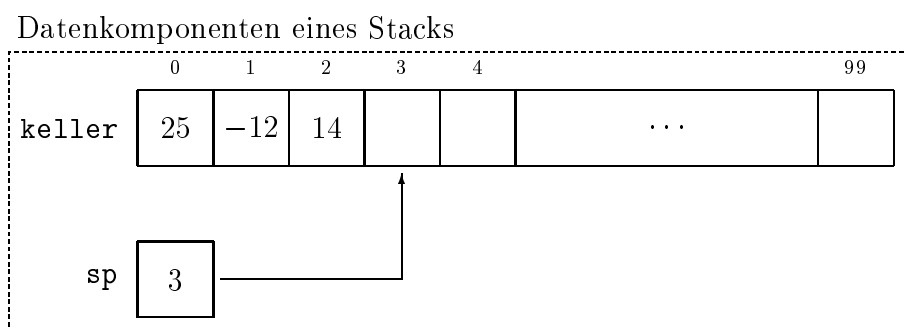
Der `public`-Teil einer Klasse stellt die öffentliche Schnittstelle zur Klasse dar, der `protected` und `private`-Teil die Implementierungsdetails.

Der Anwender kann nur auf die öffentliche Schnittstelle zugreifen — er kann seine Anwendung schreiben und hierbei die öffentlichen Komponenten verwenden — die Implementierungsdetails sind für ihn unerheblich.

Sollten die Implementierungsdetails der Klasse abgeändert werden (Änderungen am `private` oder `protected`-Teil) oder, unter Beibehaltung der Schnittstellendeklaration, die Implementierung der Member-Funktionen, so braucht der Anwender seine Anwendung nur mit der neuen Klassendeklaration (Headerdatei) neu zu übersetzen und mit der neuen Implementierungsdatei zu linken und seine Anwendung sollte nach wie vor funktionieren!

Die interne Realisierung der Klasse könnte gänzlich verändert werden, ohne dass die Anwendung (nach Neuübersetzung) hiervon in Mitleidenschaft gezogen würde!

Im bisherigen Stack-Beispiel ist die interne, für den Benutzer weitgehend unerhebliche Realisierung des Stacks wie folgt:



Im Beispiel könnte man den Kellerspeicher ganz anders, etwa über eine dynamisch wachsende Lineare Liste realisieren (und hätte hierbei das Problem des Überlaufs weitgehend eliminiert!):

Headerdatei:

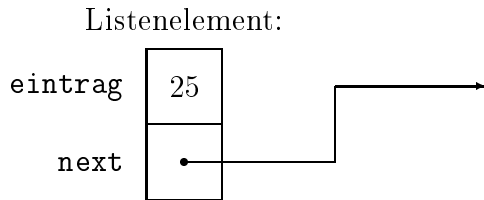
```
#ifndef _Stack_h
#define _Stack_h

class Stack { // neuer Datentyp hat Namen: Stack
protected: // Impl.-Details, nicht oeffentlich
    struct listel { // eingebetteter Typ:
        int eintrag; // Listenelement
        listel *next;
    } *p; // Zeiger auf Listenanfang

public: // oeffentliche Schnittstelle
    // wie bei Feld-Realisierung
    Stack(); // benoetigt, um Stack zu erzeugen
    void push(int); // Funktion zum Einkellern
    int pop(void); // Funktion zum Auskellern
};
#endif
```

Die Deklaration der in der öffentlichen Schnittstelle stehenden Funktionen ist gleich der früheren Feld-Realisierung.

Im `protected`-Teil steht jedoch etwas völlig anders, nämlich zunächst mal die Deklaration eines eingebetteten Hilfstypes `listel`, der den typischen Aufbau eines Listenelementes einer Linearen Liste mit `int`-Einträgen repräsentiert:



Die einzige Datenkomponente dieser `Stack`-Realisierung ist ein Zeiger auf den Anfang einer mittels des Hilfstypes aufgebauten Linearen Liste. (Der noch zu implementierende Konstruktor muss natürlich dafür sorgen, dass die Liste zunächst leer ist!)

Die Implementierung der Funktionen ist jetzt völlig anders — sie muss zur Linearen Liste passen, also neue Elemente vorne in der Liste einfügen und die Entnahme von Elementen aus der (nicht leeren!) Linearen Liste erfolgt auch vorne:

Implementationsdatei:

```
// Stack.cc: Listen-Implementierung des Datentypes Stack
// Eigene Headerdatei einbinden
#include "Stack.h"
#include <iostream>    // wegen Ein-/Ausgabe
#include <cstdlib>     // wegen exit()

using namespace std;

Stack::Stack()
{ p = 0;    // Liste zunaechst leer
}

void Stack::push(int wert)
{ listel * tmp = new listel; // neues Listenelement
    // dynamisch anfordern

    tmp -> eintrag = wert; // Wert uebernehmen
    tmp -> next    = p;    // Listenelement vorne in
    p              = tmp;  // Liste einfuegen
}

int Stack::pop(void)
{ if ( p == 0) // Liste leer?
    { cerr << "Keller leer" << endl;
      exit(-1);
    }
}
```



```

    int i = p -> eintrag; // Wert zwischenspeichern

    listel *tmp = p;      // erstes Listenelement
    p = p -> next;        // loeschen
    delete tmp;

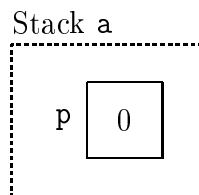
    return i; // zwischengespeicherten Wert
}           // zurueckgeben

```

Der Konstruktor erzeugt somit einen zunächst leeren Stack, d.h. die verwendete Lineare Liste ist zunächst leer — die Situation im Speicher nach Stack-Erzeugung mit dem Konstruktor:

```
stack a;
```

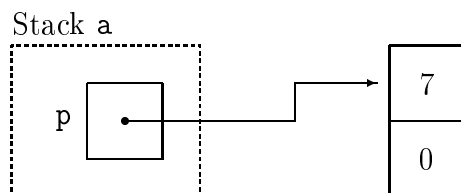
kann man sich dann etwa so vorstellen:



Nach dem Einkellern eines Elementes:

```
a.push(7);
```

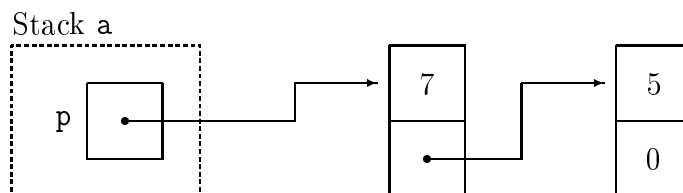
sieht die interne Struktur dann so aus:



Wird ein weiteres Element eingekellert:

```
a.push(5);
```

sieht es dann wie folgt aus:

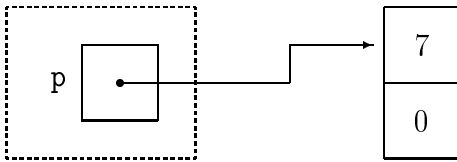


Nach dem Auskellern eines Elementes:

```
... a.pop();
```

ist die Situation dann wieder wie vormal:

Stack a



Im Rahmen der üblichen Verwendung eines Stacks kann aus einer Anwendung, welche einen Stack benötigt und die erste Feld-Realisierung des Stack verwendete, die Feld-Realisierung durch die Listen-Realisierung “ausgetauscht” werden, ohne dass die Anwendung angepasst werden müsste! (Der Softwarebaustein: *Kellerspeicher, als Feld realisiert* kann also relativ problemlos durch den Baustein *Kellerspeicher, als Lineare Liste realisiert* ausgetauscht werden!)

4.2.6 Destruktoren

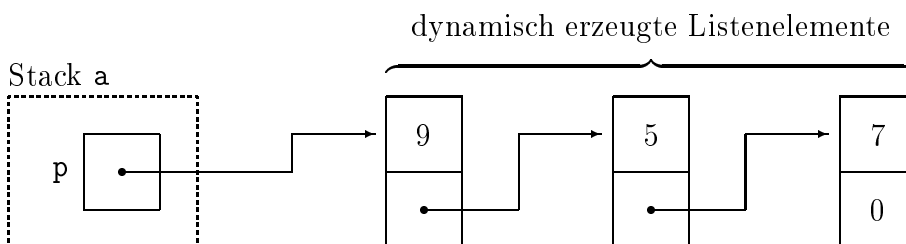
Die oben beschriebene Realisierung eines Stacks über eine Lineare Liste hat noch einen gravierenden “Schönheitsfehler”, wie folgende “lokale” Anwendung des Stacks zeigt:

```
#include "Stack.h"
...
void fkt(void)
{ Stack a;
  a.push(7);
  a.push(5);
  a.push(9);

  return;
}
```

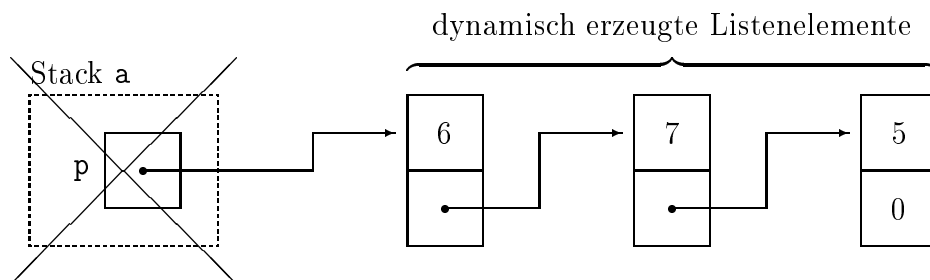
Beim Aufruf der Funktion wird der lokale Stack **a** erzeugt, hierbei über den implizit aufgerufenen Konstruktor initialisiert (ist also zunächst leer) und anschließend werden 3 Elemente “eingekellert”.

Die Situation vor dem Ende der Funktion (vor dem **return**) sieht also wie folgt aus:



anschließend ist die Funktion zu Ende, und der Speicherplatz für die lokalen Variablen (Objekte) wird wieder freigegeben, nicht jedoch der dynamisch angeforderte Speicher. D.h. der Speicher für die Komponenten des Stacks **a** wird zerstört, insbesondere auch die (**protected**) Komponente **a.p**, nicht jedoch die dynamisch angeforderten Listenelemente der eigentlichen Linearen Liste.

Die Situation nach dem Funktionsende ist also wie folgt aus:



d.h. der Rest der Linearen Liste bleibt erhalten, wobei jedoch kein Bezug mehr auf den Anfang der Linearen Liste existiert! Es bleibt also im Speicher “Speichermüll“ übrig, der zwar die Anwendung nicht direkt stört, aber im Laufe des Programms, wenn mehrfach derartige Funktionen aufgerufen werden, zu Speicherplatzproblemen führen kann!

Dual zu der (automatischen) ordnungsgemäßen Initialisierung eines Objektes bei seiner Erzeugung durch den Konstruktor müsste am Ende der Lebenszeit eines Objektes dieses (automatisch) ordnungsgemäß “zerstört“ werden.

Genau dieses ist die Aufgabe des sogenannten *Destructors*.

Dieser wird beim Ende der Lebenszeit eines Objektes automatisch aufgerufen und kann durch den Entwickler einer Klasse definiert werden. (Das System stellt den sog. *Standarddestructor* zur Verfügung, der nur den Speicherbereich der Datenkomponenten des Objektes wieder freigibt — dies ist aber, wie hier zu sehen, manchmal zu wenig!)

Als Namen hat der Destructor den Namen der Klasse, dem ein “~“ vorangestellt wird. Wie Constructoren hat ein Destructor keinen Ergebnistyp (auch nicht `void`).

Ein Destructor hat keine Parameter (Constructoren können Parameter haben!).

In obigem Beispiel des über eine Lineare Liste realisierten Kellerspeichers müsste der Destructor dafür sorgen, dass der bei der Stack-Verwendung dynamisch angeforderte Speicherbereich für die dynamisch angeforderten Listenelemente ordnungsgemäß wieder freigegeben wird. (Diese Stack-Klasse ist eine sog. Klasse mit *dynamischen Komponenten*.)

Deklaration der Klasse und Definition des Destructors müssen also wie folgt aussehen: Headerdatei:

```
#ifndef _Stack_h
#define _Stack_h

class Stack {
protected:
    ... // wie oben
public:
    ... // wie oben
    // zusätzlich: Dekl. des Destructors
    ~Stack();
};
#endif
```

Implementationsdatei:

```
... // wie oben

// zusaetzlich: Definition des Destruktors
Stack::~~Stack()
{ listel *tmp;

  while ( (tmp = p ) != 0)
  { p = p -> next;
    delete tmp;
  }
}
```

Wie Konstruktoren werden auch Destruktoren vom System automatisch aufgerufen:

```
#include "Stack.h"

void fkt(void)
{ Stack a, b, c;           // 3 Konstruktoraufrufe
  Stack stackfeld[100];    // 100 Konstruktoraufrufe
  Stack *p, *q;

  p = new Stack;           // ein Konstruktoraufruf
  q = new Stack[20];       // 20 Konstruktoraufrufe
  ...
  delete p;                // ein Destruktoraufruf
  delete[] q;              // 20 Destruktoraufufe
  ...
  return;                  // 103 Destruktoraufufe, je einen fuer a, b und c
}                           // und je einen fuer jedes Feldelement von stackfeld!
```

Durch diese Einführung des Destruktors in die Klasse **Stack** hat sich natürlich die öffentliche Schnittstelle zur Klasse geändert (Destruktoren gehören wie Konstruktoren sinnvollerweise immer in den **public**-Teil, da sie — wenn auch implizit — von der Anwendung aufgerufen werden).

Diese Listenrealisierung des Kellerspeichers ist somit nicht mehr ganz kompatibel zur ursprünglichen Feldrealisierung, die ja ohne selbstdefinierten Destruktor auskam (das Feld `int keller[100];` ist ja eine gewöhnliche Datenkomponente und wird durch den Standarddestruitor korrekt zerstört!).

Zur “Reparatur“ könnte man die Feldrealisierung auch mit einem Destruktor versehen, damit beide Schnittstellen übereinstimmen. Die Implementierung dieses Destruktors könnte leer sein, da ja keine Funktionalität benötigt wird:

```
#ifndef _Stack_h
#define _Stack_h

struct Stack { // neuer Datentyp hat Namen:  Stack
```

```

protected:    // Impl.-Details, nicht oeffentlich
    int keller[100];
    int sp;

public:        // oeffentliche Schnittstelle

    Stack();    // benoetigt, um Stack zu erzeugen
    void push(int); // Funktion zum Einkellern
    int pop(void); // Funktion zum Auskellern

    // leerer Destruktor, gleich ganz definiert:
    ~Stack() { }
};

#endif

```

(Implementierung der übrigen Funktionen wie ursprünglich!)

4.2.7 Member-Funktionen

Member-Funktionen sind zu einer Klasse gehörende Funktionen und werden in einer Anwendung nur für konkrete Objekte aufgerufen:

```

#include "Stack.h"
...
Stack a, b, c;
Stack *zeiger;
...
a.push(7); // rufe fuer Objekt a die Funktion push
           // mit dem Argument 7 auf
...
... = b.pop(); // rufe fuer das Objekt b die Funktion pop auf
...
zeiger->push(5); // rufe fuer das Objekt, auf welches zeiger zeigt,
                // die Funktion push mit Argument 5 auf
...
zeiger.pop();   // rufe fuer das Objekt, auf welches zeiger zeigt,
                // die Funktion pop auf
...

```

Beim Ablauf einer Member-Funktion gibt es also immer das “aktuelle Objekt“, für welches die Funktion aufgerufen wurde! (Bei `a.push(7)` ist `a` das aktuelle Objekt, bei `b.pop()` ist `b` das aktuelle Objekt!)

Dieses aktuelle Objekt liegt der Abarbeitung der Member-Funktion zugrunde und auf die Komponenten dieses aktuellen Objektes kann innerhalb der Member-Funktion direkt durch Angabe des Komponentennamens zugegriffen werden (es kann hier auf alle

Komponenten des aktuellen Objektes, unabhängig vom Zugriffsabschnitt zugegriffen werden!):

```
// Keller-Speicher als Lineare Liste,
// Definition der Member-Funktion push:
void Stack::push(int wert)
{ ...
    tmp -> next = p;
    //           ^
    ...
}
```

an der durch den Pfeil \wedge gekennzeichneten Stelle wird auf die Komponente p zugegriffen.

Wird diese Member-Funktion für das Objekt a aufgerufen, also $a.push(\dots)$, so wird über dieses p auf die Komponente $a.p$ zugegriffen.

Wird diese Member-Funktion für ein anderes Objekt b aufgerufen, etwa $b.push(\dots)$, so wird über dieses p auf die Komponente $b.p$ zugegriffen.

(Innerhalb der Definition einer Member-Funktion könnte auch einfach durch Angabe deren Namens eine andere Member-Funktion aufgerufen werden. Diese andere Member-Funktion wird dann für das aktuelle Objekt aufgerufen!)

In gewisser Hinsicht hat eine Member-Funktion bei Ihrem Aufruf neben den in runden Klammern stehenden Argumenten implizit ein weiteres Argument, nämlich das aktuelle Objekt, für welches sie aufgerufen wurde!

4.2.8 Der `this`-Zeiger

Wie im letzten Abschnitt erläutert, kann innerhalb einer Member-Funktion auf jede Komponente des aktuellen Objektes — das Objekt, für welches die Member-Funktion aufgerufen wurde — zugegriffen werden.

Innerhalb einer Member-Funktion zu einer Klasse A (`struct A` oder `class A`) gibt es automatisch einen konstanten Zeiger auf ein Objekt vom Typ A — und dieser Zeiger hat den Namen `this`.

Die exakte Definition dieses Zeigers müsste also lauten:

```
A * const this;
```

Dieser Zeiger zeigt innerhalb der Member-Funktion auf das aktuelle Objekt, und da der Zeiger `const` ist, kann dieser Zeiger auch nicht geändert werden (das, worauf der Zeiger zeigt — also das aktuelle Objekt — kann über den Zeiger sehr wohl verändert werden!).

In einer Member-Funktion könnte man über diesen `this`-Zeiger auf Komponenten (Daten oder Funktionen) des aktuellen Objektes zugreifen:

```
...this->Komponentenname...
```

(Hier würde es der reine Komponentename auch tun!)

Über dieses `this`-Zeiger hat man aber auch Zugriff auf das ganze aktuelle Objekt, etwa, um das aktuelle Objekt als Funktionsergebnis zurückzugeben:

```

class A {
    ...
    public:
        A& A_fkt(...); // A-Member-Funktion, welche Referenz auf
                        // ein A zurueckgibt
    ...
};

// Definition dieser Funktion
A& A::A_fkt(...)
{ ...
    return *this; // gebe aktuelles Objekt zurueck
}

```

oder eine andere Funktion mit dem aktuellen Objekt (oder seiner Adresse) als Argument aufzurufen:

```

class A {
    ...
    public:
        void A_fkt(void);
    ...
};

...
// globale Funktion
void glob_fkt( A *a)
{ ... }

...
// Definition der A-Member-Funktion
void A::A_fkt(void)
{ ...
    glob_fkt(this); // rufe globale Funktion mit Adresse des
                    // aktuellen Objektes als Argument auf
    ...
}

```

4.2.9 Konstante Member-Funktionen

Wie oben gesehen hat eine Member-Funktion implizit ein zusätzliches Argument — nämlich das Objekt, für welches sie aufgerufen wurde!

Dieses Argument taucht nicht in der Argumentliste der Funktion innerhalb der runden Klammern auf, sondern steht (i. Allg.) beim Aufruf der Member-Funktion vor dem Funktionsnamen:

aktuelles_Objekt.Member_Funktions_Name(weitere Argumente);

Bei Deklaration und Definition der Member-Funktion existiert somit auch kein expliziter Parameter, welcher zum aktuellen Objekt korrespondiert.

Gleichwohl ist das aktuelle Objekt (per Referenz, nicht als Kopie) in der Member-Funktion verfügbar — man kann ja über die Komponentennamen auf einzelne Komponenten oder über den `this`-Zeiger auf das ganze Objekt zugreifen!

Bei gewöhnlichen Parametern vom Referenz- oder Zeigertyp ist es, wie gesehen, bedeutsam, ob die Referenz bzw. Zeiger Referenz bzw. Zeiger auf `const` sind oder nicht — man kann/muss die entsprechende Qualifikation in der Parameterliste bei Funktionsdeklaration und Funktionsdefinition angeben:

```
void fkt1( T &a)           // a: Referenz auf T
{ ... }
void fkt2( const T &b)     // b: Referenz auf const T
{ ... }
void fkt3( T *a)           // a: Zeiger auf T
{ ... }
void fkt4( const T *b)     // b: Zeiger auf const T
{ ... }
```

In `fkt1` und `fkt3` wird das Argument möglicherweise geändert, diese Funktionen können also nicht mit Konstanten oder Ausdrücken als Argument aufgerufen werden; in `fkt2` und `fkt4` wird das Argument nicht geändert, diese Funktionen können also auch mit Konstanten oder Ausdrücken als Argument aufgerufen werden.

Auch für die in Member-Funktionen vorhandene Referenz auf das aktuelle Objekt (über den `this`-Zeiger zugreifbar) kann bei Deklaration und Definition der Member-Funktion vereinbart, werden, dass diese Referenz eine Referenz auf `const` ist. Dies geschieht wie durch das Schlüsselwort `const` im Anschluss an die Parameterliste der Funktion folgt:

```
...
class A {
private:
    // irgendeine Komponente:
    int a;
    ...
public:
    // Deklaration:
    int fkt(void);           // normale Funktion
    int c_fkt(void) const;   // konstante Member-Funktion
    ...
};

// Definition:
int A::fkt( void )
{ ... }
int A::c_fkt( void ) const
{ ... }
...
```


Der `this`-Zeiger in einer derartigen *konstanten Member-Funktion* `c_fkt` hat dann den Typ:

```
A const * const this;
```

d.h. er ist `const` und zeigt auf `const`.

Durch eine konstante Member-Funktion können somit (i.Allg., siehe Abschnitt 4.2.9) die Komponenten des aktuellen Objektes nicht geändert werden, folgender Implementierungsversuch führt zu einer Compiler-Fehlermeldung:

```
int    A::fkt( void )
{ ...
    a = ...;      // OK: Komponente a kann geaendert werden!
    ...
}
int    A::c_fkt( void ) const
{ ...
    a = ...;      // FEHLER: Komponente a kann nicht geaendert werden!
    ...
}
```

Eine solche konstante Member-Funktion kann somit auch für konstante Objekte (vom Typ `A`) aufgerufen werden oder für beliebige Ausdrücke, welche ein Resultat vom Typ `A` haben (etwa eine Funktion mit `A`-Ergebnis):

```
...
A glob_fkt(void);      // Funktion mit A--Ergebnis
...
A a;                  // variables Objekt
A const b;            // konstantes Objekt
...
a.fkt();              // OK
b.fkt();              // FEHLER: b const, fkt nicht!
...
b.c_fkt();            // OK: b const, c_fkt auch!
a.c_fkt();            // auch OK: a zwar nicht const, wird aber durch
                      // c_fkt nicht geaendert
...
glob_fkt().fkt();      // FEHLER: kann auf Ergebnis von glob_fkt() nicht
                      // die Funktion fkt anwenden!
glob_fkt().c_fkt();    // OK: Ergebnis von glob_fkt ist vom Typ const A,
                      // wende hierauf Funktion c_fkt an!
...
```

Das Schlüsselwort `mutable`

Bei einer komplexeren Anwendung kann es vorkommen, dass ein konstantes Objekt einer Klasse `A` eine Komponente besitzt, deren Wert zunächst aufwändig berechnet werden muss!

Manchmal ist dies bei Erzeugung des (konstanten) Objektes (durch einen Konstruktor) noch nicht möglich — etwa, weil hierzu erst Daten von Datei gelesen werden müssten, die Datei aber erst später zur Verfügung steht.

Die Angabe des Schlüsselwortes `mutable` bei einer Komponente einer Klasse `A` bewirkt, dass diese Komponente “veränderbar” ist, selbst bei “konstanten” Objekten.

Eine solche `mutable`-Komponente könnte sogar durch eine konstante Member-Funktion verändert werden.

Beispiel:

Datentyp zur Repräsentierung eines Datums. Intern als großer ganzzahliger Wert (Anzahl der Sekunden seit 1.1.1970 0 Uhr), bei Bedarf wird ein String mit dem Datum erstellt:

```
#include <iostream>
#include <ctime>
#include <cstring>

class Datum {
private:
    time_t internes_Datum;
    mutable char * Datum_als_String;
    ...
public:
    Datum(int);
    long Datum_intern(void) const;
    const char * Datum_String(void) const;
    ...
};

Datum::Datum(int i)
{ if ( i < 0 ) // heutiges Datum ermitteln
    internes_Datum = time(NULL);
  else
    internes_Datum = i;

  Datum_als_String = 0;
}

long Datum::Datum_intern(void) const
{ return static_cast<long> (internes_Datum); }

const char * Datum::Datum_String(void) const
{ // Falls String-Repraesentierung bereits vorhanden,
  // gib diese zurueck:
  if ( Datum_als_String != 0 )
    return Datum_als_String;

  // ansonsten: String-Repraesentierung ermitteln:
```

```

Datum_als_String = new char[64];
// hier wurde mutable-Komponente veraendert!

strcpy(Datum_als_String, ctime(&internes_Datum));

return Datum_als_String;
}
...

```

4.2.10 Verwenden von Klassen

Objekte einer Klasse `class A { ... }`; können wie gewöhnliche Variablen verwendet werden.

Man kann insbesondere:

1. in einer Anwendung mehrere Objekte der Klasse vereinbaren:

```

A a,b,c;           // 3 variable A-Objekte
const A d, e, f;   // 3 konstante A-Objekte

```

2. Felder definieren:

```

A A_Feld[100];     // 100 variable A-Objekte

```

3. Referenzen auf Objekte vereinbaren:

```

A a, &b = a;       // b ist Referenz auf a

```

4. Adressvariablen vom Klassentyp verwenden:

```

A *ap;             // ap ist Zeiger auf ein A

```

allerdings empfehlen Experten, solche Zeiger besser nicht mit dem symbolische Wert `NULL` zu vergleichen — sondern eher mit dem `int`-Wert `0`.

In vielen Fällen ist eine Bedingung folgender Art noch intuitiver:

```

...
if ( ap ) // wahr, falls ap auf ein g"ultiges Objekt zeigt!
{ ... }
...

```

5. Objekte/Felder dynamisch reservieren:

```

A *ap1 = new A;          // dynamisches A-Objekt
A *ap2 = new a[100];     // dynamisches A-Feld
...
delete ap1;
delete[] ap2;

```

6. Objekte per Wert, Referenz oder Adresse an Funktionen übergeben:

```

void fkt1( A);           // Uebergabe per Wert
void fkt2 ( A&);         // Uebergabe per Referenz
void fkt3( const A&);    // Referenz auf const
void fkt4( A*);          // Zeiger auf A
void fkt5(const A*);     // Zeiger auf const A

```

7. Funktionen mit einem Objekt als Funktionsergebnis (als Wert, Referenz oder Adresse) definieren:

```

A      fkt1(void);  // Ergebnis: A-Wert
A&     fkt2(void);  // Ergebnis: Referenz auf A-Objekt
const A& fkt3(void); // Ergebnis: Referenz auf const A-Objekt
A*     fkt4(void);  // Ergebnis: Adresse eines A-Objektes
const A* fkt5(void); // Ergebnis: Adresse eines const A-Objektes

```

8. Man kann natürlich auch Objekte einer Klasse als Komponente in einer anderen Klasse verwenden (die verwendete Klasse muss zuvor zumindest vollständig deklariert sein!):

```

class B { // neue Klasse
private:
    ...
    A a_Komponente;
    ...
public:
    ...
};

```

Hier muss man allerdings direkte oder indirekte Rekursion vermeiden: ein Objekt der Klasse A darf weder direkt noch indirekt eine Komponente von Typ A haben!

Eine Komponente vom eigenen Adresstyp ist jedoch möglich:

```

class B { // neue Klasse
private:
    ...
    B * b_zeiger;
    ...
public:
    ...
};

```

9. Möchte man eine Klasse **A** in einer anderen Klasse **B** als Komponente verwenden, so kann die verwendete Klasse **A** global vereinbart werden:

```
class A { ... };

class B {
    private:
        ...
        A a-Komponente;
        ...
    public:
        ...
};
```

oder als Hilfsklasse in der verwendenden Klasse vereinbart werden:

```
class B {
    private:
        ...
        class A {
            ...
            // Deklaration einer Member--Funktion der Hilfsklasse
            int fkt(void);
            ...
        } a_Komponente;
        ...
    public:
        ...
};
```

In diesem Fall müssen die Member-Funktionen der Hilfsklasse **A**, wenn sie außerhalb des Klassenrumpfes von **A** implementiert werden, wie folgt definiert werden:

```
// Definition einer Member--Funktion der Hilfsklasse:
int B::A::fkt(void)    // fkt ist Member-Funktion der Hilfsklasse A
{ ... }               // der Klasse B
```

In beiden Fällen verwendet die Klasse **B** die Klasse **A**, d.h. jedes **B**-Objekt hat eine Komponente vom Typ **A**, die **B**-Member-Funktionen können aber nur auf die Schnittstelle — also den **public**-Teil — der (eigenen) **A**-Komponente zugreifen!

10. Natürlich können Member-Funktionen einer dritten Klasse **C** lokale Variablen (Objekte) oder auch Parameter vom Typ **A** haben:

```

class C { // neue Klasse
public:
    ...
    void f( A );    // C-Methode mit A-Parameter
    void g(void);   // weitere C-Methode
    ...
private:
    ...
};

void C::g(void)
{
    A tmp;         // lokales A-Objekt
    ...
}

```

Auch hier treten die Methoden der Klasse *C* *nur* als “Anwender“ der Klasse *A* auf, d.h. hier ist nur der Zugriff auf die *A*–Schnittstelle möglich.

11. Member–Funktionen einer Klasse *C* können natürlich auch Parameter oder lokale Variablen vom eigenen Typ, also vom Typ *C* haben:

```

class C { // neue Klasse
public:
    ...
    void f( C );    // C-Methode mit C-Parameter
    void g(void);   // weitere C-Methode
    ...
private:
    ...
};

void C::g(void)
{
    C tmp;          // lokales C-Objekt
                    // in dieser Funktion sind (mind.) zwei C-Objekte
                    // bekannt:  das aktuelle Objekt (*this) und tmp!
    ...
}

```

In diesem Fall haben die Member–Funktionen Zugriff auf alle Komponenten der beteiligten *C*–Objekte (Parameter oder lokale Objekte)!

4.3 Konstruktoren im Detail

Wie bereits gesehen gibt es zu jeder Klasse (mind.) einen *Konstruktor*, der automatisch beim Erzeugen eines Objektes aufgerufen wird. Konstruktoren haben den

gleichen Namen wie die zugehörige Klasse, keinen Ergebnistyp (auch nicht `void`), können aber Parameter haben. Im Sinne von Funktionsüberladung kann es zu einer Klasse mehrere, sich in ihrer Signatur unterscheidende Konstruktoren geben!

Wird vom Entwickler einer Klasse nichts anderes vorgesehen, stellt das System folgende beiden Konstruktoren zur Verfügung:

4.3.1 Standard-Parameterloser-Konstruktor

Zu einer Klasse `A` hat dieser den Typen:

```
A::A();
```

Dieser *Standard-Parameterlose-Konstruktor* sorgt dafür, dass für das Objekt ausreichend Speicher reserviert wird — der Speicherinhalt wird nicht weiter behandelt (bei automatischen Objekten ist der Speicherinhalt somit zufällig!).

Der Standard-Parameterlose-Konstruktor wird immer implizit vom System dann aufgerufen, wenn ein neues `A`-Objekt ohne weitere Angaben — also ein *Standard-A-Objekt* erzeugt werden soll:

```
...
A a,b,c;           // 3-mal parameterloser Konstruktor
A A_Feld[100];     // 100-mal parameterloser Konstruktor

A *p = new A;      // 1-mal parameterloser Konstruktor
A *q = new a[20];  // 20-mal parameterloser Konstruktor
...
```

Alternativ ist auch folgende Schreibweise möglich:

```
A a(); // a wird mit parameterlosem Konstruktor erzeugt
```

Mit diesem parameterlosen Konstruktor kann man in gewissen Situationen auch namenlose (temporäre Standard-)Objekte erzeugen, etwa als Funktionsargument:

```
// Funktion mit A-Parameter
void fkt( A );
...
// Aufruf dieser Funktion mit Standard-A-Objekt als Argument:
fkt( A() );
...
```

oder als Fehlerobjekt:

```
...
try {
    ...
    if ( murks )
        throw A(); // Standard-A-Objekt auswerfen
    ...
}
...
```

Diese Schreibweise ist auch für Standardtypen möglich:

```
// Funktion mit int-Parameter
void fkt( int );
...
// Aufruf dieser Funktion mit Standard-int-Objekt als Argument:
fkt( int() );
...
try { ...
    if ( murks )
        throw int();    // Standard-int-Objekt auswerfen
    ...
}
```

4.3.2 Standard-Copy-Konstruktor

Dieser hat den Typen:

```
A::A( const A &);
```

und sorgt dafür, dass ein neues Objekt der Klasse **A** als Kopie eines vorhandenen Objektes der gleichen Klasse erzeugt werden kann. Dieser Konstruktor sorgt dafür, dass

- zunächst ausreichend Speicherplatz für das neue Objekt zur Verfügung gestellt wird (wie beim parameterlosen Konstruktor)
- darüberhinaus erhält jede Datenkomponente des neuen Objektes den gleichen Wert, wie die entsprechende Komponente des vorhandenen Objektes (bei Standardtypen als Komponenten wird byteweise kopiert, bei Objekten als Komponenten wird für diese wiederum der zum Komponententyp gehörende Copy-Konstruktor aufgerufen).

Dieser Copy-Konstruktor wird etwa bei expliziter Initialisierung mit einem Wert der gleichen Klasse:

```
A a;        // parameterloser Konstruktor
...         // mach irgendwas mit a

A b = a;    // b wird als Kopie von a erzeugt
A c(a);     // c wird als Kopie von a erzeugt
...
```

aufgerufen, aber auch bei Wertübergabe an eine Funktion:

```
...
// Funktion mit Parameter vom Typ A
void fkt( A a_param )
{ ... }
```



```
...
int main(void)
{ A a;
  ...
  fkt(a); // Aufruf dieser Funktion, mit Argument a
  ...
}
```

(beim Aufruf der Funktion wird der Parameter `a_param` mit dem Copy-Konstruktor als Kopie des Funktionsargumentes — hier also dem A-Objekt `a` — erzeugt!)
und auch bei Funktionsergebnissen:

```
// Funktion mit Ergebnis vom Typ A
A fkt(void)
{
  A tmp;
  ...
  return tmp;
}
```

(das Funktionsergebnis, welches im aufrufenden Programmteil ggf. weiter verwendet wird, wird als Kopie von `tmp` erstellt!)

Der Copy-Konstruktor kann auch bei `new` und `new[]` verwendet werden:

```
A a; // A-Objekt
...
A *p = new A(a); // dynamisch erzeugtes A-Objekt wird mit a
                  // initialisiert
...
A *q = new A[100](a); // jedes Feldelement wird mit a initialisiert
```

4.3.3 Selbstgeschriebene Konstruktoren

Reicht die Funktionalität der Standard-Konstruktoren nicht aus, so kann der Entwickler einer Klasse eigene Konstruktoren deklarieren und definieren.

Jeder Konstruktor stellt automatisch den für die Datenkomponenten des Objektes notwendigen Speicherplatz zur Verfügung — der Konstruktor braucht sich dann nur noch um den Inhalt dieser Datenkomponenten zu kümmern!

Konstruktoren können Parameter haben, etwa wie in folgendem Beispiel einer Klasse zur Darstellung rationaler Zahlen als Brüche:

```
class Bruch {

private:
  int zaehler;
  int nenner;
```

```

public:
    // Konstruktor mit zwei int-Parametern
    Bruch(int z, int n)
    { zaehler = z;
      nenner  = n;
    }
    ...
};
...
Bruch a(7,10);    // sieben Zehntel
Bruch b(1,3);     // ein Drittel
Bruch c(4,2);     // vier Zweite
Bruch d(1,0);     // PROBLEM, kein Fehler
...

```

Wichtig ist:

Hat eine Klasse einen selbstdefinierten, parameterbehafteten Konstruktor, so gibt es nicht mehr den ansonsten vom System automatisch bereitgestellten parameterlosen Konstruktor!

Stellt der Klassenentwickler also selbst keinen parameterlosen Konstruktor zusätzlich zur Verfügung, können keine “Standard-Objekte“ mehr erzeugt werden!

In obigem Bruch-Beispiel:

```

...
Bruch a;           // FEHLER: kein parameterloser Konstruktor vorhanden
Bruch feld[10];    // FEHLER: kein parameterloser Konstruktor vorhanden
...

```

Möchte der Klassen-Entwickler, dass Standard-Objekte (ohne explizites Argument) seiner Klasse erzeugt werden können, so muss er dann selber dafür sorgen, dass ein ohne Argumente aufrufbarer Konstruktor definiert wird.

Möglichkeiten hierzu:

- Er deklariert und definiert den parameterlosen Konstruktor, ggf. mit leerem Anweisungsteil:

```

class Bruch {
private:
    ...
public:
    ...
    Bruch()    // parameterloser Konstruktor
    { }        // leerer Anweisungsteil ist OK!
    ...
};

```

- oder er versieht einen parameterbehafteten Konstruktor so mit Defaultwerten, dass er auch ohne Argumente aufrufbar ist:

```

class Bruch {

    private:
        int zaehler;
        int nenner;

    public:
        ...
        Bruch(int z = 0, int n = 1)
        { zaehler = z;
          nenner  = n;
        }
        ...
};

...
Bruch c(3,4);    // Zaehler 3, Nenner 4
Bruch b(7);      // Zaehler 7, Nenner 1
Bruch a;         // Zaehler 0, Nenner 1
...

```

Ein mit einem Argument aufrufbarer Konstruktor kann bei `new` und `new[]` verwendet werden, etwa der Konstruktor `Bruch(int z = 0, int n = 1);` aus dem letzten Beispiel:

```

// Konstruktoraufrufe mit einem Argument:
Bruch *p = new Bruch(7);    // Bruch mit Zaehler 7 und Nenner 1
                               // initialisiert!
Bruch *q = new Bruch[20](7); // jedes Feldelement mit Zaehler 7 und
                               // Nenner 1 initialisiert!

// Konstruktoraufrufe ohne Argument:
Bruch *r = new Bruch;        // Bruch mit Zaehler 0 und Nenner 1
Bruch *s = new Bruch[100];   // 100 Brueche mit Zaehler 0 und Nenner 1

```

4.3.4 Initialisierungslisten

Hat ein Objekt einer Klasse B eine Komponente einer anderen Klasse A, so wird standardmäßig bei jedem Konstruktor für B, bevor der Anweisungsteil des B-Konstruktors durchgeführt wird, zunächst der parameterlose Konstruktor für die A-Komponente aufgerufen! Insbesondere muss ein parameterloser A-Konstruktor verfügbar sein!

```

class A {
    ...
    public:
        A ( int);    // einziger Konstruktor fuer A!
        ...
};

```

```

class B {
private:
    A a_komp1;          // erste A-Komponente
    A a_komp2;          // zweite A-Komponente
    ...
public:
    B( int i )
    // an dieser Stelle wird versucht, fuer die A-Komponenten jeweils
    // den parameterlosen A-Konstruktor aufzurufen.
    // Da es diesen nicht gibt, meldet der Compiler einen FEHLER !
    { ...
    }
    ...
};

```

Abhilfe bietet hier eine sogenannte *Initialisierungsliste*:

Bei der Implementierung des B-Konstruktors kann hinter der Parameterliste, vor dem Anweisungsteil ein Doppelpunkt und anschließend eine Liste von Konstruktoraufrufen für eventuelle Komponenten aufgeführt sein, etwa:

```

    B( int i ) : a_komp1( i ), a_komp2( 2*i+5 )
    { ... }

```

Hier wird für die A-Komponente `a_komp1` der Konstruktor `A(int)` mit dem Argument `i` (vom B-Konstruktor stammender Parameter) aufgerufen und für die zweite A-Komponente `a_komp2` derselbe Konstruktor, aber mit Argument `2*i+5`. (Der parameterlose Konstruktor für A wird also nicht mehr benötigt!)

Aus Konsistenzgründen sind auch für Komponenten von Standardtypen Einträge in einer Initialisierungsliste möglich, etwa bei folgendem Konstruktor zur Initialisierung eines Bruches:

```

class Bruch {
private:
    int zaehler;
    int nenner;
public:
    Bruch ( int z = 0, int n = 1 ) : nenner(n), zaehler(z)
    { }
    ...
};

```

Hier wird die Komponente `nenner` mit dem Parameter `n` initialisiert und die Komponente `zaehler` mit dem Parameter `z`. Der eigentliche Anweisungsteil des Konstruktors bleibt hier leer, da mit der Initialisierungsliste bereits alles Erforderliche erledigt ist.

Achtung:

Die Reihenfolge der Einträge in der Initialisierungsliste entspricht nicht unbedingt der Reihenfolge, in der die Initialisierungen tatsächlich durchgeführt werden — vielmehr

richtet sich die Reihenfolge der Initialisierung nach der Reihenfolge der Komponenten im Klassenrumpf, in obigem Beispiel wird zuerst `zaehler` initialisiert und dann erst `nenner`, da `zaehler` im Klassenrumpf vor `nenner` steht! (In diesem Beispiel ist das zwar unerheblich, in anderen Beispielen könnte es aber auf die Reihenfolge ankommen!)

Hat eine Klasse Konstanten oder Referenzen als Daten-Komponenten, so müssen diese über eine Initialisierungsliste jedes Konstruktors der Klasse vorbesetzt werden, da sie im Anweisungsteil des Konstruktors nicht mehr vorbesetzt werden können!

Wird eine Referenz nicht über die Initialisierungsliste vorbesetzt, liefert der Compiler eine Fehlermeldung.

Wird eine Konstante nicht über die Initialisierungsliste vorbesetzt, so erhält sie den Standardwert ihres Types, der später nicht mehr abgeändert werden kann!

4.3.5 Ausnahmen in Konstruktoren

Da Konstruktoren keine Rückgabe haben, können sie Fehler nicht durch Funktionsergebnisse “zurückmelden”.

Natürlich kann man die C++-Techniken der Ausnahmebehandlung verwenden und innerhalb von Konstruktoren Fehlerobjekte auswerfen und dann Objekte, bei deren Erzeugung möglicherweise Fehler ausgeworfen werden, innerhalb eines `try`-Blockes verwenden und anschließend (mittels `catch`) die möglichen Fehlerfälle abfangen:

```
class A {
    ...
public:
    struct A_Konstruktor_Fehler {}; // leere Klasse
    A ()      throw(A_Konstruktor_Fehler);
    A (int) throw(A_Konstruktor_Fehler);
    ...
};
...
A::A() throw(A_Konstruktor_Fehler)
{ ...
    if ( ... ) throw A_Konstruktor_Fehler();
    ...
}
...
A::A(int i) throw(A_Konstruktor_Fehler)
{ ...
    if ( ... ) throw A_Konstruktor_Fehler();
    ...
}
...
void fkt( int i)
{ try { A *p = new A; // hierbei wird moeglicherweise
                      // ein A_Konstruktor_Fehler ausgeworfen!
    ...
}
```

```

    }
    catch( A::A_Konstruktor_Fehler fehler)
    { ... }
    ...
}

```

Wird diese Klasse A nun von einer anderen Klasse B verwendet (oder ist die Klasse B von der Klasse A abgeleitet), so wird bei der Erzeugung eines B-Objektes ein A-Konstruktor aufgerufen (implizit der A-Standardkonstruktor oder mittels Initialisierungsliste ein anderer).

Ein B-Konstruktor kann hierbei die möglicherweise von den A-Konstrukturen ausgelösten Fehler selbst abfangen:

```

class B {    // neue Klasse

    A a;      // verwendet Klasse A
    ...
public:
    B();      // Standardkonstruktor
    B(int);   // weiterer Konstruktor
    ...
};

B::B()       // Definition des Standardkonstruktors
try          // gesamte Definition des Konstruktors, implizit auch die
            // Aufrufe der Komponentenkonstruktoren in try-Block
{
    ...      // Anweisungsteil des B-Standardkonstruktors
}
catch ( A::A_Konstruktor_Fehler fehler)
{
    ...      // Ausnahme abfangen
}

B::B(int i)  // Definition des int-Konstruktors
try : a(2*i) // Initialisierungsliste und Anweisungsteil des
{           // Konstruktors in try-Block
    ...
}
catch ( A::A_Konstruktor_Fehler fehler)
{
    ...      // Ausnahme abfangen
}

```

(Dieses Beispiel funktioniert mit unseren Compilern so wie hier aufgeschrieben, nicht jedoch, wenn die B-Konstrukturen innerhalb des Klassenrumpfes von B definiert sind!)

4.3.6 Copy-Konstruktor und dynamische Komponenten

Am Beispiel des Kellerspeichers als Lineare Liste hatten wir eingesehen, dass eine Klasse mit dynamischen Komponenten einen vernünftigen Destruktor benötigt, damit nach Verwendung solcher Objekte im Speicher kein "Müll" übrig bleibt.

Ist ein solcher Destruktor definiert, so führt der Copy-Konstruktor zu neuen Problemen, wie folgende Anwendung zeigt:

Headerdatei der Klasse `Stack`:

```
#ifndef _Stack_h
#define _Stack_h

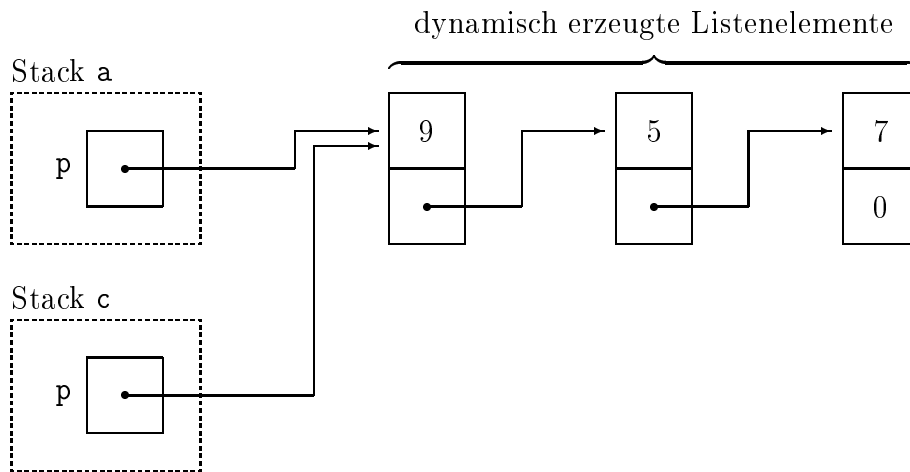
class Stack { // neuer Datentyp hat Namen: Stack
protected: // Impl.-Details, nicht oeffentlich
    struct listel { // eingebetteter Typ:
        int eintrag; // Listenelement
        listel *next;
    } *p; // Zeiger auf Listenanfang

public: // oeffentliche Schnittstelle
    Stack(); // Konstruktor, initialisiert leere Liste
    void push(int); // Funktion zum Einkellern
    int pop(void); // Funktion zum Auskellern
    ~Stack(); // Destruktor, Freigabe der Liste
};
#endif
```

Anwendung:

```
#include "Stack.h"
...
void fkt(void)
{ Stack a;
  a.push(7);
  a.push(5);
  a.push(9);
  Stack c(a); // c als Kopie von a erzeugt
  ...
  return;
}
```

Beim Aufruf der Funktion wird der lokale `Stack a` erzeugt, hierbei über den implizit aufgerufenen Konstruktor initialisiert (ist also zunächst leer) und anschließend werden 3 Elemente "eingekellert". Anschließend wird eine neuer `Stack c` als Kopie von `a` erzeugt, wobei die Komponente `p` von `c` den gleichen Wert erhält, wie die Komponente `p` von `a`, d.h. die Komponente `c.p` zeigt auch auf den Anfang der zu `a` erzeugten Linearen Liste:



Beim Ende der Funktion führt das zu dem Problem, dass sowohl für *a* als auch für *c* der Destruktor aufgerufen und somit die tatsächliche Liste zweimal freigegeben wird! Das zweimalige Freigeben von dynamisch reservierten Speicher führt zu einem Laufzeitfehler, dürfte also bestenfalls einen Programmabsturz verursachen.

Copy-Konstruktor “verbieten“

Abhilfe bietet hier, den Copy-Konstruktor einfach zu verbieten, indem man ihn im *private*-Zugriffsabschnitt deklariert und nicht (oder mit leerem Anweisungsteil) definiert:

```
#ifndef _Stack_h
#define _Stack_h

class Stack { // neuer Datentyp hat Namen: Stack
protected: // Impl.-Details, nicht oeffentlich
    struct listel { // eingebetteter Typ:
        int eintrag; // Listenelement
        listel *next;
    } *p; // Zeiger auf Listenanfang

    // Copy--Konstruktor privat deklarieren und
    // somit fuer den Anwender verbieten:
    Stack ( const Stack &);

public: // oeffentliche Schnittstelle
    Stack(); // Konstruktor, initialisiert leere Liste
    void push(int); // Funktion zum Einkellern
    int pop(void); // Funktion zum Auskellern
    ~Stack(); // Destruktor, Freigabe der Liste
};
#endif
```

in diesem Fall würde der Compiler an der Stelle:


```
...
    Stack a;
    ...
    Stack c(a); // FEHLER: c als Kopie von a erzeugen geht nicht mehr!
...
```

die Fehlermeldung ausgeben, dass der Copy-Konstruktor nicht verfügbar ist. (Compilerfehlermeldungen sind besser als Laufzeitfehler!)

Allerdings kann man jetzt keine Objekte vom Typ `Stack` mehr per Wert an eine Funktion übergeben, da auch hier der Copy-Konstruktor aufgerufen würde:

```
//Funktion mit Stack-Parameter
void fkt(Stack a_param);

...
Stack b;
...
fkt(b);    // FEHLER: Copy-Konstruktor nicht verfuegbar!
...
```

Copy-Konstruktor neu definieren

Alternativ kann man den Copy-Konstruktor neu definieren und zwar so, dass er die dynamischen Komponenten vernünftig behandelt, etwa:

```
#ifndef _Stack_h
#define _Stack_h

class Stack { // neuer Datentyp hat Namen: Stack
protected: // Impl.-Details, nicht oeffentlich
    struct listel { // eingebetteter Typ:
        int eintrag; // Listenelement
        listel *next;
    } *p; // Zeiger auf Listenanfang

public: // oeffentliche Schnittstelle
    Stack(); // Konstruktor, initialisiert leere Liste
    void push(int); // Funktion zum Einkellern
    int pop(void); // Funktion zum Auskellern
    ~Stack(); // Destruktor, Freigabe der Liste

    // Copy-Konstruktor deklarieren
    Stack(const Stack &);
};
#endif
```

Dieser Copy-Konstruktor könnte wie folgt definiert werden:

```

Stack::Stack( const Stack &alt)
{
    p = 0;    // neue Liste zunaechst leer

    // zwei Hilfszeiger
    listel *tmp_alt;
    listel *tmp_neu;

    if ( (tmp_alt = alt.p) != 0) // falls alte Liste nicht leer
    { // erstes Element kopieren
        p = new listel;
        p->inhalt = tmp_alt->inhalt;
        p->next   = 0;

        tmp_neu = p;
        tmp_alt = tmp_alt->next;

        // ggf. alle weiteren Elemente kopieren
        while ( tmp_alt != 0 )
        { tmp_neu->next = new listel;
          tmp_neu->next->inhalt = tmp_alt->inhalt;
          tmp_neu->next->next = 0;

          tmp_alt = tmp_alt->next;
          tmp_neu = tmp_neu->next;
        }
    }
}

```

(Hiermit sind noch nicht alle Probleme für Klassen mit dynamischen Komponenten beseitigt, der Zuweisungsoperator = macht auch noch Schwierigkeiten! Dieser wird im Abschnitt 5.2 noch behandelt.)

4.3.7 Konstruktoren und Typumwandlung

Durch einen Konstruktor für eine Klasse **A** mit einem Parameter vom Typ **T** (Standardtyp oder auch Klassentyp) ist eine Typumwandlung von **T** nach **A** definiert, die ggf. vom System auch implizit angewendet wird.

Als Beispiel soll nochmals die Bruch-Klasse dienen mit dem mit Default-Parametern versehenen Konstruktor

```
Bruch( int z = 0, int n = 1);
```

Dieser kann dazu verwendet werden, anhand eines `int`'s einen Bruch zu erzeugen, etwa:

```
Bruch b(7);
Bruch c = 5;
```

Sollte eine Funktion mit einem Parameter vom Typ `Bruch` deklariert sein, etwa:

```
void fkt(Bruch);
```

und keine gleichnamige Funktion mit einem `int`-Parameter, so könnte diese Funktion trotzdem mit einem `int`-Argument aufgerufen werden:

```
fkt(7);
```

Hier wird aus dem `int`-Wert anhand des `Bruch`-Konstruktors mit dem Argument 7 ein (temporärer) `Bruch` erzeugt und die Funktion mit diesem temporären `Bruch` als Argument aufgerufen.

Dies kann sinnvoll sein, bei einigen Anwendungen ist ein solcher zur Typumwandlung in Ausdrücken verwendeter impliziter Konstruktoraufwurf unerwünscht!

Um dies zu verhindern, kann der entsprechende Konstruktor im Klassenrumpf als `explicit` deklariert werden:

```
class Bruch {
private:
    ...
public:
    // explicit-Deklaration des Konstruktors
    explicit Bruch( int z, int n);
    ...
};
```

Ein solcher, als `explicit` deklariert Konstruktor wird vom System nicht mehr zur impliziten Typumwandlung verwendet!

Explizite Typumwandlungen sind dann immer noch möglich:

```
void fkt(Bruch);
...
fkt( 7 );    // Compiler-FEHLER-Meldung: keine implizite Typumwandlung!
...
fkt( Bruch(7) );           // OK, expliziter Konstruktoraufwurf
fkt( (Bruch) 7 );          // OK, explizite Typumwandlung
fkt( static_cast<Bruch> (7) ); // OK, explizite Typumwandlung
...
```

4.3.8 `new` bzw. `new[]` und parameterbehaftete Konstruktoren

Erzeugt man in folgender Form dynamisch Objekte einer Klasse mittels `new` oder `new[]`:

```
class A {
    ...
public:
    A();           // parameterloser Konstruktor
    A(int);        // Konstruktor mit int Parameter
    A(double);     // Konstruktor mit double Parameter
    A(int, int);   // Konstruktor mit zwei int Parametern
};
```

```

    ...
};

A *p = new A;          // 1-mal Konstruktor A()
A *q = new A[100];     // 100-mal Konstruktor A()
...

```

so werden die erzeugten Objekte mittels des zugehörigen parameterlosen Konstruktors initialisiert. Ist kein parameterloser Konstruktor verfügbar, ist so die dynamische Definition der Objekte nicht möglich.

Auch bei `new` bzw. `new[]` kann man die erzeugten Objekte explizit mit einem anderen Konstruktor erzeugen lassen. Hierzu muss man bei `new` hinter dem Typnamen und bei `new[]` hinter der dem Typnamen folgenden Dimensionierung in runden Klammern für den gewünschten Konstruktor passende Argumente angeben. Anhand Anzahl und Typ der Argumente wird der entsprechende Konstruktor ausgewählt:

```

int i,j;
double x;
...
A *p1 = new A;                // 1-mal parameterloser Konstruktor A()
A *p2 = new A(i);             // 1-mal Konstruktor A(int)
A *p3 = new A(x);             // 1-mal Konstruktor A(double)
A *p4 = new A(i,j);           // 1-mal Konstruktor A(int,int)

A *q1 = new A[100];           // 100-mal parameterloser Konstruktor A()
A *q2 = new A[100](i);        // 100-mal Konstruktor A(int)
A *q3 = new A[100](x);        // 100-mal Konstruktor A(double)
A *q4 = new A[100](i,j);      // 100-mal Konstruktor A(int,int)
...

```

Bei `new[]` werden alle Feldelemente mit dem gleichen Konstruktor mit gleichen Argumenten erzeugt!

4.3.9 Adressen von Konstruktoren

Wie wir gesehen haben, sind Konstruktoren spezielle Funktionen, die auf spezielle Art und Weise mit Speicherverwaltung zu tun haben: Bei Konstruktoraufrufen wird vom System zunächst mal Speicher für die Komponenten des Objektes bereitgestellt und dieser Speicherbereich wird durch den Anweisungsteil des Konstruktors bearbeitet. Diese enge Zusammenarbeit mit Speicherverwaltung unterscheidet Konstruktoren von “normalen” Funktionen — aus diesem Grund kann man keine Adressen von Konstruktoren erhalten und somit sind Zeiger auf Konstruktoren nicht möglich!

4.4 Destruktoren im Detail

Jede Klasse `A` besitzt standardmäßig einen Destruktor `~A()`, der immer dann aufgerufen wird, wenn ein `A`-Objekt ans Ende seiner Lebenszeit angelangt ist:

1. bei Objekten der Speicherklasse **auto** am Ende des Anweisungsblockes, in dem sie definiert sind,
2. bei Objekten als Funktionsparametern das Ende der Funktion,
3. bei Objekten der Speicherklasse **extern** oder **static** das Programmende,
4. bei temporär angelegten Objekten — etwa als Zwischenergebnis eines komplexeren Ausdrucks (siehe Operatorueberladung, Abschnitt 5.2) oder als temporäres Funktionsergebnis — dann, wenn das temporäre Objekt nicht mehr benötigt wird,
5. bei mittels **new** oder **new[]** angelegten Objekten beim zugehörigen Aufruf von **delete** bzw. **delete[]** (nicht bei Freigabe mit **free** bei mittels **malloc** o.ä. dynamisch angelegten Objekten!),
6. bei temporären Objekten, bei Objekten der Speicherklasse **auto** oder bei Objekten als Funktionsparametern kann das Ende ihrer Lebenszeit auch aufgrund einer ausgeworfenen Ausnahme erreicht werden:

```
class A { ... }

void fkt1(A a)
{ A b;
  ...
  if ( ... ) throw ausnahme();
  ...
}

int fkt2(void)
{
  try
  { A c;
    ...
    fkt1(c);
    ...
  }
  catch( ausnahme fehler)
  { ...
  }
}
```

Innerhalb von **fkt2** wird die Funktion **fkt1** aufgerufen. Sollte während dieser Ausführung von **fkt1** die Ausnahme ausgeworfen werden, werden alle innerhalb des **try**-Blockes erzeugten Variablen, insbesondere das zu **fkt2** lokale **A**-Objekt **c**, der bei Aufruf von **fkt1** erzeugte Funktionsparameter **a** und das zu **fkt1** lokale Objekt **b**, zerstört. Hier sind also ebenfalls implizit Destruktoren beteiligt.

Ein Destruktor kann auch explizit aufgerufen werden:

```

class A {
    ...
};
...
void fkt(void)
{ A a;
    ...
    a.~A();    // expliziter Konstruktoraufruf
    ...
}
...

```

(Auf unseren Systemen wird der Destruktor dann allerdings am Ende der Funktion nochmals aufgerufen!)

Der implizit vorhandene (Standard-)Destruktor gibt den Speicherbereich für die Komponenten des Objektes frei.

Reicht (etwa bei Klassen mit dynamischen Komponenten, vgl. etwa die Listenversion des Kellerspeichers, siehe Abschnitt 4.2.6) die Funktionalität dieses (Standard-)Destruktors nicht aus, so kann man selbst einen Destruktor `~A()`; definieren!

Da der Destruktor (implizit) von der Anwendung benötigt wird, muss ein selbstdefinierter Destruktor im öffentlichen Zugriffsabschnitt der Klasse (**public**) sein!

Der Name eines Destruktors besteht aus dem Tilde-Zeichen `~` und angehängtem Klassennamen, der Destruktor hat keine Parameter (kann somit auch nicht mit unterschiedlicher Signatur überladen werden) und kein Ergebnis (auch nicht `void`)!

Hat ein Objekt einer Klasse **B** eine Komponente vom Typ der Klasse **A** (oder ist die Klasse **B** von der Klasse **A** abgeleitet), so wird bei der Zerstörung eines **B**-Objektes zunächst der Anweisungsteil des **B**-Destruktors ausgeführt, anschließend der **A**-Destruktor für die **A**-Komponente (oder den **A**-Teil) aufgerufen und schließlich der Speicherbereich für das ganze **B**-Objekt freigegeben (für alle Komponenten, einschließlich der **A**-Komponente).

Da ein Destruktor keine Parameter/Argumente haben kann, ist ein Überladen des Destruktors mit unterschiedlichen Signaturen nicht möglich. Es kann somit nur parameterlose Destruktoren geben und entsprechend ist eine den Initialisierungslisten bei Konstruktoren analoge Technik für Destruktoren nicht erforderlich und auch nicht vorhanden!

Die enge Verzahnung von Destruktor und systeminterner Speicherverwaltung hat — wie bei Konstruktoren — die Konsequenz, dass man von Destruktoren keine Adresse erhalten kann und es somit keine Zeiger auf Destruktoren gibt!

Entwirft man eine Klasse, welche zur Ableitung geeignet sein soll, so sollte man aus Gründen, welche erst in Kapitel 7 ersichtlich sind, unbedingt einen eigenen Destruktor als **virtuell** selbst definieren — selbst wenn der Destruktor noch keine eigene Funktionalität aufweist (leerer Anweisungsteil).

Destruktoren und Ausnahmen

Natürlich können Destruktoren ggf. Ausnahmen auswerfen, entweder direkt:

```

class ausnahme {};    // Ausnahmetyp
...
A::~~A() throw(ausnahme)
{ ...
    if (... ) throw ausnahme();
    ...
}

```

oder indirekt, indem der Destruktor eine Funktion aufruft, welche möglicherweise eine Ausnahme auswirft:

```

class ausnahme {};    // Ausnahmetyp
...
void f(void) throw(ausnahme); // Funktion, welche moeglicherweise
                               // eine Ausnahme auswirft
A::~~A() throw(ausnahme)
{ ...
    f(); // f erzeugt ggf. eine Aushnahme
    ...
}

```

Bei der normalen Zerstörung von Objektes ist dies durchaus sinnvoll und erwünscht. Doch wie oben erwähnt, werden bei ausgeworfenen Ausnahmen für innerhalb des zugehörigen `try`-Blockes definierte Objekte Destrukturen aufgerufen und diese bei der Ausnahmeabwicklung aufgerufenen Destrukturen sollten ihrerseits keine weiteren Ausnahmen auswerfen. Denn eine beim Ausnahmebehandlungsmechanismus ausgeworfene Ausnahme wird als Fehler behandelt, der zum sofortigen Programmabbruch führt (es wird die Funktion `terminate()` aufgerufen, welche das Programm abbricht)! Ähnlich wie bei Konstruktoren kann der Anweisungsteil eines Destruktors ganz als `try`-Block verwendet werden, an den sich dann eine Fehlerbehandlung anschließt:

```

class ausnahme {};    // Ausnahmetyp
...
void f(void) throw(ausnahme); // Funktion, welche moeglicherweise
                               // eine Ausnahm auswirft
A::~~A() throw(ausnahme)
try
{ // Anweisungsteil des Destruktors
    ...
    f(); // f erzeugt ggf. Ausnahme
    ...
    // Destruktor wirft ggf. selbst Ausnahme
    if ( ... ) throw ausnahme();
    ...
}
catch( ausnahme fehler)
{ ... // Ausnahme behandeln
}
...

```

Dieser Destruktor wirft entsprechend keine Ausnahme (jedenfalls keine von diesem Typ `ausnahme`) aus!

Alternativ kann man mit der in der Headerdatei `<exception>` deklarierten Funktion:

```
bool uncaught_exception();
```

in Erfahrung bringen, ob eine Ausnahme ausgeworfen wurde und noch ansteht (noch nicht abgefangen ist), in diesem Fall ist das Funktionsergebnis `true`, ansonsten `false`. Man könnte also einen Destruktor wie folgt definieren:

```
#include <exception>
...
A::~~A()
{ if ( uncaught_exception() )
  { // Destruktor wurde innerhalb der Abwicklung
    // einer Ausnahme ausgeworfen -> keine neuen
    // Ausnahmen auswerfen!
    ...
  }
  else
  { // Destruktor wurde "normal" aufgerufen
    // hier stoert der Auswurf einer Ausnahme nicht!
    ...
    if ( ... ) throw ausnahme();
    ...
  }
}
```

4.5 Ressourcenmanagement über Konstruktoren und Destrukturen

Wenn man eine Ressource vom Betriebssystem anfordert, muss man sie in der Regel auch wieder freigeben, etwa:

- Speicher dynamisch reservieren, nach Gebrauch wieder freigeben,
- eine Datei öffnen und nach deren Verwendung wieder schließen,
- eine Netzwerkverbindung aufbauen und nach Durchführung der Kommunikation wieder schließen,
-

Das Freigeben der nicht mehr benötigten Ressource darf nicht vergessen werden, da ansonsten das System schnell überlastet wird.

Um dem Benutzer die Verwendung einer solchen Ressource zu vereinfachen, kann man einen *Ressourcenverwaltungstyp* definieren. Im (sinnvoll zu definierenden) Konstruktor

dieses Types wird dann die entsprechende Ressource angefordert und im (ebenfalls sinnvoll zu definierenden) Destruktor wird sie wieder freigegeben — möglicherweise kann man über diesen Ressourcenverwaltungstyp mittels Operatorüberladung (siehe Abschnitt 5.2) einen einfachen Zugriff auf die Ressource ermöglichen.

Wenn der Benutzer die Ressource benötigt, definiert er ein entsprechendes Objekt dieses Types und der Konstruktor dieses Types sorgt dann dafür, dass die entsprechende Ressource angefordert wird.

Der Benutzer kann dann über dieses Objekt auf die reservierte Ressource zugreifen und der Destruktor dieses Types, der ja beim Ende der Lebenszeit des Ressourcenverwaltungsobjektes automatisch aufgerufen wird, sorgt (ohne Zutun des Benutzers) dafür, dass die Ressource ordnungsgemäß wieder freigegeben wird.

Bei dieser Vorgehensweise ist der Benutzer von der lästigen Pflicht befreit, selbst für die Freigabe der Ressource sorgen zu müssen!

4.6 Statische Klassenkomponenten

Member-Daten und -Funktionen einer Klasse können als `static` vereinbart werden.

4.6.1 Statische Member-Daten

Eine in einem Klassenrumpf als `static` vereinbarte Daten-Komponente wird beim Programmstart erzeugt (Speicher bereitgestellt) und alle (später erzeugten) Objekte dieser Klasse teilen diese Komponente, d.h. die Komponente ist — unabhängig von der Anzahl der Objekte der Klasse — genau einmal vorhanden und wird von allen Objekten gemeinsam benutzt. (Bei gewöhnlichen Datenkomponenten hat jedes Objekt seine eigene entsprechende Komponente — bei einer statischen Datenkomponente nicht!)

Eine im Klassenrumpf deklarierte Komponente muss in einer Implementierungsdatei separat noch definiert werden:

```
// Klassenrumpf, etwa in Headerdatei:
class A {
    private:
        // gewoehnliche Member-Daten
        int kompl;
        ...
    public:
        // statisches Member-Datum, hier ausnahmsweise public
        static int st_kom;

        // Member-Funktionen
        A (int);
        int f(void);
        ...
};
...
```

```
// Implementierung, etwa in *.cc-Datei:

// Definition des Konstruktors
A::A(int i)
{ ... }

// Definition weiterer Member-Funktionen
int A::f(void)
{ ... }

// Definition der statischen Klassenkomponente,
// ggf. mit Initialisierung:
int A::st_komp = 12;
...
```

Diese Komponente `int st_komp`; ist nur einmal vorhanden und auf diese eine Komponente können alle A-Objekte “zugreifen”:

```
int main(void)
{
    A a,b;

    a.st_komp = 5;    // Zugriff ueber Objekt a
    b.st_komp = 7;    // Zugriff ueber Objekt b
    cout << a.st_komp << endl; // Zugriff ueber Objekt a,
                             // Ausgabe des Wertes 7!!
    ...
}
```

Eine statische Daten-Komponente wird von allen Objekten gemeinsam genutzt — sie ist sogar vorhanden, wenn gar kein einziges Objekt der Klasse definiert ist!

Ist diese Komponente (wie hier im Beispiel) im öffentlichen Zugriffsabschnitt, kann in einer Anwendung über explizite Qualifikation über den Klassennamen auf sie zugegriffen werden:

```
int main(void)
{
    // Zugriff ueber explizite Qualifikation, obwohl gar kein
    // einziges A-Objekt vorhanden ist:
    A::st_komp = 25;
    ...
}
```

Da eine solche Komponente eher zur ganzen Klasse als zu einem Objekt “gehört”, heißen solche statischen Member-Daten auch *Klassenvariablen*.

Da eine statische Komponente nur eimal vorhanden ist und somit bei Rekursion keine Schwierigkeiten macht, kann eine Klasse eine statische Komponente vom eigenen Typ besitzen:

```

class Datum {
private:
    // normale Komponenten
    int t, m, j;
    // statische Komponente vom eigenen Typ
    static Datum standardDatum;
    ...
};

```

4.6.2 Statische Member-Funktionen

Auch Member-Funktionen können (im Klassenrumpf) als **static** deklariert werden! Diese dürfen nur auf statische Member-Daten zugreifen und, falls in ihrer Definition eine andere Member-Funktion aufgerufen wird, muss diese andere Member-Funktion ebenfalls **static** sein!

Wie bei statischen Member-Daten kann auf eine statische Member-Funktion mittels expliziter Qualifikation über den Klassennamen zugegriffen (d.h. aufgerufen) werden:

```

// Klassenrumpf:
class A {
private:
    static int st_komp;
    ...
public:
    static void setze_st_komp(int);
    ...
};
...
// Implementierung:

int A::st_komp;                // statisches Member-Datum

void A::setze_st_komp(int i)   // statische Member-Funktion
{ st_komp = i;
  ...
}

int main(void)                 // Anwendung
{ A a;

    // Aufruf der statischen Member-Funktion ueber explizite Qualifikation
    A::setze_st_komp(5);

    // Aufruf derselben Funktion ueber das Objekt a
    a.setze_st_komp(7);
    ...
}

```

4.7 Konstanten oder Referenzen als Komponenten

Konstanten und Referenzen sind als Komponenten einer Klasse ebenfalls möglich:

```
// irgendeine Klasse
class B { ... };

// weitere Klasse
class A {
private:
    const int i_komp;
    B &b_komp;
    ...
public:
    A( int, B&);
    ...
};
```

Zu beachten ist, dass solche Konstanten oder Referenzen bei ihrer Erzeugung — also bei der Erzeugung eines Objektes durch einen Konstruktor — gleich initialisiert werden!

Dies kann nur über eine Initialisierungsliste geschehen (nicht im Anweisungsteil des Konstruktors, da dort nur zugewiesen werden könnte):

```
// Implementierung des Konstruktors:
A::A( int i, B &b) : i_komp( i + 3), b_komp(b) { ... }
```

Bei einer solchen Konstante (bzw. Referenz) hat jedes Objekt seine eigene Konstante, bei unterschiedlichen Objekten kann der Wert der Konstanten somit unterschiedlich sein!

4.7.1 Klassenkonstanten

Man kann natürlich eine solche Konstante auch **static** vereinbaren. In diesem Fall verfügen alle Objekte der Klasse über die gleiche Konstante (der Wert dieser Konstanten ist nur einmal abgespeichert, alle Objekte greifen auf diese Konstante zu, mittels expliziter Qualifikation über den Klassennamen kann man ggf. auch auf diese Konstante zugreifen!).

Eine solche statische Konstante muss natürlich separat definiert und dabei initialisiert werden:

```
// Klassenrumpf:
class A {
private:
    // statische Konstante:
    static const double PI;
    ...
};
```

```
...
// Implementierung:
const double A::PI = 3.1415926;
...
```

Seit dem neuen Standard kann man eine integrale Klassenkonstante gleich bei ihrer Deklaration im Klassenrumpf “initialisieren“, um deren Wert im Klassenrumpf (etwa für die Länge eines Feldes) weiter zu verwenden. Auch eine solche Klassenkonstante muss dennoch, dann aber ohne Initialisierung, “implementiert“ werden:

```
class A {
private:
    // Deklaration und Initialisierung
    // der Klassenkonstanten
    static const int FELDLAENGE = 100;

    // Verwendung dieser Konstanten
    double d_feld[FELDLAENGE];

    ...
};
...
// Implementierung:
const int A::FELDLAENGE;
...
```

Diese Konstante `FELDLAENGE` gehört zur Klasse `A`, steht dort in einem Zugriffsabschnitt und ist keine Präprozessorkonstante!

Auf älteren Compilern, bei denen keine integralen Klassenkonstanten im Klassenrumpf initialisiert werden können, kann man sich mit einem eingebauten `enum`-Typen behelfen:

```
class A {
private:
    // eingebauter namenloser enum-Typ:
    enum { FELDLAENGE = 100 };

    // Verwendung dieses Namens
    double d_feld[FELDLAENGE];

    ...
};
```

4.8 Komponentenzeiger

Komponentenzeiger zu einer Klasse sind Zeiger (eines gewissen Types), welche nur auf Komponenten (vom entsprechenden Typ) dieser Klasse zeigen dürfen!

4.8.1 Zeiger auf Datenkomponenten

Ein Zeiger `p` auf Datenkomponenten eines gewissen Typs `T` einer Klasse `A` wird wie folgt definiert:

```
T A::* p;
```

zu lesen: `p` ist ein Zeiger (*) auf Komponenten der Klasse `A` (`A::`) vom Typ `T`, d.h. `p` darf nur auf Komponenten vom Typ `T` der Klasse `A` zeigen.

Ist dann `T_komp` eine Komponente der Klasse `A` vom Typ `T`, so kann dem Zeiger `p` die Adresse dieser Komponente zugewiesen werden:

```
p = & A::T_komp;
```

Sprechweise: `p` zeigt auf die Komponente `T_komp` der Klasse `A`!

Ist jetzt `a` ein konkretes Objekt der Klasse `A`, so kann für `a` dieser Komponentenzeiger verwendet werden:

```
a.*p
```

`.*` ist hierbei ein neuer Operator in C++, der nur bei Komponentenzeigern verwendet wird!

Sprechweise: `a.*p` ist die Komponente des Objektes `a`, auf die der Komponentenzeiger `p` zeigt — in unserem Fall ist dies die Komponente `T_komp` — (im Beispiel ist also `a.*p` gleichwertig zu `a.T_komp`).

Sprechweise: `p` zeigt auf die Komponente `T_komp` eines `A`-Objektes, `a.*p` ist diese Komponente `T_komp` des konkreten Objektes `a`.

Für Adressen von Objekten (der Klasse `A`) kann entsprechend der neue Operator `->*` verwendet werden:

```
A a;           // A-Objekt
A *op = &a;     // A_zeiger, zeigt auf a
...
op->*p...       // Komponente T_komp des Objektes, auf welches op zeigt!
...
```

(Sprechweise: die Komponente des Objektes, auf welches `op` zeigt, auf die der Komponentenzeiger `p` zeigt!)

4.8.2 Zeiger auf Funktionskomponenten

Zeiger auf Funktionskomponenten einer Klasse sind Zeiger auf Funktionen (eines gewissen Typs), welche aber nur auf entsprechende (vom Typ passende) Member-Funktionen einer Klasse zeigen dürfen, etwa:

```
int (A::*fp) (void);
```

Sprechweise: `fp` ist ein Zeiger auf eine Funktion mit keinem Argument und `int`-Rückgabe, darf aber nur auf derartige Member-Funktionen der Klasse `A` zeigen, etwa:

```
class A {
    ...
public:
    int f(void);
```

```

    int g(void);
    ind h(double);
    ...
};
...
int (A::*fp) (void); // fp zeigt auf A-Memberfunktionen mit
                    // int Ergebnis und keinem Argument
...
fp = A::f;    // OK, f hat int Ergebnis und kein Argument
...
fp = A::g;    // OK, g hat int Ergebnis und kein Argument
...
fp = A::h;    // FEHLER, g hat int Ergebnis und double-Argument
...

```

Wie gesehen ist die Zuweisung `fp = A::f`; der Adresse der Member-Funktion `f` und den Member-Funktionszeiger `fp` zulässig — der Zeiger `fp` zeigt auf die Funktion `f` der Klasse `A`!

Für ein konkretes Objekt kann nun diese Funktion über den Komponentenzeiger aufgerufen werden:

1. über das Objekt selbst mit dem Operator `.*`:

```

...
A a;
...
int i = (a.*fp) (); // es wird die Member-Funktion von a
...                // aufgerufen, auf die fp zeigt!

```

2. über die Adresse des Objektes mit dem Operator `->*`:

```

...
A a, *op = &a; // op zeigt auf a
...
int i = (op->*fp) (); // es wird die Member-Funktion, auf die fp
// zeigt, fuer das Objekt, auf welches op zeigt, aufgerufen!
...

```

Zeiger auf Member-Funktionen werden hauptsächlich dann verwendet, wenn eine Member-Funktion als Argument an eine andere Funktion übergeben werden muss. Wie bei Funktionszeigern überhaupt, empfiehlt es sich auch bei Member-Funktions-Zeigern, den Zeigertypen mittels `typedef` zu definieren:

```

typedef int (A::*fp_typ) (void);
// fp_typ ist der Typ: Zeiger auf eine Member-Funktion der Klasse A
// mit keinem Argument und int Ergebnis
...

```

```

fp_typ fp;
// fp ist ein entsprechender Zeiger
...
fp = A::f;
// fp zeigt auf die (vom Typ her passende) Funktion f der Klasse A

A a;           // a ist A-Objekt
A *op = &a;    // op zeigt auf A-Objekt
...
(a.*fp) ();    // Aufruf von f fuer a
(op->*fp) ();   // Aufruf von f fuer *op, also a
...

```

4.8.3 Komponentenzeiger und void *

Während einem void *-Zeiger ansonsten jede beliebige Adresse zugewiesen werden kann, darf einem solchen Zeiger nicht der Wert eines Komponentenzeigers zugewiesen werden:

```

class A { ... };
...
int A::*ip;           // Zeiger auf int-Komponente
int (A::*fp) (void);  // Zeiger auf Funktions-Komponente
...
void *p;
...
p = ip;              // FEHLER: ip Komponentenzeiger
p = fp;              // FEHLER: fp Komponentenzeiger
...

```

4.9 Befreundete Klassen und Funktionen

Durch die Zugriffsabschnitte `private`, `protected` und `public` einer Klasse A wird die Zugreifbarkeit auf die Komponenten der Klasse gesteuert:

1. eine Member-Funktion der Klasse A kann auf alle Komponenten eines A-Objektes zugreifen — natürlich nur bei solchen A-Objekte, mit denen die Funktion zu tun hat, also das *aktuelle Objekt*, in der Funktion definierte lokale Objekte der Klasse A, Funktionsparameter vom Typ A (ggf. globale Objekte der Klasse A).
2. andere Funktionen (“normale“ Funktionen oder Member-Funktionen einer anderen Klasse B) können nur auf die `public`-Teile der A-Objekte zugreifen, mit denen sie zu tun haben! Auf die `private` und `protected` Komponenten können diese Funktionen nicht zugreifen.

Dieser Zugriffsschutz sollte vom Entwickler der Klasse A wohl durchdacht sein und ist für den “normalen“ Anwender der Klasse A gedacht!

Häufig ist es so, dass der Entwickler (oder das Entwicklungsteam) der Klasse **A** nicht allein nur diese eine Klasse entwickelt, sondern gleich mehrere, irgendwie zusammenhängende Klassen **A**, **B** und **C** und zugehörige weitere “normale” Funktionen. In Member-Funktionen einer Klasse **B** werden etwa Parameter oder lokale Variablen der Klasse **A** verwendet, in einer “normalen” Funktion **f** tauchen etwa **A**- und **B**-Objekte auf und irgendwie wird durch alle beteiligten Klassen und Funktionen die Funktionalität zur Verfügung gestellt, welche ein Anwender dann in seiner Applikation verwenden kann.

Bei der Implementierung der auch vom Entwickler der Klasse **A** entworfenen Klasse **B** und “normalen” Funktionen kann der standardmäßige Zugriffsschutz, der für den eigentlichen Anwender gelten soll, hinderlich sein!

Aus diesem Grund kann der Entwickler der Klasse **A** innerhalb des Klassenrumpfes von **A** eine “normale” Funktion, eine Member-Funktion der Klasse **B** oder alle Member-Funktionen einer Klasse **C** als “befreundet” deklarieren! Für als “befreundet” deklarierte Funktionen ist der Zugriffsschutz aufgehoben, d.h. die entsprechende Funktion kann auf alle Komponenten eines **A**-Objektes, mit dem sie zu tun hat, zugreifen!

```
class A {
public:
    ...
private:
    ...

    friend int f( A &); // friend-Dekl. einer "normalen" Funktion
                        // mit int-Ergebnis und A-Referenz-Parameter.
                        // f ist insbesondere keine Member-Funktion zu A,
                        // sondern eine, irgendwo anders definierte Funktion!

    friend void B::B_fkt(void); // friend-Dekl. der Member-Funktion
                                // B_fkt der Klasse B ohne Ergebnis und Parameter
    friend C; // Klasse C ist friend-deklariert
};
```

Die normale Funktion **f**, die Member-Funktion **B_fkt** und alle Member-Funktionen der Klasse **C** können auf alle Komponenten der **A**-Objekte, mit denen sie zu tun haben, zugreifen! (Die Klasse **C** und die Funktionen können später definiert sein!)

Die **friend**-Deklarationen unterliegen keinem Zugriffsabschnitt, die hier im **private**-Teil stehenden **friend**-Deklarationen hätten genausogut im **public** oder **protected**-Teil stehen können!

Nochmals zur Verdeutlichung: befreundete Klassen und Funktionen sind keine x-beliebigen Funktionen, sondern man muss diese Klassen und Funktionen als von einem Entwickler stammende Einheit auffassen, mit welcher der Entwickler eine Gesamt-Funktionalität verwirklichen möchte!

Kapitel 5

Operatoren

5.1 Übersicht über die Operatoren

Folgende Tabelle gibt eine Übersicht über alle Operatoren in C++.

Operatoren im gleichen “Bereich“ (gleiche Nummer, nicht durch einen waagerechten Strich getrennt) haben den gleichen Vorrang (Priorität), Operatoren eines weiter oben stehenden Bereiches (kleinere Nummern) haben höhere Priorität als Operatoren aus einem weiter unten stehenden Bereich (größere Nummer). Bei Operatoren im selben Bereich (nicht durch einen waagerechten Strich abgetrennt — gleiche Nummer) entscheidet (wie in C) die Assoziativität über die Auswertungsreihenfolge.

Nr.	Operator	Assoziativität
1	::	von links nach rechts
2	. -> [] () ++ -- typeid() dynamic_cast<>() static_cast<>() reinterpret_cast<>() const_cast<>()	von links nach rechts
3	! ~ ++ -- + - * & (type) sizeof new new[] delete delete []	von rechts nach links
4	.* ->*	von links nach rechts
5	* / %	von links nach rechts
6	+ -	von links nach rechts
7	<< >>	von links nach rechts
8	< <= > >=	von links nach rechts
9	== !=	von links nach rechts
10	&	von links nach rechts
11	^	von links nach rechts
12		von links nach rechts
13	&&	von links nach rechts
14		von links nach rechts
15	?:	von rechts nach links
16	= += -= *= /= %= &= ^= = <<= >>=	von rechts nach links
17	,	von links nach rechts

Die Tabelle ähnelt (abgesehen von den in C++ neuen Operatoren) stark der entspre-

chende Tabelle der C-Operatoren.

Auffallend ist, dass die Inkrement- bzw. Dekrement-Operatoren ++ und -- zweimal — und dann noch mit unterschiedlicher Priorität (und Assoziativität) auftauchen!

Die Operatoren ++ und -- in Bereich 2 stehen für die Postfix-Form der Operatoren, sind also die in der Form `a++` bzw. `a--` verwendeten.

Die in Bereich 3 stehenden ++ und -- sind die entsprechenden Präfix-Operatoren, sind also in der Form `++a` bzw. `--a` zu verwenden.

Die Postfix-Form hat somit höhere Priorität als die Präfix-Form.

Darüberhinaus ist in C++ der Ergebnistyp der Präfix-Form von ++ bzw. -- ein *L-Value*, der Ergebnistyp der Postfix-Form jedoch nicht! (So hat bei einem `int i` der Ausdruck `++i` eine Adresse, die man sich etwa mit `&++i` verschaffen könnte, der Ausdruck `i++` jedoch nicht, so dass `&i++` nicht erlaubt ist! In C war sowohl bei Präfix als auch bei Postfix das Ergebnis kein *L-Value*!)

Die aus C geerbten, alten Operatoren haben die gleiche Bedeutung und Besonderheiten wie in C.

Insbesondere ist bei einem binären Operator `op` im Allgemeinen (Ausnahme logisches UND `&&`, logisches ODER `||` und Komma-Operator `,`) in einem Ausdruck

`a op b`

nicht spezifiziert, welcher der beiden Operanden `a` oder `b` zuerst ausgewertet wird (bei den genannten Ausnahmen ist es jeweils der Linke, bei `&&` und `||` wird — bei Operanden von Standardtypen — der Rechte Operand nur bei Bedarf ausgewertet). Die Auswertungsreihenfolge in einem Ausdruck kann natürlich durch Klammerung (mit runden Klammern) beeinflusst werden.

In folgenden, nach Priorität (entsprechend der Nummer in Tabelle auf Seite 135) gestaffelten Tabellen ist Anwendung und Bedeutung der Operatoren kurz erläutert. Die In C++ neuen Operatoren sind in der dritten Spalte mit einem * gekennzeichnet.

1. Bereichsauflösung/Bereichszuordnung, Assoziativität: von Links nach Rechts

Nr.	Op.		Anwendung	Bedeutung
1	::	*	<i>Klasse::Element</i> <i>Namensbereich::Element</i> <i>::Name</i>	Klassenzuordnung Zuordnung zu Namensbereich Zugriff auf globale Variable

2. Komponentenzugriff, Indizierung, Funktionsaufruf, Postfix-In-/Dekrement, neue Operatoren zu Typen, Assoziativität: von Links nach Rechts

Nr.	Op.		Anwendung	Bedeutung
2	.		<i>Objekt.Komponente</i>	Selektion über Objekt
	->		<i>Objekt-Zeiger->Komponente</i>	Selektion über Zeiger
	[]		<i>Zeiger[Index]</i>	Feldindizierung
	()		<i>Ausdruck(Ausdrucksliste)</i>	Funktionsaufruf
	++	*	<i>Typ(Ausdruck)</i> <i>Ausdruck++</i>	Typumwandlung/ Werterzeugung Postfix-Inkrementierung
	--		<i>Ausdruck--</i>	Postfix-Dekrementierung

Nr.	Op.		Anwendung	Bedeutung
2	typeid()	*	typeid(<i>Typ</i>)	Typinformation
			typeid(<i>Ausdruck</i>)	Typinformation
	dynamic_cast<>()	*	dynamic_cast< <i>Typ</i> >(<i>Ausdruck</i>)	Typumwandlung
	static_cast<>()	*	static_cast< <i>Typ</i> >(<i>Ausdruck</i>)	Typumwandlung
	reinterpret_cast<>()	*	reinterpret_cast< <i>Typ</i> >(<i>Ausdruck</i>)	Typumwandlung
	const_cast<>()	*	const_cast< <i>Typ</i> >(<i>Ausdruck</i>)	Typumwandlung

3. Negation, Komplement, Präfix-In-/Dekrement, Vorzeichen, Adress/Verweis, Cast, sizeof und die neue Speicherverwaltung, Assoziativität: von Rechts nach Links

Nr.	Op.		Anwendung	Bedeutung
3	!		! <i>Ausdruck</i>	logische Negation
	~		~ <i>Ausdruck</i>	Bit-Komplement
	++		++ <i>Ausdruck</i>	Präfix-Inkrementierung
	--		-- <i>Ausdruck</i>	Präfix-Dekrementierung
	+		+ <i>Ausdruck</i>	positives Vorzeichen
	-		- <i>Ausdruck</i>	negatives Vorzeichen
	*		* <i>Ausdruck</i>	Verweisoperator
	&		& <i>Ausdruck</i>	Adressoperator
	(type)		(<i>Typ</i>) <i>Ausdruck</i>	Typumwandlung (Cast)
	sizeof		sizeof <i>Objekt</i>	Speichergröße
			sizeof (<i>Typ</i>)	Speichergröße
	new	*	new <i>Typ</i>	Objekt anlegen
			new <i>Typ</i> (<i>Ausdruck</i>)	Objekt anlegen,
				mit Initialisierung
			new(<i>Ausdruck</i>) <i>Typ</i>	Objekt platzieren
	new[]	*	new(<i>Ausdruck</i>) <i>Typ</i> (<i>Ausdruck</i>)	Objekt platzieren,
				mit Initialisierung
			new <i>Typ</i> [<i>Ausdruck</i>]	Objekt-Feld anlegen
			new <i>Typ</i> <i>Ausdruck</i>	Objekt-Feld anlegen,
	delete	*		mit Initialisierung
			new(<i>Ausdruck</i>) <i>Typ</i> [<i>Ausdruck</i>]	Objekt-Feld platzieren
			new(<i>Ausdruck</i>) <i>Typ</i> <i>Ausdruck</i>	Objekt-Feld platzieren,
				mit Initialisierung
	delete	*	delete <i>Zeiger</i>	Freigabe
	delete []	*	delete [] <i>Zeiger</i>	Feld-Freigabe

4. Operatoren für Elementzeiger, Assoziativität: von Links nach Rechts

Nr.	Op.		Anwendung	Bedeutung
4	.*	*	<i>Objekt</i> .* <i>Elementzeiger</i>	Elementzugriff über Elementzeiger
	->*	*	<i>Objekt-Zeiger</i> ->* <i>Elementzeiger</i>	Elementzugriff über Elementzeiger
				über Objekt-Zeiger

5. Multiplikations-/Divisionsoperatoren, Assoziativität: von Links nach Rechts

Nr.	Op.	Anwendung	Bedeutung
5	*	<i>Ausdruck</i> * <i>Ausdruck</i>	Multiplikation
	/	<i>Ausdruck</i> / <i>Ausdruck</i>	Division
	%	<i>Ausdruck</i> % <i>Ausdruck</i>	Modulo (Rest bei ganzzahliger Division)

6. Addition/Subtraktion, Assoziativität: von Links nach Rechts

Nr.	Op.	Anwendung	Bedeutung
6	+	<i>Ausdruck</i> + <i>Ausdruck</i>	Addition
	-	<i>Ausdruck</i> - <i>Ausdruck</i>	Subtraktion

7. Shift-Operatoren, Assoziativität: von Links nach Rechts

Nr.	Op.	Anwendung	Bedeutung
7	<<	<i>Ausdruck</i> << <i>Ausdruck</i>	Links-Shift
	>>	<i>Ausdruck</i> >> <i>Ausdruck</i>	Rechts-Shift

8. Vergleichs-Operatoren, Ergebnis vom Typ `bool`, Assoziativität: von Links nach Rechts

Nr.	Op.	Anwendung	Bedeutung
8	<	<i>Ausdruck</i> < <i>Ausdruck</i>	Test auf Kleiner
	<=	<i>Ausdruck</i> <= <i>Ausdruck</i>	Test auf Kleiner-Gleich
	>	<i>Ausdruck</i> > <i>Ausdruck</i>	Test auf Größer
	>=	<i>Ausdruck</i> >= <i>Ausdruck</i>	Test auf Größer-Gleich
9	==	<i>Ausdruck</i> == <i>Ausdruck</i>	Test auf Gleichheit
	!=	<i>Ausdruck</i> != <i>Ausdruck</i>	Test auf Ungleichheit

9. Bit-Operatoren, Assoziativität: von Links nach Rechts

Nr.	Op.	Anwendung	Bedeutung
10	&	<i>Ausdruck</i> & <i>Ausdruck</i>	bitweises UND
11	^	<i>Ausdruck</i> ^ <i>Ausdruck</i>	bitweises exklusives ODER
12		<i>Ausdruck</i> <i>Ausdruck</i>	bitweises inklusives ODER

10. Logische Operatoren, Assoziativität: von Links nach Rechts

Nr.	Op.	Anwendung	Bedeutung
13	&&	<i>Ausdruck</i> && <i>Ausdruck</i>	logisches UND
14		<i>Ausdruck</i> <i>Ausdruck</i>	logisches ODER

11. Fragezeichen-Operator, Assoziativität: von Rechts nach Links

Nr.	Op.	Anwendung	Bedeutung
15	?:	<i>Ausdruck</i> ? <i>Ausdruck</i> : <i>Ausdruck</i>	bedingter Ausdruck

12. Zuweisungen, *a* und *b* seien Ausdrücke, Assoziativität: von Rechts nach Links

Nr.	Op.	Anwendung	Bedeutung
16	=	<i>a</i> = <i>b</i>	Zuweisung
	+=	<i>a</i> += <i>b</i>	Zuweisung, entspricht: <i>a</i> = <i>a</i> + <i>b</i>
	-=	<i>a</i> -= <i>b</i>	Zuweisung, entspricht: <i>a</i> = <i>a</i> - <i>b</i>
	*=	<i>a</i> *= <i>b</i>	Zuweisung, entspricht: <i>a</i> = <i>a</i> * <i>b</i>
	/=	<i>a</i> /= <i>b</i>	Zuweisung, entspricht: <i>a</i> = <i>a</i> / <i>b</i>
	%=	<i>a</i> %= <i>b</i>	Zuweisung, entspricht: <i>a</i> = <i>a</i> % <i>b</i>
	<<=	<i>a</i> <<= <i>b</i>	Zuweisung, entspricht: <i>a</i> = <i>a</i> << <i>b</i>
	>>=	<i>a</i> >>= <i>b</i>	Zuweisung, entspricht: <i>a</i> = <i>a</i> >> <i>b</i>
	&=	<i>a</i> &= <i>b</i>	Zuweisung, entspricht: <i>a</i> = <i>a</i> & <i>b</i>
	^=	<i>a</i> ^= <i>b</i>	Zuweisung, entspricht: <i>a</i> = <i>a</i> ^ <i>b</i>
	=	<i>a</i> = <i>b</i>	Zuweisung, entspricht: <i>a</i> = <i>a</i> <i>b</i>

13. Komma-Operator, Assoziativität: von Links nach Rechts

Nr.	Op.	Anwendung	Bedeutung
17	,	<i>Ausdruck</i> , <i>Ausdruck</i>	Ausdrucksfolge

Das in Argumentlisten eines Funktionsaufrufs auftretende Komma ist nicht der Komma-Operator, sondern dient zur Trennung der einzelnen Argumente:

```
...f( a, b)... // Aufruf einer Funktion f mit zwei Argumenten
```

möchte man im Funktionsaufruf in einem Argument den Komma-Operator verwenden, so ist das Funktionsargument entsprechend zu klammern:

```
...g( (a,b) )... // Aufruf der Funktion mit einem Argument,
                // naemlich dem Ergebnis des Ausdrucks: a,b
```

5.2 Operatorüberladung

C++ bietet die Möglichkeit, Operatoren für eigene Typen zu definieren, man könnte etwa für eine Klasse *A*:

```
class A { ... };
```

den Operator + definieren, so dass in einer Anwendung folgendes möglich wird:

```
...
A a,b;           // zwei A-Objekte
...
... a + b ...    // "Addition" von a und b
...
```

d.h. man könnte mit selbstdefinierten Typen Ausdrücke (Verknüpfung von Operanden mit Operatoren) formulieren.

5.2.1 Standardmäßig für Klassen vorhandene Operatoren

Für Objekte aller (mittels `enum`, `struct` oder `class`) selbstdefinierten Typen sind standardmäßig bereits folgende Operatoren vorhanden und anwendbar:

1. Der (einfache) Zuweisungsoperator `=`.

Bei Klassen (`struct` oder `class`) erfolgt eine Zuweisung der einzelnen Komponenten:

```
class A { ... };
```

```
A a, b;
```

```
...
```

```
a = b;      // Zuweisung von Objekten: es werden die einzelnen
             // Komponenten einander zugewiesen
```

```
...
```

Auch hier wird ggf. auf der rechten Seite der Zuweisung implizit eine Typangleichung vorgenommen:

```
class Bruch {
    private:
        int zaehler;
        int nenner;
    public:
        Bruch ( int z = 0, int n = 1) // Konstruktor mit
            : zaehler(z), nenner(n) // Initialisierungsliste
        { }
        ...
};
```

```
...
```

```
Bruch a (1,3), b, c; // drei Brueche
```

```
...
```

```
b = a;      // Zuweisung, es wird komponentenweise zugewiesen
```

```
...
```

```
c = 7;      // um die Zuweisung eines Bruches an einen Bruch
             // durchfuehren zu koennen, wird mittels des
             // Konstruktors aus dem int 7 ein temporaerer
             // Bruch 7/1 erzeugt und dieser dem c zugewiesen!
```

```
...
```

```
a = b = c;  // auch moeglich, Wert einer Zuweisung ist Wert
             // der linken Seite nach der Zuweisung
```

Der Standard-Zuweisungsoperator ist für eine Klasse nicht definiert, falls

- eine (nicht `static`) Komponente eine Referenz ist, oder

- eine (nicht `static`) Komponente konstant ist, oder
- eine Komponente der Klasse selbst keinen Zuweisungsoperator hat.

2. Der Adress-Operator `&`.

Er liefert bei einem *L-Value* eines beliebigen Typen die Adresse des Speicherbereichs, in dem das Objekt abgespeichert ist:

```
class Bruch { ... };
...
Bruch a;          // Bruch-Objekt
Bruch *p;         // Zeiger auf Bruch
...
p = &a;           // &a: Adresse von a
...
```

Das Ergebnis des Adress-Operators hat natürlich den entsprechenden Adress-Typen — hier etwa: *Adresse eines Bruches*.

3. Der Komma-Operator `,`.

Hier wird — wie üblich — zunächst der linke Operand und dann der rechte Operand ausgewertet und der Wert des Gesamtausdrucks ist der des ausgewerteten rechten Operanden:

```
class A {
    ...
    public:
        void f(void)
        ...
};
...
A a,b;
...
... a,b ... // Komma-Operator
...
```

Ist das Ergebnis eines Komma-Ausdrucks ein Objekt, so kann auf eine seiner Komponenten zugegriffen werden, im obigen Beispiel etwa:

```
...
(a,b).f(); // fuer das Ergebnis des Ausdrucks (a,b), also
           // fuer b, wird die Funktion f aufgerufen
...
```

Selbstdefinierte Typen können natürlich auch in bedingten Ausdrücken vorkommen:

```

class A {
    ...
    public:
        void f(void)
        ...
};
...
A a,b;
...
((Bedingung) ? a : b ).f();
...

```

In diesem Beispiel wird, je nachdem, ob die Bedingung wahr ist oder falsch, die Funktion `f` für das Objekt `a` aufgerufen oder für `b`.

5.2.2 Grundlagen der Operatorüberladung

Folgende Operationszeichen kann man für eigene Datentypen (neu-)definieren:

+	-	*	/	%	^
&		~	!	=	<
>	+=	-=	*=	/=	%=
^=	&=	=	<<	>>	<<=
>>=	==	!=	<=	>=	&&
	++	--	->*	,	->
[]	()	new	new[]	delete	delete[]

Die übrigen, in folgender Tabelle aufgeführten Operationszeichen kann man nicht überladen:

<code>dynamic_cast<>()</code>	<code>static_cast<>()</code>	<code>typeid()</code>	<code>(type)</code>	<code>.</code>	<code>::</code>
<code>reinterpret_cast<>()</code>	<code>const_cast<>()</code>	<code>sizeof</code>	<code>.*</code>	<code>?:</code>	

Man kann keine “neuen” Operationszeichen definieren (etwa `**` für’s Potenzieren). Anzahl der Operanden, Priorität und Assoziativität können auch nicht umdefiniert werden, sondern sind so wie für die Standardtypen im Standard definiert.

Ein binärer Operator `op` (Operator mit zwei Operanden, etwa der Divisionsoperator `/`) kann nur binär überladen werden, wird im Allgemeinen in der Form

`a op b`

aufgerufen (`a` oder `b` oder beide sind hierbei selbstdefinierte Typen).

Ein unärer Operator `op` (Operator mit nur einem Operanden, etwa der Komplement-Operator `~`) kann nur unär überladen werden, wird bei Präfix-Operatoren (Operatoren, die vor ihrem Operanden stehen — die meisten unären Operatoren sind Präfix-Operatoren) in der Form

`op a`

und bei Postfix-Operatoren (Operatoren, die hinter ihrem Operanden stehen) wie folgt

`a op`

aufgerufen (`a` ist hierbei ein selbstdefinierter Typ).

Gibt es einen Operator `op` sowohl unär als auch binär (etwa der Operator `*`, unär als Verweisoperator, binär als Multiplikation) so kann man separat beide Formen überladen.

Überlädt man nur die unäre Form, so ist `op a` erlaubt, `a op b` jedoch nicht, überlädt man nur die binäre Form, so ist `a op b` erlaubt, `op a` jedoch nicht. Man kann natürlich auch beide Formen überladen und dann sind beide Aufrufe möglich!

Die (unären) Inkrement- und Dekrementoperatoren können jeweils (separat) in Präfixform und Postfixform überladen werden.

Überlädt man (jeweils) nur die Präfixform, so sind die Präfix-Aufrufe `++a` und `--a` erlaubt, die Postfix-Aufrufe `a++` und `a--` jedoch nicht, und umgekehrt. Man kann natürlich auch beide Formen überladen.

Während die Syntax der Operatoren durch den Standard festgelegt ist, ist die Semantik (das, was der Operator für einen eigenen Typen machen soll) frei wählbar (der Additionsoperator `+` muss somit für einen eigenen Typen nicht zwangsläufig eine “Addition“ verursachen, dennoch sollte man sich an die Assoziation halten, die ein gewöhnlicher C-Programmierer mit einem Operator verbindet, etwa für die Klasse der Brüche sollte der Operator `+` tatsächlich das Addieren von Brüchen bedeuten!)

Darüberhinaus generiert das System aus überladenen Operatoren selbst keine naheliegenden Kombinationen der überladenen Operatoren — hat man etwa für einen eigenen Typen `T` den Multiplikationsoperator `*` und den Zuweisungsoperator `=` definiert, so gibt es noch lange nicht den Operator `*=` für die multiplikative Zuweisung! Wenn auch dieser erwünscht ist, müsste man diesen (ggf. unter Bezugnahme auf die bereits überladenen Operatoren `*` und `=`) separat noch definieren!

Ein Operator `op` wird überladen, indem man eine Funktion mit dem Namen:

`operator op`

mit entsprechender Signatur und gewünschtem Ergebnistyp deklariert und definiert (beginnt das Token für den Operator nicht mit einem Buchstaben, so kann im Namen der Operatorfunktion das Leerzeichen zwischen dem Schlüsselwort `operator` und dem Operator `op` weggelassen werden, etwa `operator*` für den Multiplikationsoperator `*`!).

Binäre Operatoren haben hierbei zwei Argumente, unäre Operatoren nur eins!

Generell muss bei Operatorüberladungen mindestens eins der beteiligten Argumente von einem “selbstdefinierten“ Typ sein — es dürfen also nicht nur Standardtypen beteiligt sein!

Man muss unterscheiden, ob man einen Operator als “normale“ Funktion überlädt oder als Member-Funktion zu einer Klasse!

5.2.3 Operatorüberladung als globale Funktion

Globale Überladung eines binären Operators

Als globale Funktion wird ein binärer Operator `op` durch eine wie folgt zu deklarierende und definierende Funktion:

```
Typ1 operator op( Typ2, Typ3 );
```

überladen, wobei mindestens einer der beiden Typen `Typ2` oder `Typ3` ein selbstdefinierter Typ (kein Standardtyp) sein muss!

Ist dann `a` ein Objekt vom Typ `Typ2` und `b` ein Objekt vom Typ `Typ3`, so wird in folgendem Ausdruck:

```
a op b
```

diese “Operator-Funktion“ aufgerufen, dieser Ausdruck ist also gleichwertig zu folgendem expliziten und in C++ ebenfalls möglichen Aufruf dieser Funktion:

```
operator op(a,b);
```

das Ergebnis ist jeweils das von der Operator-Funktion gelieferte Resultat (vom Typ `Typ1`).

Als Beispiel wollen wir hier den Multiplikationsoperator `*` für Brüche — als globale Funktion definieren:

```
class Bruch {
private:
    int zaehler;
    int nenner;
public:
    Bruch (int z=0, int n=1) // Konstruktor mit
        : zaehler(z), nenner(n) // Initialisierungsliste
    { }
    ...
    // friend-Deklaration der Multiplikation
    friend Bruch operator *(Bruch, Bruch);
    ...
};
...
// Definition der Multiplikation von Bruechen
Bruch operator * (Bruch a, Bruch b)
{ Bruch tmp;

    tmp.zaehler = a.zaehler * b.zaehler;
    tmp.nenner  = a.nenner  * b.nenner;

    return tmp;
}
```

Da die Multiplikation auf die `private`-Komponenten Zähler und Nenner aller beteiligten Brüche (Parameter `a` und `b` und lokaler `Bruch tmp`) zugreift, muss diese als globale Funktion realisierte Operatorfunktion in der Klasse `Bruch` als `friend` deklariert sein!

Mit dieser Definition sind folgende Anwendungen dieses Operators möglich:

```
...
Bruch a, b, c;
...
a = b * c;    // b mit c multiplizieren und a das Ergebnis zuweisen
```

```

a = operator *(b,c); // gleichwertig
...
a = b * 7; // aus 7 wird mit dem Konstruktor der temporaere Bruch 7/1,
           // b wird mit diesem multipliziert und das Ergebnis
           // dem a zugewiesen
a = operator*(b,7); // gleichwertig
...
a = 7 * c; // aus 7 wird mit dem Konstruktor der temporaere Bruch 7/1,
           // und dieser wird mit c multipliziert und das Ergebnis
           // dem a zugewiesen
a = operator*(7,c); // gleichwertig
...

```

Globale Überladung eines unären Operators

Als globale Funktion wird ein unärer Operator `op` durch eine wie folgt zu deklarierende und definierende Funktion:

```
Typ1 operator op ( Typ2 );
```

überladen, wobei `Typ1` und `Typ2` Typen sind und `Typ2` kein Standardtyp sein darf.

Ist dann `a` ein Objekt vom Typ `Typ2`, so wird bei einem Präfix-Operator im Ausdruck:

```
op a
```

und bei einem Postfix-Operator im Ausdruck:

```
a op
```

diese “Operator-Funktion“ aufgerufen, diese Ausdrücke sind also jeweils gleichwertig zu folgendem expliziten und in C++ ebenfalls möglichen Aufruf dieser Funktion:

```
operator op(a);
```

das Ergebnis ist jeweils das von der Operator-Funktion gelieferte Resultat (vom Typ `Typ1`).

Als Beispiel wollen wir hier den unären Minusoperator `-` für Brüche (negatives Vorzeichen) — als globale Funktion definieren:

```

class Bruch {
private:
    int zaehler;
    int nenner;
public:
    Bruch (int z=0, int n=1) // Konstruktor mit
        : zaehler(z), nenner(n) // Initialisierungsliste
    { }
    ...
    // friend-Deklaration der Operatoren
    friend Bruch operator *(Bruch, Bruch);
    friend Bruch operator -(Bruch);
};
...
// Definition der Minusoperators fuer Brueche

```

```

Bruch operator - (Bruch a)
{ Bruch tmp;

    tmp.zaehler = -a.zaehler;
    tmp.nenner  =  a.nenner;

    return tmp;
}

```

(Auch hier ist aufgrund des Zugriffs auf `private`-Komponenten die `friend`-Deklaration der Operatorfunktion im Klassensrumpf notwendig!)

Anwendung:

```

...
Bruch a, b, c;
...
a = -b;           // a wird der negierte Wert von b zugewiesen
a = operator -(b); // gleichwertig
...
a = b - c; // FEHLER: Operation Bruch - Bruch nicht definiert!
...

```

Mit diesen beiden Operatoren sind jetzt auch komplexere Ausdrücke möglich:

```

Bruch a, b, c, d;
...
a = b * c * -d;
// entspricht:
a = operator *( b, operator *(c, operator -(d)));
...

```

Globale Operatorüberladung für `enum`-Typen

Bei der Überladung eines Operators als globale Funktion muss einer der beteiligten Typen “selbstdefiniert“, also kein Standard-Typ sein. Dies kann auch ein `enum`-Typ sein — es muss nicht unbedingt eine Klasse sein.

Man könnte etwa einen (mathematischen Zahl-) Körper mit 2 Elementen als `enum`:

```
enum galois2 { Null = 0, Eins};
```

(die Namen der Köperelemente sind somit `Null` und `Eins`)

und Addition und Multiplikation in diesem Körper wie folgt definieren:

```

galois2 operator+(galois2 a, galois2 b)
{ if ( a != b )
    return Eins;
  else
    return Null;
}

```

```
galois2 operator*(galois2 a, galois2 b)
{
    if ( ( a == Eins) && ( b == Eins) )
        return Eins;
    else
        return Null;
}
```

5.2.4 Operatorüberladung als Member-Funktion

Man kann eine Operatorfunktion (Funktion zu Überladung eines Operators) auch als Member-Funktion zu einer Klasse deklarieren und definieren.

Als Member-Funktion hat eine solche Operatorfunktion sofort mindestens schon mal ein Argument, nämlich das aktuelle Objekt, für welches sie aufgerufen wurde! Dieses aktuelle Objekt ist damit gleich ein (der erste) Operand für diesen Operator — unäre Operatoren (vom Postfix ++ bzw. -- abgesehen) benötigen somit kein weiteres Argument und binäre Operatoren nur noch genau ein weiteres (zweiter Operand).

Wird das aktuelle Objekt durch die Operator-Funktion nicht geändert, so sollte die Operatorfunktion als `const` vereinbart werden!

Überladung eines binären Operators als Member-Funktion

Ein binärer Operator `op` zu einer Klasse `A` wird als Member-Funktion wie folgt deklariert und definiert:

```
class A {
    private:
        ...
    public:
        // Deklaration des Operators
        Typ1 operator op(Typ2);
        ...
};

Typ1 A::operator op(Typ2 arg)
{ ...
    return Ausdruck;
}
```

Ist jetzt `a` ein Objekt der Klasse `A` und `b` ein Objekt (Variable/Ausdruck) vom Typ `Typ2`, so ist der Ausdruck:

`a op b`

definiert und dieser führt zu folgendem Funktionsaufruf:

`a.operator op(b);`

Sprechweise: Für das Objekt `a` wird die Operatorfunktion `operator op` mit dem Argument `b` aufgerufen. Das Ergebnis ist vom Typ `Typ1`. (Dieser zweite, explizite Schreibweise für den Aufruf der Operatorfunktion ist in C++ ebenfalls erlaubt!) Als Beispiel soll wiederum die Multiplikation zweier Brüche mittels `*` dienen:

```
class Bruch {
private:
    int zaehler;
    int nenner;
public:
    Bruch ( int z=0, int n=1) // Konstruktor mit
        : zaehler(z), nenner(n) // Initialisierungsliste
    { }

    // Deklaration des *-Operators:
    Bruch operator *(Bruch) const;
    ...
};

// Definition des *-Operators:
Bruch Bruch::operator *(Bruch b) const
{ Bruch tmp;
  tmp.zaehler = zaehler * b.zaehler;
  tmp.nenner  =  nenner * b.nenner;

  return tmp;
}
...
```

Als Member-Funktion hat diese Operatorfunktion vollen Zugriff auf alle Komponenten der Klasse `Bruch` — eine `friend`-Deklaration ist nicht erforderlich!

Die Operatorfunktion ist als `const` vereinbart, da das aktuelle Objekt durch die Funktion nicht abgeändert wird und auch für konstante Objekte aufrufbar sein soll!

Zu beachten ist: bei einem als Member-Funktion überladenen Operator wird im ersten Argument keine implizite Typumwandlung durchgeführt.

Anwendung:

```
Bruch a, b;           // variable Brueche
Bruch const c(1,3);   // konstanter Bruch
...
a = c * b; // c mit b multiplizieren und a das Ergebnis zuweisen

a = c.operator *(b); // gleichwertig
...
a = b * 7; // aus 7 wird mit dem Konstruktor der temporaere Bruch 7/1,
           // b wird mit diesem multipliziert und a das Ergebnis zugewiesen
a = b.operator *(7); // gleichwertig
```



```
...
a = 7 * c;          // FEHLER: keine Typumwandlung im ersten Argument!!!
a = 7.operator*(c); // FEHLER: unsinniger Aufruf, 7 ist kein Bruch!!!
...
```

Überladung eines unären Operators als Member-Funktion

Ein unärer Operator `op` (vom Postfix `++` bzw. `--` abgesehen) zu einer Klasse `A` wird als Member-Funktion wie folgt deklariert und definiert:

```
class A {
    private:
        ...
    public:
        // Deklaration des Operators
        Typ1 operator op(void);
        ...
};
...
// Definition des Operators
T A::operator op(void)
{ ...
    return ausdruck;
}
...
```

`Typ1` ist hierbei der Ergebnistyp der Operatorfunktion.

Ist jetzt `a` ein Objekt der Klasse `A`, so wird bei Präfix-Operatoren der Ausdruck:

`op a`

und bei Postfix-Operatoren der Ausdruck:

`a op`

durch folgenden (auch explizit so erlaubten) Aufruf dieser Operatorfunktion umgesetzt:

`a.operator op()`

Ergebnis ist jeweils vom Typ `Typ1`.

Als Beispiel soll wieder die Klasse `Bruch` und der `--`-Operator (negatives Vorzeichen) erhalten:

```
class Bruch {
    private:
        int zaehler;
        int nenner;
    public:
        Bruch ( int z=0, int n=1)    // Konstruktor mit
            : zaehler(z), nenner(n) // Initialisierungsliste
        { }
}
```

```

    // Deklaration des Vorzeichenminus
    Bruch operator -(void) const;
    ...
};

Bruch Bruch::operator -(void) const
{ Bruch tmp;
  tmp.zaehler = - zaehler;
  tmp.nenner = nenner;

  return tmp;
}

```

Als Member-Funktion hat diese Operatorfunktion vollen Zugriff auf alle Komponenten aller beteiligten Brüche, so dass auch hier eine **friend**-Deklaration der Funktion nicht notwendig ist!

Anwendung:

```

Bruch a,b,c;
...
a = -b;           // a wird der negierte Wert von b zugewiesen
a = b.operator -(); // gleichwertig
...
// Aber
a = b - c;        // FEHLER: Operation Bruch - Bruch
...              //      nicht definiert
...
a = b * c * -d;
// entspricht:
a = b.operator*( c.operator *(c.operator -()));
...

```

5.2.5 Zusammenfassung: Operatorüberladung global oder als Member

Der wesentliche Unterschied zwischen Operatorüberladung als globale Funktion oder als Memberfunktion ist (neben der unterschiedlichen Art und Weise, sie zu deklarieren und zu definieren) der, dass bei Realisierung als Member-Funktion keine implizite Typumwandlung beim ersten Operanden durchgeführt wird und dass globale Member-Funktionen häufig als **friend** des beteiligten selbstdefinierten Datentypes (mindestens einer der Operanden darf kein Standardtyp sein!) deklariert werden muss (z.B. wenn auf **private**-Komponenten des Types zugegriffen werden soll!).

Ansonsten gilt in Bezug auf Funktionsüberladung für Operatorfunktionen das gleiche wie für gewöhnliche Funktionen: Man kann für einen eigenen Typen einen (binären) Operator mehrfach — mit unterschiedlicher — Signatur überladen (Konstanz gehört wohlbemerkt mit zur Signatur!). Anhand der konkreten Argumente wird beim

aktuellen Aufruf entschieden, welche der Überladungen genommen wird. Ggf. wird hierbei eine Typangleichung vorgenommen. Stehen mehrere gleichwertige Operatoren zur Verfügung, so ist dies ein Fehler!

```
class A {
    ...
public:
    ... operator+(const char *); // 2. Operand ist Zeiger auf const char
    ... operator+(double);      // 2. Operand ist double
    ... operator+(double) const; // 2. Operand ist double, aber
                                // konstante Member-Funktion
    ...
};
...
A a;          // variables Objekt
const A b;    // Konstante
int i;
double x;
...
... a + x ...; // OK, Aufruf von: operator+(double);
... a + i ...; // OK, Aufruf von: operator+(double);
                // hierbei wird int i nach double umgewandelt
... b + x ...; // OK, Aufruf von: operator+(double) const;
... a + "hallo" ... // OK, Aufruf von operator+(const char *);
...

```

5.2.6 Keine Defaultparameter für Operatorfunktionen

Da Operatorfunktionen implizit in Ausdrücken aufgerufen werden und anhand des Ausdrucks die Operandenzahl feststehen muss, können Operatorfunktionen (mit Ausnahme des Funktionsaufrufoperators ()) keine Defaultparameter haben!

5.2.7 Operatorüberladung bei “großen“ Typen und Ergebnistypen

Man könnte durch Operatorüberladung für die Klasse `Bruch` die gewöhnliche Arithmetik (+ für Vorzeichen und Addition, - für Vorzeichen und Subtraktion, * für Multiplikation, / für Division, ...) nachbilden.

Damit werden dann komplexere Ausdrücke für Brüche möglich, etwa:

```
Bruch a, b, c, d;
a = b * (-c) - d * 5 + (-a);
...

```

Derartige Ausdrücke werden durch entsprechende Aufrufe der Operatorfunktionen umgesetzt, es kommt dabei schnell zu einer ganzen Reihe von (Operator-)Funktionsaufrufen (in diesem Beispiel sind es bereits 6 Aufrufe, für jeden Operator ein Aufruf).

Insbesondere bei großen Typen sollte man hier die Anzahl lokaler Variablen und “großer” Funktionsparameter minimieren und überlegen, ob nicht zumindest der ein- oder andere Parameter als (“kleine”) Referenz (ggf. auf `const`) vereinbart werden kann.

Darüberhinaus hängt vom Ergebnistyp der Operatorfunktion ab, in welchem Zusammenhang der Operator aufgerufen und was mit dem Ergebnis der Operation weiter angestellt werden kann.

Im nächsten Unterabschnitt werden wir sehen, dass man auch den Zuweisungsoperator = (zwar nur als Member-Funktion) überladen kann (bei Klassen mit dynamischen Komponenten muss dieser im Allgemeinen auch überladen werden)!

Wenn man ihn wie folgt überlädt (deklariert und definiert):

```
class A {
    ...
public:
    void Operator=(const A& b);
    ...
};
```

ist zwar folgender Aufruf möglich:

```
A a,b;
a = b;
...
```

jedoch folgender Aufruf nicht:

```
A a,b,c;
a = b = c;    // FEHLER!!!
...
```

da dieser Ausdruck intern wie folgt:

```
a = ( b = c );
```

ausgewertet wird, der Teilausdruck (`b = c`) jedoch kein Ergebnis hat (`void`), welches dem `a` zugewiesen werden könnte.

Soll auch eine solche Zuweisungskette möglich sein, müsste man den Operator wie folgt deklarieren und definieren:

```
class A {
    ...
public:
    A& Operator=(const A& b);
    ...
};
```

wobei man noch überlegen könnte, ob das Ergebnis eine normale Referenz auf `A` oder besser eine Referenz auf `const A` sein sollte.

5.2.8 Besonderheiten spezieller Operatoren

Nur als nicht statische Member-Funktionen überladbare Operatoren

Folgende Operatoren können nur als nicht statische Member-Funktionen überladen werden:

=	Zuweisungsoperator
[]	Feldindizierung
()	Funktionsaufruf
->	Komponentenzugriff über Adresse

(Die übrigen Zuweisungsoperatoren += -= *= /= %= &= |= ^= <<= >>= und der Komponentenzugriffsoperator ->* für Komponentenzeiger können merkwürdigerweise auch global überladen werden!)

Der Zuweisungsoperator =

Der Zuweisungsoperator ist, wie früher bereits erwähnt, standardmäßig zunächst mal für alle Typen vorhanden — für Klassen allerdings nur dann, wenn der Zuweisungsoperator für alle Komponenten vorhanden ist und keine nicht statische Konstante oder nicht statische Referenz als Komponente auftritt!

Bei Objekten einer Klasse:

```
class A { ... };
...
A a,b;
a = b;
...
```

verursacht die Standardzuweisung eine Zuweisung der einzelnen Komponenten, d.h. für jede der Komponente wird der zum Komponententyp gehörende Zuweisungsoperator aufgerufen (dieser muss wie gesagt verfügbar sein!).

Der Zuweisungsoperator kann nur als nicht statische Memberfunktion einer Klasse A überladen werden. Natürlich ist die Semantik eines selbstgeschriebenen Zuweisungsoperators frei wählbar, allerdings ist es eine schlechte Idee, durch den Zuweisungsoperator etwas anderes als eine Zuweisung zu implementieren!

Üblicherweise deklariert man einen Zuweisungsoperator zu einer Klasse A in einer der folgenden Formen:

```
A& A::operator=(const A&);
```

oder

```
const A& A::operator=(const A&);
```

und sorgt in der Implementierung dafür, dass der Wert des Zuweisungsausdrucks der Wert der linken Seite nach der Zuweisung ist. Damit sind dann auch geschachtelte Aufrufe möglich:

```
A a, b, c;
a = b = c;
...
```

Bei ernsthafter, anwendbarer Überladung der Zuweisung muss immer der Sonderfall *Zuweisung an sich selber*, also die spezielle Anwendung der Form:

```
A a;
a = a;    // Zuweisung an sich selber
...
```

abgefangen und besonders behandelt werden, etwa:

```
class A {
    ...
public:
    A& operator=( const A& b);
    ...
};
A& A::operator=(const A &b)
{ if ( &b == this )    // Zuweisung an sich!
    return *this;

    // sonst: keine Zuweisung an sich
    ...

    return *this;
}
```

Klassen mit dynamischen Komponenten benötigen neben Destruktor und Copy-Konstruktor im Allgemeinen immer einen selbstdefinierten Zuweisungsoperator — ansonsten bekommt man wie beim Copy-Konstruktor Probleme im Freispeicher:

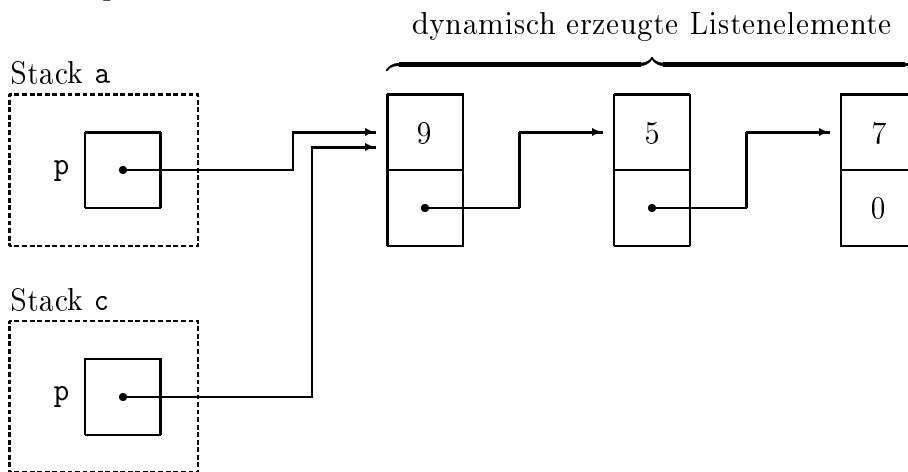
```
class Stack { // neuer Datentyp hat Namen:  Stack
protected:  // Impl.-Details, nicht oeffentlich
    struct listel {      // eingebetteter Typ:
        int eintrag;     // Listenelement
        listel *next;
    } *p;                // Zeiger auf Listenanfang

public:       // oeffentliche Schnittstelle
    Stack();   // Konstruktor, initialisiert leere Liste
    Stack( const Stack &); // Copy-Konstruktor
    void push(int); // Funktion zum Einkellern
    int pop(void);  // Funktion zum Auskellern
    ~Stack();       // Destruktor, Freigabe der Liste
};
```

Anwendung:

```
...
void fkt(void)
{ Stack a, c;
  a.push(7);
  a.push(5);
  a.push(9);
  c = a;    // Standardzuweisung
  ...
  return;
}
```

Die Situation nach der Standardzuweisung, vor dem Funktionsende sieht wiederum wie folgt aus:



was beim Funktionsende mit den dazugehörigen Destruktoraufrufen zum Problem führt: zweimalige Freigabe desselben dynamisch reservierten Speicherbereichs. Abhilfe bietet wiederum entweder:

1. Die Zuweisung verbieten, indem man etwa den Zuweisungsoperator im **private**-Teil des Klassenrumpfes deklariert:

```
class Stack {
private:
  ...
  // Zuweisung verbieten:
  // Implementierung kann entfallen!
  Stack& operator=(const Stack&);
  ...
};
...
Stack a, b;
a = b;    // FEHLER: Zuweisung nicht verfuegbar!
...
```

oder

2. die Zuweisung selbst vernünftig implementieren:

```
class Stack {
protected:    // Impl.-Details, nicht oeffentlich
    struct listel {        // eingebetteter Typ:
        int eintrag;      // Listenelement
        listel *next;
    } *p;          // Zeiger auf Listenanfang

public:
    // Zuweisung deklarieren:
    Stack& operator=(const Stack&);
    ...
};
...
// und implementieren:
Stack& Stack::operator=(const Stack b)
{ if ( &b == this)    // Zuweisung an sich!
    return *this;

    // Lineare Liste des aktuellen Objektes
    // erstmal freigeben:
    listel *tmp_neu;
    while ( (tmp_neu = p ) != 0)
    { p = p->next;
      delete tmp_neu;
    }

    // neue Liste wie beim Copy-Konstruktor als Kopie
    // der Liste zu b erzeugen:
    listel *tmp_alt;

    if ( (tmp_alt = b.p) != 0) // falls Liste von b nicht leer
    { // erstes Element kopieren
      p = new listel;
      p->inhalt = tmp_alt->inhalt;
      p->next   = 0;

      tmp_neu = p;
      tmp_alt = tmp_alt->next;

      // ggf. alle weiteren Elemente kopieren
      while ( tmp_alt != 0 )
      { tmp_neu->next = new listel;
        tmp_neu->next->inhalt = tmp_alt->inhalt;
```



```

        tmp_neu->next->next = 0;

        tmp_alt = tmp_alt->next;
        tmp_neu = tmp_neu->next;
    }
}
return *this;
}

```

Wie man sieht, sind Implementierung von Copy-Konstruktor und Zuweisungsoperator sehr ähnlich. In vielen Fällen kann man den Copy-Konstruktor auf den Zuweisungsoperator zurückführen:

```

// Zuweisungsoperator sei vorhanden!

// Definition des Copy-Konstruktors,
// Rueckgriff auf Zuweisung:
Stack::Stack(const Stack& b)
{
    p = 0;    // Zunaechst mal leer!

    *this = b; // Rueckfuehrung auf Zuweisung
}

```

Der Operator [] zur Feldindizierung

Auch der Operator [] zur Feldindizierung kann nur als nicht statische Member-Funktion mit einem Argument überladen werden (erster Operand: aktuelles Objekt, zweiter Operand: Argument):

```

class A {
    ...
public:
    // Deklaration des Index-Operators
    Typ1 operator[](Typ2);
    ...
};

...
// Definition des Index-Operators
Typ1 A::operator[] (Typ2 b)
{ ... }

```

Der Aufruf dieses Operators geschieht wie folgt:

```

A a;
Typ2 arg;
...

```

```
// impliziter Aufruf:
...a[arg]...;      // entspricht: a.operator[](arg)
...
// expliziter Aufruf:
...a.operator[](arg)...;  // auch erlaubt!
...
```

Ergebnis des Aufrufs ist jeweils vom Typ `Typ1`. Der “Indextyp” `Typ2` kann beliebig sein, muss also nicht unbedingt ganzzahlig sein (es sind etwa auch Zeichenketten als “Indextyp” und somit ein *Assoziativer Speicher* möglich!).

Als Beispiel soll hier ein Datentyp für ein `double`-Feld vorgestellt werden, bei dem der Indexoperator so überladen wird, dass Feldüber-/unterlauf abgefangen werden:

```
class DoubleFeld {
private:
    double *feldzeiger;
    int feldlaenge;
public:
    // lokale Fehlerklassen
    struct FeldUnterlauf {};
    struct FeldUeberlauf {};

    // Konstruktor, Destruktor und Zuweisung wegen
    // dynamischer Komponente:
    DoubleFeld(unsigned int);
    DoubleFeld(const DoubleFeld*);
    DoubleFeld& operator=(const DoubleFeld&);

    // Index-Operator:
    double& operator[](int);           // fuer variable Felder
    const double& operator[](int) const; // fuer const Felder
    ...
};
...
// Definition des Konstruktors:
DoubleFeld::DoubleFeld( unsigned int laenge)
{ feldzeiger = new double[laenge];
  feldlaenge = laenge;
}
// Definition des Index-Operators fuer variable Felder:
double& DoubleFeld::operator[](int i)
{ if ( i < 0 )
    throw DoubleFeld::FeldUnterlauf();
  if ( i >= feldlaenge )
    throw DoubleFeld::FeldUeberlauf();
  return feldzeiger[i];
}
```

```
// Definition des Index-Operators fuer const Felder:
const double& DoubleFeld::operator[](int i) const
{ if ( i < 0 )
    throw DoubleFeld::FeldUnterlauf();
  if ( i >= feldlaenge )
    throw DoubleFeld::FeldUeberlauf();
  return feldzeiger[i];
}
```

Hier sind zwei Versionen des Indexoperators definiert, einmal die nicht `const`-Form, die ein `double`-Objekt per Referenz zurückgibt, damit kann dem Ergebnis etwa auch etwas zugewiesen werden.

Bei der `const`-Form wird eine Referenz auf `const double` zurückgegeben (ansonsten gleiche Implementierung), damit kann etwa dem Ergebnis dieses Operators nichts zugewiesen werden:

```
DoubleFeld a(100);          // Feld der Laenge 100
const DoubleFeld b(10);     // const Feld der Laenge 10
...
a[0] = .1;                  // OK!
++a[7];                     // OK!
cout << a[99];              // OK!
a[100] = ...;               // LAUFZEITFEHLER: muesste Ueberlauf auswerfen
a[-5] = ...;                // LAUFZEITFEHLER: muesste Unterlauf auswerfen
...
cout << b[5];               // OK!
b[5] = 7;                   // COMPILERFEHLER: Ergebnis ist const,
                           // Zuweisung nicht erlaubt!
b[7]++;                     // COMPILERFEHLER: Ergebnis ist const,
                           // Erhoehung nicht erlaubt!
...b[-1]...;                // LAUFZEITFEHLER: muesste Unterlauf auswerfen
...b[11]...;                // LAUFZEITFEHLER: muesste Ueberlauf auswerfen
...
```

Der Funktionsaufruf-Operator ()

Der Funktionsaufruf-Operator () kann auch nur als nicht statische Member-Funktion überladen werden.

Der erste Operand für diesen Operator ist das aktuelle Element der Klasse, für welchen er definiert wird — in gewissem Sinne kann dieser Operator kein, ein oder mehrere weitere “Operanden” — hier spricht man besser von weiteren Argumenten — haben. Der Einfachheit halber fangen wir mit dem Funktionsaufruf-Operator mit einem zusätzlichen Argument an:

```

class A {
    ...
public:
    // Deklaration der Operatorfunktion operator() mit einem
    // Argument vom Typ Typ2 und Ergebnis vom Typ Typ1:
    Typ1 operator() (Typ2);
    ...
};

// Definition dieser Funktion:
Typ1 A::operator() (Typ2 arg)
{ ... }
...

```

Der Ausdruck:

```

A a;
...
a(7);
...

```

wird vom System umgesetzt zu:

```
a.operator() ( 7 );
```

d.h. zum Objekt `a` wird die Operatorfunktion `operator()` mit dem Argument `7` aufgerufen. (Das *Objekt* `a` kann wie eine *Funktion* verwendet werden: `a(7)`. Man spricht manchmal auch von *Funktionsobjekten*!)

Dieser explizite Aufruf `a.operator() (7)` der Operatorfunktion `operator()` ist syntaktisch in C++ ebenfalls erlaubt!

Der Funktionsaufrufoperator kann mit unterschiedlicher Signatur überladen werden — Defaultparameter sind hier auch erlaubt:

```

class A {
    ...
public:
    // ein double Argument, Standardparameter 1.0
    int operator() ( double = 1.0);

    // zwei Argumente, eins int, eins char
    int operator() ( int, char);

    // zwei Argumente, eins int, eins char,
    // aber konstante Member-Funktion
    int operator() ( int, char) const;

    // ein Argument, Typ char *
    int operator() ( char*);

```

```

    // ein Argument, Typ const char*
    int operator() ( const char*);
    ...
};

// Anwendung
A a;
const A b;
char s[10];
double x,
...
a(x);      // a.operator() (x),
           // Aufruf von operator() ( double = 1.0);
a();       // a.operator() (),
           // Aufruf von operator() ( double = 1.0);
a(7,'c');  // a.operator() (7,'c'),
           // Aufruf von operator() ( int, char);
b(7,'c');  // b.operator() (7,'c'),
           // Aufruf von operator() ( int, char) const;
a(s);      // a.operator() (s),
           // Aufruf von operator() ( char*);
a("hallo"); // a.operator() ("hallo"),
           // Aufruf von operator() ( const char*);
...

```

Der Operator ->

Die Überladung des Operators -> ist ebenfalls nur als nicht statische Member-Funktion zu einer Klasse möglich — und zwar nur unär:

```

class A {
    ...
    public:
        Typ operator->(void);
    ...
};

```

Allerdings muss der Ergebnistyp `Typ` ein Typ sein, für den selbst wiederum der `->-` Operator anwendbar ist (i.Allg. — aber nicht unbedingt — ein Zeiger auf einen Struktur-/Klassentyp).

Der Aufruf:

```

A a;
...
... a->m ... ;
...

```

wird umgesetzt zu:

```
... (a.operator->()) -> m ...; // Operatorfunktion,
                             // auch so explizit aufrufbar!
```

d.h. es wird die Funktion `a.operator->()` (kein Argument) aufgerufen und auf das Ergebnis hiervon wird selbst der `->`-Operator (mit Komponentennamen `m`) angewendet (es wird hierbei davon ausgegangen — und vom Compiler überprüft — dass für das Ergebnis `erg` der Operatorfunktion `operator->()` der Aufruf `erg->m` möglich ist!).

Der Aufruf `a->m` der Operatorfunktion `operator->()` ist somit unabhängig von dem dahinterstehenden `m` — dieses `m` ist also kein Argument zu dieser Operatorfunktion — das `m` spielt erst beim Ergebnis der Operatorfunktion eine Rolle!

Das Überladen des verwandten, binären Operators `->*` (für Komponentenzeiger) scheint nach den gewohnten Regeln zu funktionieren,

- entweder global:

```
T1 operator->* ( T2, T3);
```

(T1, T2 und T3 irgendwelche Typen, wobei T1 oder T2 oder T3 kein Standardtyp ist.)

Der Aufruf

```
T2 a;
T3 b;
...
... a ->* b ...
...
```

wird als

```
... operator->* ( a, b) ...
```

umgesetzt!

- oder als Member-Funktion zu einer Klasse:

```
class A {
    ...
    public:
    T1 operator->*( T2);
    ...
};
```

(T1 und T2 beliebige Typen!)

Der Aufruf:

```

A a;
T2 b;
...
... a ->* b ...
...

```

wird als

```
... a.operator->*(b) ...
```

umgesetzt!

Die Operatoren ++ und --

Die Operatoren ++ und -- (Postfix oder Präfix) können jeweils sowohl global als auch als Member-Funktion überladen werden.

Beim Überladen werden die Präfix-Formen als unäre Operatoren aufgefasst und die Postfix-Formen als binäre Operatoren, wobei hier als zweites Argument ein *Dummy-Argument* vom Typ `int` vorgesehen ist. (Dieses in Deklaration und Definition der Operatorfunktion stehende Dummy-Argument dient zur formalen Unterscheidung der jeweiligen Postfix-Form von der Präfixform und sollte in der Implementierung nicht verwendet werden!)

Als globale Überladung für einen eigenen Typen `A` könnte dies wie folgt aussehen:

```

A& operator++(A&);           // Praefix
A  operator++(A&, int);      // Postfix
A& operator--(A&);           // Praefix
A  operator--(A&, int);      // Postfix
...
A a, b;
...++a...                    // Praefix, Aufruf von A& operator++(A&)
...--a...                    // Praefix, Aufruf von A& operator--(A&)
...b++...                    // Postfix, Aufruf von A operator++(A&, int)
...b--...                    // bzw. A operator--(A&, int)
                             // anstelle des Dummy--Wertes kommt ein
                             // zufaelliger an!
...

```

Die Semantik der Operatoren ist wiederum beliebig — man sollte sich jedoch an den Standard halten, dass durch ++ etwas “erhöht” und durch -- etwas “erniedrigt” wird. Üblicherweise sollte dann die Präfixform den erhöhten Wert (per Referenz) zurückgeben und in der Postfixform den “alten” Wert, und zwar nicht als Referenz (dies macht die Implementierung der Postfix-Operatoren aufwändiger und i.Allg. auch langsamer als die der Präfixform!).

Als Member-Funktionen müsste die Überladung von ++ und -- wie folgt aussehen:

```

class A {
    ...
public:
    A& operator++();      // Praefix
    A  operator++(int);   // Postfix
    A& operator--();      // Praefix
    A  operator--(int);   // Postfix
    ...
};

...
A a, b;
...++a...      // Praefix, Aufruf von A& A::operator++()
...--a...      // Praefix, Aufruf von A& A::operator--()
...b++...      // Postfix, Aufruf von A A::operator++(int)
...b--...      // bzw. A A::operator--(int)
                // anstelle des Dummy--Wertes kommt ein
                // zufaelliger an!
...

```

Die Operatoren `new`, `new[]`, `delete` und `delete[]`

Diese Operatoren können sowohl global als auch als Member einer Klasse überladen werden.

Eine genauere Erläuterung der (schwierigen) Überladung dieser Operatoren würde den Rahmen dieser Ausarbeitung sprengen.

Die Operatoren `&&` und `||`

Diese Operatoren können sowohl global als auch als Member-Funktion überladen werden.

Das Besondere bei diesen Überladungen ist, dass die “Kurzschlussauswertung” (erst Auswertung des linken Operanden und dann erst — und zwar nur bei Bedarf — Auswertung des rechten Operanden) nur bei den Standardtypen — und nicht bei den Überladungen durchgeführt wird. D.h. bei einer Überladung von `&&` bzw. `||` werden auf jeden Fall beide Operanden ausgewertet!

5.3 Übersicht über die Überladungsmöglichkeiten

Folgende Tabelle gibt eine Übersicht über die Überladungsmöglichkeiten der C++-Operatoren.

In der Spalte 2 ist vermerkt, ob der entsprechende Operator unär (*u*) oder binär (*b*) ist. (Bei einigen Operatoren, wo dies nicht ganz klar ist, ist der entsprechende Eintrag offen!)

Die in der Spalte 5 mit * gekennzeichneten Operatoren sind hierbei vom Standard für selbstdefinierte Typen bereits vordefiniert.

Op.	Bedeutung	2	überladbar		5	Bemerkung
			global	als Member		
::	Klassenzuordnung Namensbereichszuordnung Zugriff auf globalen Namen		nein	nein		
.	Komponentenzugriff über Objekt	<i>b</i>	nein	nein		nur unär überladbar!
->	Komponentenzugriff über Zeiger	<i>b</i>	nein	ja		
[]	Feldindizierung	<i>b</i>	nein	ja		
()	Funktionsaufruf		nein	ja		
++	Postfix-Inkrementierung	<i>u</i>	ja	ja		beliebig viele weitere Operanden binär mit Dummy-Argument vom Typ int
--	Postfix-Dekrementierung	<i>u</i>	ja	ja		binär mit Dummy-Argument vom Typ int
typeid()			nein	nein		
dynamic_cast<>()			nein	nein		aber Konvers.-Oper.
static_cast<>()			nein	nein		aber Konvers.-Oper.
reinterpret_cast<>()			nein	nein		aber Konvers.-Oper.
const_cast<>()			nein	nein		aber Konvers.-Oper.
!	logische Negation	<i>u</i>	ja	ja		* aber Konvers.-Oper. schwierig noch schwieriger schwierig noch schwieriger
~	Bit-Komplement	<i>u</i>	ja	ja		
++	Präfix-Inkrementierung	<i>u</i>	ja	ja		
--	Präfix-Dekrementierung	<i>u</i>	ja	ja		
-	negatives Vorzeichen	<i>u</i>	ja	ja		
+	positives Vorzeichen	<i>u</i>	ja	ja		
*	Verweisoperator	<i>u</i>	ja	ja		
&	Adressoperator	<i>u</i>	ja	ja		
(type)	Typumwandlung (Cast)		nein	nein		
sizeof	Speichergröße	<i>u</i>	nein	nein		
new	Objekt anlegen		ja	ja		
new[]	Objekt-Feld anlegen		ja	ja		
delete	Freigabe		ja	ja		
delete []	Feld-Freigabe		ja	ja		
.*	Komponentenzeiger	<i>b</i>	nein	nein		
->*	Komponentenzeiger	<i>b</i>	nein	nein		
*	Multiplikation	<i>b</i>	ja	ja		
/	Division	<i>b</i>	ja	ja		
%	Modulo	<i>b</i>	ja	ja		
+	Addition	<i>b</i>	ja	ja		
-	Subtraktion	<i>b</i>	ja	ja		
<<	Links-Shift	<i>b</i>	ja	ja		zur Ausgabe verwenden!
>>	Rechts-Shift	<i>b</i>	ja	ja		zur Eingabe verwenden!

Fortsetzung der Tabelle mit den Überladungsmöglichkeiten der C++-Operatoren:

Op.	Bedeutung	2	überladbar		5	Bemerkung
			global	als Member		
<	Test auf Kleiner	<i>b</i>	ja	ja		
<=	Test auf Kleiner–Gleich	<i>b</i>	ja	ja		
>	Test auf Größer	<i>b</i>	ja	ja		
>=	Test auf Größer–Gleich	<i>b</i>	ja	ja		
==	Test auf Gleichheit	<i>b</i>	ja	ja		
!=	Test auf Ungleichheit	<i>b</i>	ja	ja		
&	bitweises UND	<i>b</i>	ja	ja		
^	bitweises exklusives ODER	<i>b</i>	ja	ja		
	bitweises inklusives ODER	<i>b</i>	ja	ja		
&&	logisches UND	<i>b</i>	ja	ja		kein “Kurzschluss“
	logisches ODER	<i>b</i>	ja	ja		kein “Kurzschluss“
?:	bedingter Ausdruck	<i>b</i>	nein	nein		
=	Zuweisung	<i>b</i>	nein	ja	*	
+=	Zuweisung	<i>b</i>	ja	ja		
-=	Zuweisung	<i>b</i>	ja	ja		
*=	Zuweisung	<i>b</i>	ja	ja		
/=	Zuweisung	<i>b</i>	ja	ja		
%=	Zuweisung	<i>b</i>	ja	ja		
<<=	Zuweisung	<i>b</i>	ja	ja		
>>=	Zuweisung	<i>b</i>	ja	ja		
&=	Zuweisung	<i>b</i>	ja	ja		
^=	Zuweisung	<i>b</i>	ja	ja		
=	Zuweisung	<i>b</i>	ja	ja		
,	Azdrucksfolge	<i>b</i>	ja	ja	*	

5.4 Konvertierungs–Operatoren

Entwirft und definiert man eine neue Klasse **A**, so kann man mittels Konstruktoren “Umwandlungen“ von einem bereits vorhandenen Typen **B** in den Neuen Typen **A** definieren:

```
class A {
    ...
public:
    A(B);           // konstruiere aus einem B ein A!
    ...           // B bekannter Typ
};
```

Da nicht als **explicit** vereinbart, wird dieser Konstruktor vom System ggf. auch implizit zur Typumwandlung von **B** nach **A** verwendet:

```

int fkt(A);
B b;
...
fkt(b);    // OK: aus b wird mit dem Konstruktor ein A
           // erzeugt und Funktion mit diesem A aufgerufen

```

Doch wie kann man ein Objekt des neuen Typen A in den alten Typen B umwandeln, ohne in die Definition des alten Typen B eingreifen zu müssen oder zu können (etwa wenn B ein Standardtyp oder ein seit langem bewährter Typ ist, dessen Definition nicht mehr abgeändert werden kann)?

Hierzu kann als Memberfunktion zur neuen Klasse A wie folgt ein *Konversions-Operator* von A nach B deklariert (und entsprechend definiert) werden:

```
// B bekannter Typ, moeglicherweise auch ein Standardtyp
```

```

class A {
    ...
    public:
        // Deklaration des Konversions-Operators:
        operator B();
    ...
};
...
// Definition des Konversions-Operators:
A::operator B()
{ B tmp;
    ...
    return tmp;
}
...

```

Man definiert also eine Member-Funktion mit dem Namen:

```
operator Typ_in_den_umgewandelt_werden_soll
```

ohne Argument und formal ohne Rückgabotyp (wie Konstruktor und Destruktor).

Obwohl bei Deklaration und Definition kein Rückgabotyp angegeben ist, wird dennoch im Anweisungsteil der Konversionsfunktion ein Wert des Typen, in den umgewandelt werden soll, mittels **return** zurückgegeben!

Für Brüche soll eine Typumwandlung nach **double** definiert werden:

```

class Bruch {
    private:
        int zaehler;
        int nenner;
    public:
        Bruch ( int z=0, int n=1)    // Konstruktor mit
            : zaehler(z), nenner(n)  // Initialisierungsliste
        { }
}

```

```

    // Deklaration der Konversion nach double:
    operator double();
    ...
};
...
// Definition der Konversion nach double:
Bruch::operator double()
{
    return (double) zaehler/nenner;
}

```

Überall, wo jetzt in Ausdrücken ein **Bruch** auftritt, keine für Brüche passende Operation (Funktion oder Operator) vorhanden ist, aber eine für **double** passende Operation existiert, wird mittels dieser Konversion aus dem **Bruch** ein **double**-Wert erzeugt und die Operation mit diesem **double**-Wert durchgeführt:

```

// Anwendung:
double fkt(void)
{
    Bruch a,b;
    double f(double);
    double x, y;
    ...
    x = a;          // implizite Konversion
    y = f(b);       // implizite Konversion

    ... = y * (double) a; // explizite Konversion
    ... = y * double(a);  // explizite Konversion
    ... = y * static_cast<double>(a); // explizite Konversion

    ... = y * a; // implizite Konversion, da kein
                // anderer *-Operator vorhanden

    return a; // implizite Konversion
}

```

5.5 Operatorüberladung und Konversionsoperatoren in der Standardbibliothek

Konversionsoperatoren und Operatorüberladung werden extensiv in der Standardbibliothek angewendet.

Die Standardausgabe **cout** und Standardfehlerausgabe sind etwa Objekte der Klasse **ostream** (Ausgabestrom) und der Operator **<<** ist für ein Objekt dieser Klasse als ersten Operanden und alle erdenklichen Typen als zweiten Operanden (global oder als

Memberfunktion zu Klasse `ostream`) so überladen, dass die üblichen Aufrufe möglich werden:

```
int i;
double x;
char s[100];
...
cout << i;
cout << x;
cout << s;
cout << "hallo";
...
```

Darüberhinaus sind die Überladungen von `<<` so definiert, dass das Ergebnis der Operation wiederum der erste Operand, also das Objekt vom Typ `ostream` ist. Somit sind aufgrund der Assoziativität von `<<` (von Links nach Rechts) Aufrufe folgender Art möglich:

```
cout << i << x ;
      └──┬──┘
      cout  << x ;
```

zuerst wird dir Ausgabeoperation `cout << i` aufgerufen, das Ergebnis hiervon ist wiederum `cout`, und mit dem Ergebnis wird die zweite Ausgabeoperation durchgeführt, also `cout << x`.

Auch Konversionsoperatoren werden verwendet, etwa um ein Objekt vom Typ `istream` (Eingabestrom) bei Bedarf in einen boolschen Wert umzuwandeln, etwa:

```
if ( cin )    // istream nach bool: interpretiert als:
{             // falls istream in Ordnung
    ...
}
```

Da auch der Eingabeoperator `>>` für alle erdenklichen Typen überladen ist und jeweils wiederum den `istream` nach dem Lesen als Wert zurückgibt, sind auch Schleifen folgender Art möglich:

```
double x;
...
while ( cin >> x ) // solange der istream nach Lesen eines double
{                 // noch in Ordnung ist, d.h. solange das Lesen
    ...           // eines double klappt!
}
...
```

5.6 Konversionen, Konstruktoren und Überladung

Viele Autoren warnen vor der leichtfertigen Definition von zu vielen Konversionsoperatoren und zu vielen Überladungen von Funktionen und Operatoren.

Hierdurch sind leicht — vor allem im Zusammenhang mit Konstruktoren — in Ausdrücken und Funktionsaufrufen Mehrdeutigkeiten möglich, d.h. es gibt unterschiedliche gleichwertige Möglichkeiten, die Typen im Ausdruck umzuwandeln, um eine passende Funktions aufrufen zu können. Derartige Mehrdeutigkeiten werden vom Compiler als Fehler gemeldet.

Es wird geraten, anstelle eines Konversionsoperators, etwa:

```
Bruch::double();
```

lieber eine Memberfunktion:

```
double Bruch::als_double();
```

zu implementieren, mit der man nur durch expliziten Aufruf eine Umwandlung von `Bruch` nach `double` erreichen kann. Erst, wenn eine solche Umwandlung sehr häufig verwendet wird, sollte man die unelegante explizite Umwandlung durch eine implizite Umwandlung mit einem Konversionoperator ermöglichen.

Kapitel 6

Templates

C++ ist (wie C) eine typgebundene Sprache. Es kommt häufig vor, dass eine Funktion, welche eine gewisse Aktion durchführen soll, mehrfach für unterschiedliche Typen definiert werden muss.

Hier konnte man sich in C (und auch in C++) mit Präprozessor-Makros mit Argumenten behelfen, doch handelt man sich hierbei Nachteile bezüglich Argumentprüfung, Seiteneffekte, ... ein.

Das gleiche Problem tritt auch bei Klassen auf, mit denen andere Objekte verwaltet werden (sog. *Containerklassen*), so ist z.B. ein Stack für `int`-Größen etwas anderes als ein Stack für `double`-Werte. Hier wären zwei Klassen zu definieren, die strukturell sehr ähnlich sind.

Abhilfe bieten hier *Templates* oder auch *Schablonen* genannt.

Durch Templates werden *Muster* für Funktionen oder Klassen vorgegeben, aus denen in der Anwendung im konkreten Fall für konkrete Typen die wirklichen (gewöhnlichen) Funktionen und Klassen erzeugt werden.

Bei der Definition und Deklaration von Templates arbeitet man daher mit *Typen* und weniger mit konkreten *Objekten*.

Das "Programmieren mit Typen" in Zusammenhang mit Templates, aus denen dann bei Bedarf die gewöhnlichen Funktionen bzw. Klassen *generiert* werden, kann auch als weiteres Programmierparadigma aufgefasst werden.

Dieses Paradigma wird im Allgemeinen *Generische Programmierung* genannt.

6.1 Template-Funktionen

Template-Funktionen dienen dazu, mehrere Funktionen, die sich nur durch den Datentyp der Parametern unterscheiden, aber ansonsten den gleichen Anweisungsteil besitzen, nur einmal programmieren zu müssen und dann für verschiedene Datentypen verwenden zu können.

6.1.1 Deklaration, Definition und Verwendung von Template-Funktionen

Als einfaches Beispiel für eine Template-Funktion soll hier die Maximumsbestimmung von zwei Größen eines beliebigen Types (der Name des Types sei `T`) beschrieben

werden.

Die Template-Funktion wird wie folgt deklariert:

```
template <class T>
const T& max ( const T&, const T&);
```

Hierdurch ist für jeden beliebigen Typen `T` eine Funktion mit Namen `max`, zwei `const T`-Referenzen als Parametern und einer `const T`-Referenz als Funktionsergebnis deklariert.

Der durch die Zeile

```
template <class T>
```

eingeführte Name `T` tritt hier als *Platzhalter für einen beliebigen Typen* auf und dieser Name gilt nur für die nachfolgende Deklaration (bzw. Definition, s.u.).

Obwohl der Name `T` mit `class T` eingeführt wird, steht `T` für jeden beliebigen Typen — nicht unbedingt für eine Klasse (`class`).

Neuerdings kann man hier anstelle des Schlüsselwortes `class` das neue Schlüsselwort `typename` verwenden, die entsprechende Deklaration der Template-Funktion würde dann wie folgt lauten:

```
template <typename T>
const T& max ( const T&, const T&);
```

Eine Template-Funktion muss genau einmal definiert werden. Die Definition könnte wie folgt aussehen:

```
template <class T>
inline const T& max ( const T& a, const T& b)
{
    return ( a > b ? a : b);
}
```

Auch hier wird der Name `T` (es muss nicht unbedingt der gleiche Name wie bei der Deklaration sein!) durch `template <class T>` (`template <typename T>` wäre gleichwertig) als Platzhalter für einen beliebigen Typen eingeführt und dieser Name gilt nur für die folgende Funktionsdefinition. (Die Template-Funktion ist hier, weil sie so klein ist, als `inline` definiert — Template-Funktionen müssen aber nicht zwangsläufig `inline` sein!)

Hier wird also für jeden Typen `T` die Funktion mit Namen `max`, zwei `const T`-Referenzen als Parametern und einer `const T`-Referenz als Ergebnis definiert.

Im Anweisungsteil werden die beiden Parameter mittels `>` verglichen und je nach Ergebnis dieses Vergleichs der eine oder andere Parameter als Funktionsergebnis zurückgegeben!

Mittels dieser Definition der Template-Funktion erzeugt das System, bei entsprechendem Aufruf, die zu den Argumenten des Aufrufs “passende” Realisierung der Funktion:


```

int i, j;
long k, l;
double x, y;
Bruch a, b;
...
max( i, j); // zwei int-Argumente: System erzeugt aus dem Template
            // die Funktion const int& max( const int &, const int &);
...
max( k, l); // zwei long-Argumente: System erzeugt aus dem Template
            // die Funktion const long& max( const long&, const long&);
...
max( x, y); // zwei double-Argumente: System erzeugt aus dem Template
            // die Funktion const double& max( const double&, const double&);
...
max( a, b); // zwei Bruch-Argumente: System erzeugt aus dem Template
            // die Funktion const Bruch& max( const Bruch&, const Bruch&);
...

```

Diese “Erzeugung” der konkreten Funktionen aus dem Template heißt *Instanziierung*. Bei der Instanziierung von Template-Funktionen kann es zu einer neuen, von Compilern jedoch entdeckten Art von Fehlern kommen: obiges Template kann nur für solche Typen *T* instanziiert werden, für welche der Vergleichsoperator *>* definiert ist, in obigem Beispiel wird also davon ausgegangen, das für Objekte *a* und *b* der Klasse *Bruch* der Vergleich mit *a > b* möglich ist. Ist dies nicht der Fall, meldet der Compiler bei der Instanziierung einen Fehler — die Definition des Templates an sich ist jedoch fehlerfrei!

Eine Instanziierung einer Template-Funktion für einen konkreten Typ wird pro Übersetzungseinheit nur einmal vorgenommen, d.h. wird etwa in obigem Beispiel später (im gleichen Quelltext) nochmals die Funktion *max* für zwei *int*’s aufgerufen, so wird die beim vorherigen Aufruf *max(i, j)* erzeugte *int*-Instanziierung *const int& max(const int &, const int &);* des Templates wiederverwendet. Wird in einer anderen, zum gleichen Projekt gehörenden Übersetzungseinheit ebenfalls die *int*-Version des Templates instanziiert, so sorgt das System dafür (bzw. sollte dafür sorgen), dass sich die beiden Instanziierungen nicht stören.

Hierbei wird allerdings vom Standard vorgeschrieben, dass die im zweiten Quelltext verwendete Template-Funktion Token für Token mit gleicher Bedeutung gleich definiert ist (*ODR, One Definition Rule, Eine-Definitions-Regel!*).

6.1.2 Typumwandlungen und Template-Funktionen

Bei Template-Instanziierungen wird keine Typangleichung vorgenommen, d.h. in obigem Beispiel des Templates

```

template <class T>
const T& max( const T&, const T&);

```

wird der “Aufruf“

```
int i;
double x;
...
max(i,x);    // FEHLER!
...
```

vom Compiler als Fehler gemeldet. Der Compiler wandelt das `int` also nicht in ein `double` um, um das Template instanziiieren zu können!

6.1.3 Explizite Instanziierung

Neben der impliziten Instanziierung einer Template-Funktion, bei der — wie gesehen — keine Typumwandlung durchgeführt wird:

```
template <class T>
const T& max( const T&, const T&);
...
int i,j;
int x,y;
...
max(i,j);    // implizite Instanziierung, erzeugt wird:
              // const int& max( const int &, const int &);
max(x,y);    // implizite Instanziierung, erzeugt wird:
              // const double& max( const double &, const double &);
max(x,i);    // FEHLER!
...
```

kann man eine Template-Funktion explizit instanziiieren:

```
template <class T>
const T& max( const T&, const T&);
...
int i,j;
int x,y;
...
max<int>(i,j);    // explizite Instanziierung, erzeugt wird:
                  // const int& max( const int &, const int &);
max<double>(x,y); // explizite Instanziierung, erzeugt wird:
                  // const double& max( const double &, const double &);
max<double>(x,i); // KEIN Fehler!, ggf. erzeugt und mit Typumwandlung
// verwendet wird: const double& max( const double &, const double &);
max(x,i);        // immer noch FEHLER!
max<double>(j,i); // KEIN Fehler!, ggf. erzeugt und mit Typumwandlung
// verwendet wird: const double& max( const double &, const double &);
...
```

Die explizite Instanziierung erfolgt durch Angabe des gewünschten Types in spitzen Klammern hinter dem Namen, vor der Argumentliste der Funktion, wobei jetzt

bei diesem einen Aufruf ggf. eine automatische Typumwandlungen vorgenommen werden!

Eine explizite Instanziierung ist ebenfalls dann erforderlich, wenn der Typ `T` gar nicht in der Parameterliste der Funktion, sondern “nur“ irgendwo im Anweisungsteil vorkommt:

```
template <class T>
void f(void)
{
    T tmp;
    ...
}

// Instanziierung
...
f<int>();      // T ist int
f<double>();   // T ist double
...
```

6.1.4 Template-Parameter

Bei einer Template-Deklaration bzw. -Definition

```
template <class T>
...
```

heißt dieses `T` ein *Template-Parameter*, hier ist es ein sogenannter *Typparameter* (wegen des voranstehenden `class` bzw. `typename` — bei der Instanziierung wird der Typparameter durch einen beliebigen konkreten Typen — das sog. *Template-Argument* — ersetzt).

```
template <class T>          // T ist Template-Parameter
cont T& max( const T&, const T&);
...
int i, j;
...
max<double>(i,j); // Template-Argument ist (explizit): double
max(i,j);        // Template-Argument ist (implizit): int
...
```

Ein Typparameter kann bei der Template-Definition wie ein gewöhnlicher Typ behandelt werden, bei einer konkreten Instanziierung wird aus diesem Typparameter anhand des verwendeten Template-Argumentes ein gewöhnlicher Typ.

Man kann etwa in einer Template-Definition lokale “Variablen vom Typ `T`“ vereinbaren, etwa in folgendem Template `swap` zur typunabhängigen Vertauschung zweier Elemente eines Types:

```
template <class T>
void swap( T& a, T& b)
{
    T tmp;          // lokale Variable vom "Typ T"
    tmp = a;
    a = b;
    b = tmp;
}
```

Ein Template kann mehrere Parameter besitzen (auch Typparameter):

```
template <class U, class V>
bool kleiner ( U a, V b)
{
    return a < b;
}
```

dieses Template hat zwei Typparameter, dieses Template kann für alle Typ-Paare U und V instanziiert werden, für die der Vergleich mittels < definiert ist:

```
int i, j;
double x,y;
...
kleiner(i,j);    // U ist int, V ist int, Funktion
                  // bool kleiner(int,int) wird erzeugt
kleiner(x,y);    // U ist double, V ist double, Funktion
                  // bool kleiner(double,double) wird erzeugt
kleiner(i,y);    // U ist int, V ist double, Funktion
                  // bool kleiner(int,double) wird erzeugt
kleiner<long, double>(i,j); // U ist explizit long, V ist explizit
                           // double, Funktion bool kleiner(long, double) wird erzeugt
kleiner<long>(i,j); // U ist explizit long, V ist implizit int,
                   // Funktion bool kleiner(long,int) wird erzeugt
...
```

Neben Typparametern kann ein Template auch "gewöhnliche" Parameter, etwa vom Typ int besitzen:

```
template <class T, int laenge>    // T: Typparameter
void roedel(...)                // laenge: normaler Parameter
{
    T hilfsfeld[laenge];        // Parameter T und laenge werden verwendet!
    ...
}
```

(Hier sind Gleitkommatypen vom Standard explizit ausgenommen — diese dürfen nicht als Template-Parameter verwendet werden — Adressen von Gleitkommatypen sind jedoch erlaubt!)

Bei der Instanziierung eines solchen Templates mit einem “gewöhnlichem Funktionsargument” als Template-Parameter gibt es für das zugehörige Template-Argument die Einschränkung, dass der Wert des Argumentes zur Compiler-Zeit bereits feststehen muss (dies ist für konstante Ausdrücke und Adressen von globalen Objekten der Fall, nicht jedoch für String-Literale wie "hallo").

In unserem Beispiel könnte das Template nur mit einem konstanten ganzzahligen Ausdruck als Template-Argument instanziiert werden:

```
int i=128;
...
roedel<char, 128>(...);           // T ist char und laenge ist 128
roedel<char,i>(...);             // Fehler: i kein konstanter Ausdruck
...
```

Es ist ein Fehler, einen gewöhnlichen Funktionsparameter als Template-Parameter in der Template-Definition abzuändern:

```
template <class T, int laenge>    // T: Typparameter
void roedel(...)                // laenge: normaler Parameter
{
    T hilfsfeld[laenge];        // T und laenge werden verwendet!
    ...
    laenge++;                   // FEHLER!
    ...
}
```

Neben Typparameter und gewöhnlichem Funktionsparameter kann ein Template-Parameter selbst ein Template-Klassentyp sein, d.h. als entsprechendes Argument ist bei der Instanziierung der Name eines Klassentemplates anzugeben. Hierauf wird später eingegangen, siehe Seite 184.

Template-Parameter dürfen allerdings wiederum Default-Argumente haben, d.h. wie bei Default-Argumenten von Funktionen kann bei Template-Parametern von hinten beginnend den Parametern ein Default-Argument zugewiesen werden, etwa:

```
template <class T, int laenge = 128> // Default fuer laenge ist 128
void roedel1(...);
....
roedel1<char>();           // T ist char und laenge ist 128
roedel1<int,512>();        // T ist int und laenge ist 512
...
template <class T = int, int laenge = 128> // Default fuer laenge
void roedel2(...);           // ist 128 und Default fuer T ist int
...
roedel2();                 // T ist int und laenge ist 128
roedel2<char>();           // T ist char und laenge ist 128
roedel2<int,512>();        // T ist int und laenge ist 512
...
template <class T = int, int laenge >    // FEHLER: T hat Default-Wert
void roedel3(...);                     // und laenge nicht!
...
```

6.1.5 Überladen und Spezialisierung von Funktions-Templates

Es kann ein Funktionstemplate und eine gleichnamige normale Funktion geben, etwa, um die Template-Realisierung für einen konkreten Typen abzuändern:

```
// Template zum Vergleich zweier Objekte,
// fuer die der < Operator definiert ist
template <class T>
bool kleiner( T a, T b)
{
    return a < b;
}

// normale Funktion zum Vergleichen
// von Zeichenketten
bool kleiner( const char *a, const char *b)
{
    return strcmp(a,b) < 0;
}
```

Hier werden für Zeiger auf `char` nicht die Zeiger selbst, sondern deren "Inhalte" verglichen!

Wann immer eine "normale" Funktion (ggf. mit Default-Argumenten) ohne Typumwandlung oder mit trivialen Typumwandlungen aufrufbar ist, wird diese normale Funktion einer ggf. ebenfalls möglichen Template-Instanziierung vorgezogen:

```
int i, j;
char w[10], v[10];
...
kleiner(i,j);    // Aufruf der Template-Instanziierung
                // bool kleiner(int, int)
kleiner(v,w);    // Aufruf der "normalen" Funktion
                // bool kleiner(const char *, const char *)
...
```

Man kann eine Template-Funktion mit einer anderen, gleichnamigen Template-Funktion überladen, etwa:

```
template <class T>
const T& max( const T&, const T&);           // Maximum von 2 T's
template <class T>
const T& max( const T&, const T&, const T&); // Maximum von 3 T's
```

Aus dem konkreten Aufruf sollte dem Compiler hier anhand der Anzahl der Argumente klar werden, welche Funktion er zu nehmen hat.

Eine Überladung folgender Form:

```
template <class T>
void fkt(T);
template <class T>
void fkt(T*);
```

heißt *Spezialisierung*, die erste Version ist für jeden Typ anwendbar, die zweite Version jedoch nur für Zeigertypen.

Genauer: die zweite Version heißt Spezialisierung der ersten Version, denn jeder Aufruf, der durch Instanziierung der zweiten Version umgesetzt werden könnte, könnte auch durch Instanziierung der ersten Version realisiert werden — aber nicht umgekehrt!

Bei Templates mit mehreren Parametern kann es auch *partielle Spezialisierung* geben:

```
template <class U, class V>
void fkt( U, V);    // Version fuer beliebige Typen
template <class U>
void fkt(U, int);   // Version, falls 2-tes Argument ein int ist
```

Hier ist wiederum die zweite Version spezieller als die erste, denn jeder Aufruf, der durch Instanziierung der zweiten Version umgesetzt werden könnte, könnte auch durch Instanziierung der ersten Version realisiert werden — aber nicht umgekehrt!

Wann immer für einen Funktions-Aufruf zwei Template-Funktionen aufgerufen werden könnten, von denen die eine spezieller ist als die andere, wird die speziellere Template-Funktion instanziiert!

Zu einem Template kann man eine derartige Spezialisierung schreiben, die keinen Template-Parameter mehr benötigt (alle Template-Parameter sind “wegspezialisiert”). Hierzu muss man leere spitze Klammern hinter dem Schlüsselwort `template` schreiben:

```
template <>
void fkt(double, int);
```

(Ohne dieses `template<>` wäre die Funktion eine “normale” Funktion und keine Template-Funktion!)

6.1.6 Regeln zum Auffinden der “richtigen” Funktion

Die Regeln, nach denen der Compiler nach der richtigen Funktion für einen Funktionsaufruf sucht, seien hier kurz zusammengestellt:

1. findet der Compiler im aktuellen Gültigkeitsbereich die Deklaration einer normalen Funktion, die er (abgesehen von den trivialen Umwandlungen *Typ* nach *const Typ*, *Feldname* nach *Adresse des Feldanfang* oder *Funktionsname* nach *Adresse der Funktion*) ohne Typumwandlung aufrufen kann, so ruft er diese auf,
2. findet der Compiler ein oder unterschiedlich spezialisierte Templates, durch das/die er den Aufruf umsetzen könnte, so nimmt er dieses Template bzw. das speziellste der Templates,

3. falls bislang der Funktionsaufruf noch nicht umgesetzt werden konnte, versucht der Compiler passende “normale” Funktionen mit Umwandlungen (Standard-Umwandlung, Benutzerdefinierte Umwandlung, . . . -Umwandlungen) zu finden.

Findet er keine passende Funktion, so ist dies ein Fehler.

Findet er mehrere gleich gut passende Funktionen, so ist dies ebenfalls ein Fehler.

6.2 Template-Klassen

Template-Klassen (oder auch Klassen-Templates genannt) werden häufig dann verwendet, wenn eine Klasse Objekte eines anderen Types (ggf. auch Klasse) verwalten sollen und es primär nicht auf den “verwalteten” Typ ankommt.

So haben wir bereits die Klasse *Kellerspeicher für int-Objekte* kennengelernt, genauso kann man einen *Kellerspeicher für double-Objekte* oder einen *Kellerspeicher für Bruch-Objekte* etc. definieren.

Das primäre an all diesen Definitionen wäre das Kellerspeicherprinzip, sekundär ist der Typ der im Kellerspeicher abgespeicherten Elemente.

6.2.1 Deklaration, Definition und Verwendung von Template-Klassen

Als Beispiel wollen wir ein Template für Kellerspeicher deklarieren und definieren:

- Deklaration:

```
template <class T>
class Stack {
    private:
        T feld[100];
        int sp;
    public:
        Stack();    // Konstruktor
        void push(T&); // Einkellern eines Elementes
        T pop();    // Auskellern eines Elementes
};
```

Durch `template <class T>` wird wiederum der Name `T` als Typparameter des Templates eingeführt (`template <typename T>` wäre gleichwertig gewesen)! Dieser Name `T` wird in der ganzen Deklaration (Klassenrumpf) wie ein Typname verwendet: es wird ein Feld von diesem “Typ” angelegt und Member-Funktionen mit Parameter oder Ergebnis dieses “Types” vereinbart.

- Definition:

Die Member-Funktionen dieses Klassen-Templates müssen noch definiert werden (hätte auch im Klassenrumpf, anstelle der Deklaration geschehen können!).

Wie bei Klassen üblich, muss auch bei Template-Klassen bei der Definition der Member-Funktionen auf die Klasse Bezug genommen werden. Da aber die Template-Klasse noch vom Parameter `T` abhängt, lautet der Klassenname jetzt `Stack<T>`. Da hier wieder der Name `T` auftaucht, muss dieser wiederum vorher mittels `template<class T>` eingeführt werden:

```
template <class T>
Stack<T>::Stack()
{ sp = 0; }

template <class T>
void Stack<T>::push( T& elem)
{
    if ( sp < 100 )
        feld[sp++] = elem;
    else
    {
        cerr << "Stack voll!" << endl;
        exit(1);
    }
}

template <class T>
T Stack<T>::pop()
{
    if ( sp > 0)
        return feld[--sp];
    else
    {
        cerr << "Stack empty!" << endl;
        exit(1);
    }
}
```

Jede der Member-Funktionen hängt vom Template-Parameter `T` ab und da die Bekanntmachung dieses Namens durch `template <class T>` immer nur für die nächste Definition bzw. Deklaration gilt, muss bei der Definition jeder der Funktionen der Name `T` erneut durch `template <class T>` bekannt gemacht werden.

Ansonsten stellen die Implementierung von `push` und `pop` eine einfache (sicherlich noch ausbaufähige) Standard-Realisierung der üblichen `Stack`-Funktionen dar.

Mit dieser Template-Klasse können nun für unterschiedliche konkrete Typen “wirkliche” Klassen “erzeugt” werden. Da im Gegensatz zu Template-Funktionen bei einer Template-Klasse die Template-Argumente nicht implizit erkannt werden können, müssen Klassen-Templates im Allgemeinen immer explizit instanziiert werden:

```
// als Objekte
Stack<int> intStack;           // Stack fuer int's
Stack<float*> floatPtrStack;   // Stack fuer float-Zeiger
Stack<Bruch> bruchStack;      // Stack fuer Brueche
...
// als Funktionsparameter und lokale Variable
void f( Stack<int> s)          // Parameter: int-Stack
{
    Stack<int> istack[10];     // lokales Feld von int-Stacks
    ...
}
```

Natürlich kann man mittels `typedef` einem häufig verwendeten Stack-Typ einen eigenen Namen geben:

```
typedef Stack<int> IntStack;   // Name fuer den Typen: Stack von int's
void f( IntStack s)           // Parameter: int-Stack
{
    IntStack istack[10];      // lokales Feld von int-Stacks
    ...
}
```

Member-Funktionen eines Klassen-Templates sind formal eigenständige Templates (erkennbar an dem eigenen `template <class T>`) und diese werden für konkrete Typen erst dann instanziiert, wenn die entsprechende Member-Funktion wirklich benötigt wird!

Die Instanziierung eines Klassen-Templates und der zugehörigen Member-Funktionen geht natürlich nur für solche Typen `T`, für die alle in der Implementierung verwendeten Operationen (Operatoren und Funktionen) verfügbar sind — hier im Beispiel muss für den Typen `T` zumindest der Zuweisungsoperator verfügbar sein!

6.2.2 Template-Parameter

Wie schon bei Funktions-Templates kann auch ein Klassentemplate mehrere, in den spitzen Klammern bei `template` anzugebende Template-Parameter haben, wie etwa folgendes, in der Standard-Template-Library definierte Klassentemplate zur Bildung von "Paaren" von Objekten beliebiger Typen:

```
template <class T1, class T2>
class pair {
public:
    T1 first;
    T2 second;
    ...
};
```

Mit diesem Template könnte jetzt der Typ `pair<int,double*>`, Paar aus einem `int` und einem `double`-Zeiger, vereinbart werden und man könnte jetzt auch einen Kellerspeicher zur Abspeicherung solcher Paare definieren:

```
Stack<pair<int, double*>> PaarStack;
```

Natürlich können auch wieder neben solchen Typparametern auch gewöhnliche Parameter als Template-Parameter verwendet werden, etwa:

```
template <class T, int laenge>
class Stack {
private:
    T feld[laenge];
    int sp;
public:
    ...
};

// Instanziierung:
Stack<int,50> istack1, istack2; // int-Stack's der Laenge 50
Stack<int,10> istack3;         // int-Stack der Laenge 10
Stack<char,1000> cstack;       // char-Stack der Laenge 1000
...
```

Hier gelten die gleichen Einschränkungen wie bei Funktions-Templates:

der gewöhnliche Parameter darf kein Gleitkommatyp sein und das entsprechende Argument bei der Instanziierung muss im Wesentlichen konstant sein!

Zu beachten ist, dass Template-Instanzierungen mit unterschiedlichen Template-Argumenten unterschiedliche Typen sind! So haben in obigem Beispiel *istack1* und *istack2* den gleichen Typen *int-Stack der Länge 50*, *istack3* hat aber den anderen Typ *int-Stack der Länge 10*.

So wäre formal die Zuweisung *istack1 = istack2;* erlaubt (gleiche Typen) — die Zuweisung *istack1 = istack3;* aber nicht (unterschiedliche Typen).

Wie bei Template-Funktionen kann ein Template-Parameter selbst wieder ein Template-Typ sein.

Seien beispielsweise folgende zwei Templates zur Speicherung von Daten eines beliebigen Types *T* gegeben:

```
template <class T> class Stack { /* ... */ }; // Kellerspeicher
template <class T> class LListe { /* ... */ }; // Lineare Liste
```

und es soll eine neue Template-Klasse (der Name sei *A*) definiert werden, in der (u.a.) auch Daten von einem Typ *T* abgespeichert werden sollen, wobei die Art der Abspeicherung offen ist, möglicherweise als *Stack* oder aber auch als *LListe*. Die entsprechende Template-Definition könnte wie folgt aussehen:

```
template <class T, template <class U> class SPEICHER>
class A {
    SPEICHER<T> speicher;
    ...
};
```

Neben dem Typparameter `class T` taucht hier das Template

```
template <class U> SPEICHER
```

als Parameter auf. Der Name `SPEICHER` ist jetzt als Platzhalter für eine bei der Instanziierung anzugebende Template-Klasse zu verwenden.

Instanziiert könnte dieses Template etwa wie folgt:

```
A<int, Stack> a1;      // es werden int's abgespeichert, Abspeicherung
                      // erfolgt in einem Stack
A<char *, Lliste> a2;  // es werden char--Zeiger abgespeichert,
                      // Abspeicherung erfolgt in einer LLISTE
...
```

Solche Templates als Parameter eines Templates sind auch bei Template-Funktionen möglich:

```
template <class T, template class<U> class SPEICHER>
void fkt (...)
{
    SPEICHER<T*> speicher;
    ...
}
```

Auch bei Klassen-Templates können von hinten beginnend die letzten Template-Parameter wiederum Default-Werte besitzen:

```
template<class T = int, int i = 10>
class Stack { /* ... */ };
...
Stack<double, 100> a;    // double-Stack der Laenge 100
Stack<char *>          b; // char*-Stack der Laenge 10
Stack<>                 c; // int-Stack der Laenge 10,
                        // leere spitze Klammern erforderlich!
...
```

6.2.3 Spezialisierung von Klassen-Templates

Natürlich können auch Klassen-Templates spezialisiert werden, etwa:

```
// Vektorklasse fuer allgemeine Typen:
template <class T>
class Vektor {
{
    ... // Implementierung fuer allgemeine Typen
};
...
// Vektorklasse fuer Zeigertypen:
template <class T>
```

```

class Vektor<T*> {
{
...    // spezielle Implementierung fuer Zeigertypen
};
...
// Vektorklasse fuer void *:
template <>
class Vektor<void*>
{
...    // spezielle Implementierung fuer void *
};
...

```

Das Template für Zeigertypen ist spezieller als das für allgemeine Typen und das für `void *` ist wiederum spezieller als das für Zeigertypen.

Da bei der Deklaration der spezielleren Templates auf das allgemeine Template Bezug genommen wird (`class Vektor<T*>` bzw. `class Vektor<void *>`) muss das allgemeine Template vor den spezielleren deklariert werden!

Sind bei einer Instanziierung formal mehrere Template-Klassen möglich, wird vom System immer die speziellste Template-Klasse genommen.

6.2.4 Element-Templates

Template-Klassen können (übrigens gewöhnliche Klassen auch!) Komponenten (Member-Funktionen oder oder Member-Daten) haben, die selbst wiederum Templates (mit anderen Template-Parametern) sind.

Als Beispiel wollen wir eine Template-Klasse zur Darstellung komplexer Zahlen entwickeln, wobei der Typ von Realteil und Imaginärteil durch den Template-Typparameter festgelegt wird:

```

template <class Scalar>
class complex {
public:
    Scalar re;           // Real- und Imaginärteil
    Scalar im;           // vom Typ Scalar
    // Standard-Konstruktor
    complex() : re(0), im(0) {};           // komplett definiert
    // Konstruktor aus zwei Skalaren, komplett definiert
    complex(Scalar a, Scalar b) : re(a), im(b) { };

    Scalar operator+(Scalar) const; // Addition, nur deklariert
    Scalar operator*(Scalar) const; // Multiplikation, nur deklariert
    ...
};

```

Hiermit ist es jetzt möglich, komplexe Zahlen mit unterschiedlichem Skalartyp zu definieren:

```

complex<int>    ci;    // Skalartyp ist int
complex<double> cd;    // Skalartyp ist double
complex<float> cf;    // Skalartyp ist float
...

```

Zusätzlich soll jetzt möglich werden, eine komplexe Zahl anhand einer anderen komplexen Zahl zu initialisieren, wobei etwa eine komplexe Zahl mit Skalartyp `double` auch mit einer komplexen Zahl mit Skalartyp `int` initialisierbar sein soll. Dies kann durch folgenden Konstruktor, der selbst wiederum ein Template ist, erfolgen:

```

template <class Scalar>
class complex {
public:
    Scalar re;          // Real- und Imaginarteil
    Scalar im;          // vom Typ Scalar
    // Standard-Konstruktor, komplett definiert
    complex() : re(0), im(0) {};
    // Konstruktor aus zwei Skalaren, komplett definiert
    complex(Scalar a, Scalar b) : re(a), im(b) { };

    // Template-Konstruktor:
    template <class U>
    complex( const complex<U> &c)    // mit complex<U> c initialisiert
        : re( c.re), im(c.im) { }    // komplett definiert

    Scalar operator+(Scalar) const; // Addition, nur deklariert
    Scalar operator*(Scalar) const; // Multiplikation, nur deklariert
    ...
};

```

Mit diesem Konstruktor kann eine komplexe Zahl mit Skalartyp *Typ1* aus einer anderen komplexen Zahl mit Skalartyp *Typ2* initialisiert werden, wenn eine Typumwandlung von *Typ2* nach *Typ1* möglich ist, denn diese wird in der Initialisierungsliste `: re(c.re), im(c.im)` angewendet:

```

...
complex<int> i(0,1);
complex<double> b(i);    // initialisiere complex<double> b
...                      // mit complex<int> i

```

Merkwürdigerweise wird dieser Template-Konstruktor nicht zur Erzeugung des Copy-Konstruktors verwendet, der hätte den Typ: `complex(const complex<Scalar> &c);` (hier wären also speziell die Typen `Scalar` und `T` gleich) — sondern es wird vom System der (hier ausreichende) Standard-Copy-Konstruktor generiert!

Dieser Template-Konstruktor ist hier gleich innerhalb des Klassenrumpfes der Template-Klasse komplett definiert. Sollte er außerhalb definiert werden, so müsste dies wie folgt geschehen:

```

template <class Scalar>
class complex {
public:
    Scalar re;           // Real- und Imaginarteil
    Scalar im;           // vom Typ Scalar
    // Standard-Konstruktor, komplett definiert
    complex() : re(0), im(0) {};
    // Konstruktor aus zwei Skalaren, komplett definiert
    complex(Scalar a, Scalar b) : re(a), im(b) { };

    // Template-Konstruktor:
    template <class U>
    complex( const complex<U> &c); // mit complex<U> c initialisiert,
                                   // nur Deklaration

    Scalar operator+(Scalar) const; // Addition, nur deklariert
    Scalar operator*(Scalar) const; // Multiplikation, nur deklariert
    ...
};
...
// Definition des Konstruktors eines complex<Scalar>
// aus einem complex<U>:
template <class Scalar>
    template <class U>
        complex<Scalar>::complex( complex<U> &c) : re(c.re), im(c.im)
        { }
...

```

Hier dürfen die beiden `template <class Scalar>` und `template <class U>` nicht zu einem `template <class Scalar, class U>` zusammengefasst werden!

6.2.5 Templates und Vererbung

Es können Vererbungsbeziehungen zwischen Template-Klassen und gewöhnlichen Klassen und auch zwischen Template-Klassen und anderen Template-Klassen bestehen. Hierauf wird im Anschluss an die Behandlung von Vererbung eingegangen.

6.3 Implementierungsmöglichkeiten

Es gibt zwei Strategien zur Verwendung von Templates (Funktionen und Klassen):

1. Deklaration und Definition des Templates in jedem Quelltext, wo das Template verwendet wird.

Hierdurch wird das Template ggf. in unterschiedlichen Übersetzungseinheiten gleichartig instanziiert, aber diese gleichartigen Instanzierungen werden vom Linker nicht als störend empfunden (doppelte oder mehrfache Instanzierungen werden von guten Linkern weitgehend eliminiert!).

Hierbei ist aber die *ODE* (*One Definition Rule*) zu beachten: die Template-Definitionen müssen in allen Quelltexten syntaktisch und semantisch gleich sein! Es ist ein Fehler, ein- und dieselbe Template-Definition mehrfach in einem Quelltext aufzuführen.

Eine sichere Methode ist daher, die Template-Deklaration und Definition in eine Headerdatei zu schreiben, welche durch `#ifndef...` vor mehrmaligem Einbinden in eine Übersetzungseinheit geschützt ist, und diese Headerdatei in allen Quelltexten, wo das Template verwendet wird, einzubinden. Nachteilig ist die ggf. mehrfache gleichartige Instanziierung desselben Templates in mehreren Quelltexten durch den Compiler und deren anschließender Eliminierung durch den Linker.

2. Deklaration des Templates in einer Headerdatei, diese Headerdatei in allen Übersetzungseinheiten einzubinden, wo das Template verwendet wird, und Definition in einer separaten Implementierungsdatei und die Übersetzung dieser Implementierungsdatei beim Linken angeben.

Auf unseren Systemen ist es (zur Zeit) allerdings erforderlich, gleich in der Implementierungsdatei alle in der Anwendung benötigten Instanziierungen mit zu erzeugen — in einer anderen Übersetzungseinheit kann keine neue Instanz des Templates generiert werden.

Der Standard sieht zwar das Schlüsselwort `export` für die Definition des Templates vor:

```
export template <class T>
void out ( const T& t) { /* ... */ }
...
```

so dass auf diese Art in einer Übersetzungseinheit definierte Templates auch in anderen Übersetzungseinheiten (durch den Linker?) instanziiert werden können — doch dieses Schlüsselwort wird durch unsere C++-Systeme noch nicht unterstützt!

Kapitel 7

Vererbung

7.1 Grundlagen

Klassen sollte man so konzipieren, dass sie später, etwa in einer weiteren Anwendung, wiederverwendet werden können (Headerdatei mit Klassenrumpf und übersetzte Implementierung zur Verfügung stellen — Headerdatei includen und Implementierung der neuen Anwendung hinzulinken!).

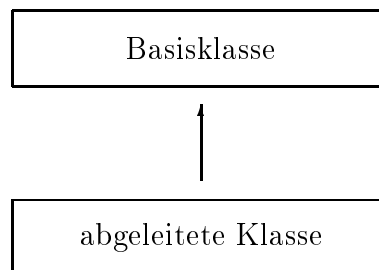
Hin- und wieder ist es erforderlich, die Definition der Klasse leicht zu modifizieren bzw. zu ergänzen, um die Klasse den neuen Anforderungen anzupassen. Häufig ist es dann so, dass einige wenige neue Datenkomponenten und Methoden hinzukommen und einige wenige, bereits vorhandene Funktionskomponenten abgeändert werden müssen.

Hierzu braucht man keine völlig neue Klasse zu definieren und implementieren, man verwendet die bereits vorhandene (hoffentlich gut ausgetestete und bewährte) Klasse und definiert das neu, was in der (neuen) Anwendung angepasst werden muss.

Auf diese Weise entsteht aus der “alten“, vorhandenen Klasse eine neue Klasse — es stehen jetzt also zwei Klassen (mit leicht unterschiedlicher Definition) zur Verfügung. Diesen Vorgang nennt man in der OOP (*einfache*) *Vererbung* (engl.: *(simple) Inheritance*).

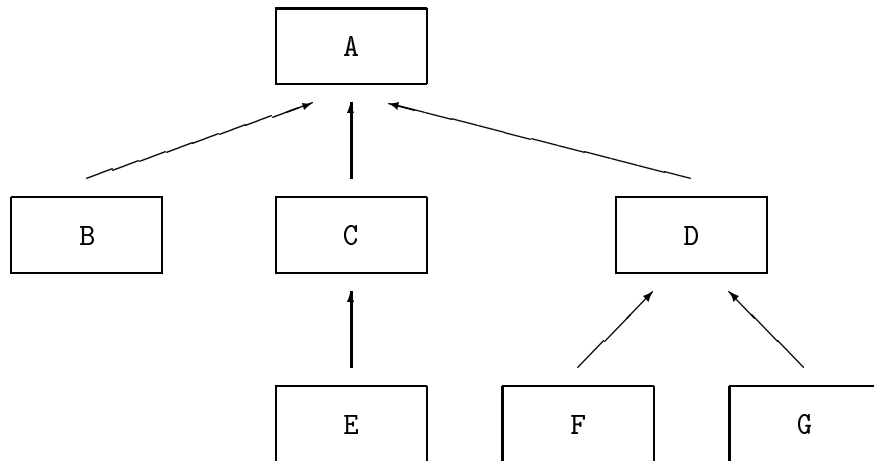
Die “alte“, bereits vorhandene Klasse heißt *Basisklasse* und die neue Klasse heißt *abgeleitete Klasse*.

In Schaubildern wird das häufig wie folgt dargestellt:



Ein Pfeil in derartigen Schaubildern bedeutet: die Klasse, von der der Pfeil ausgeht, ist abgeleitet von der Klasse, auf welche der Pfeil zeigt. (Im Allgemeinen stehen die Basisklassen weiter oben als die abgeleiteten Klassen!)

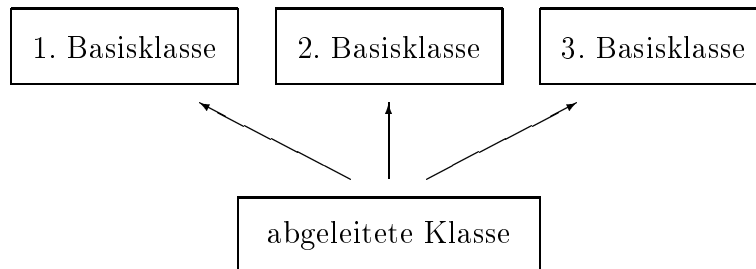
Von abgeleiteten Klassen kann man natürlich wiederum weitere Klassen ableiten, als “Klassenhierarchie” erhält man im Schaubild einen *Baum* als (gerichteten) Graphen:



In einer Klassenhierarchie kann eine abgeleitete Klasse *direkt* von einer Basisklasse abgeleitet sein (im Bild ist etwa die Klasse D direkt von A abgeleitet und F direkt von D) — die Basisklasse heißt dann auch *direkte Basisklasse* der abgeleiteten Klasse.

Indirekt ist etwa die Klasse F von der Klasse A abgeleitet (A ist *indirekte Basisklasse* der Klasse F).

Eine neue Klasse kann auch (gleichzeitig) aus mehreren “alten” Klassen abgeleitet werden. Diese Art der Vererbung heißt *Mehrfachvererbung* (engl.: *Multiple Inheritance*), Schaubild:



Mittels einfacher und mehrfacher Vererbung sind beliebige, zyklfreie gerichtete Graphen als “Klassenhierarchien” möglich.

Einfache Vererbung wird von allen Objektorientierten Sprachen unterstützt — Mehrfachvererbung nur von einigen, u.a. auch von C++.

7.1.1 Einfache Vererbung

Ist A eine bestehende Klasse

```

class A {
...
... // A-Komponenten
...
};
  
```

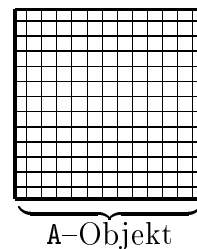
so kann man in C++ aus der Klasse A wie folgt eine neue Klasse B (öffentlich) ableiten:

```
class B : public A {
...
... // neue B-Komponenten
...
};
```

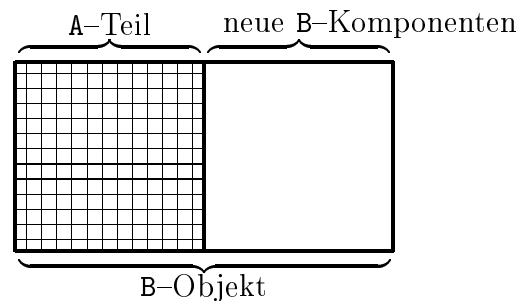
Ein Objekt der Klasse B hat alle Komponenten (Daten und Funktionen), welche auch ein A-Objekt hat (bereits im Klassenrumpf von A deklariert, oben mit A-Komponenten bezeichnet) — darüberhinaus kann ein B-Objekt weitere Komponenten haben, welche ein A-Objekt nicht hat (im Klassenrumpf von B deklariert, oben mit **neue B-Komponenten** bezeichnet).

Ein B-Objekt hat also einen “A-Teil“, darüberhinaus aber noch einen “neuen“ Teil, den ein A-Objekt nicht hat:

```
class A {
...
... // A-Komponenten
...
};
```



```
class B : public A {
...
... // neue B-Komponenten
...
};
```



7.2 Zugriffsschutz

Ist eine Klasse B von einer Klasse A abgeleitet:

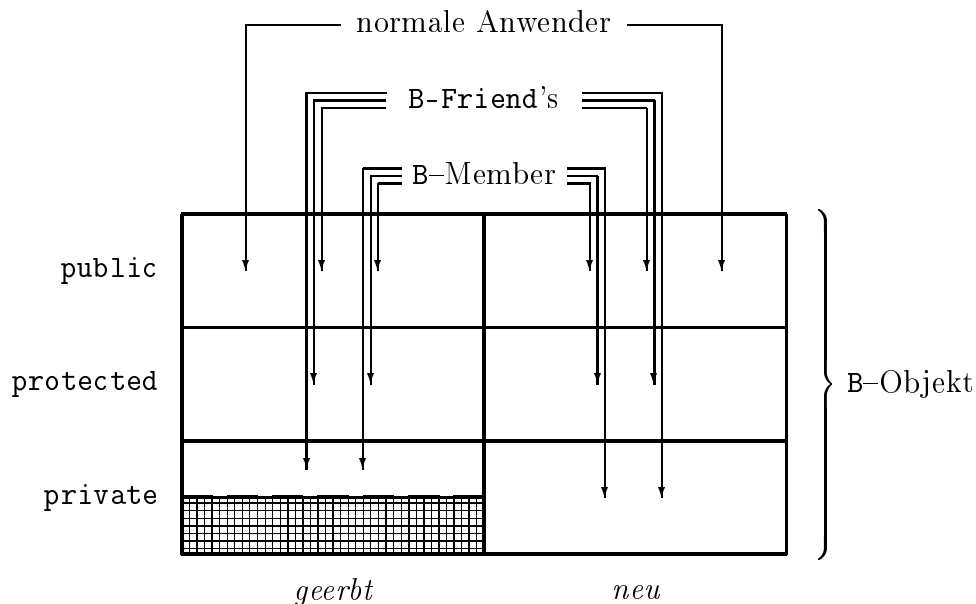
```
class A { ... };

class B : public A {
...
};
```

so hat ein B-Objekt alle Komponenten (Daten und Funktionen), welche auch ein A-Objekt hat (*geerbte* Komponenten) und ggf. weitere neue Komponenten — man kann (muss) also bei einem B-Objekt zwischen *geerbten* und *neuen* (normalen) Komponenten (Funktionen und Daten) unterscheiden.

Sowohl *geerbte* aus auch *neue* (normale) Komponenten stehen in einem Zugriffsabschnitt, man kann (muss) somit auch bei den Zugriffsabschnitten **private**, **protected** und **public** jeweils zwischen dem *geerbten* und dem *neuen* (normalen) entsprechenden Zugriffsabschnitt unterscheiden!

Aus Sicht der Klasse B ist der Zugriff (für B-Member-Funktionen, zu B befreundete Funktionen und normale Anwender der Klasse B) auf die entsprechenden (*geerbten* und *neuen*) Komponenten eines B-Objektes in folgender Skizze festgehalten (Pfeil bedeutet: Zugriff möglich!):



Bemerkenswert ist, dass auf einen Teil der *geerbten*, **private** Komponenten (im Bild schraffiert eingezeichnet) “aus B-Sicht“ gar keine Funktionen mehr zugreifen können (weder normale Funktionen, noch B-Friends, noch B-Member-Funktionen). Erst wenn ein B-Objekt als ein Objekt einer Basisklasse von B (etwa die Klasse A) “auftritt“ — und zwar in einer A-Member-Funktion oder in einer zur Klasse A befreundeten Funktion —, kann in dieser Funktion auf die von A geerbten, privaten Komponenten des B-Objektes zugegriffen werden.

Beim *geerbten*, **private** Zugriffsabschnitt muss also wiederum unterschieden werden zwischen dem Teil, auf welchen B-Member- und B-befreundete Funktionen Zugriff haben und dem Teil, auf welchen diese Funktionen keinen Zugriff mehr haben (in dieser und den folgenden Skizzen schraffiert!).

In welchem geerbten Zugriffsabschnitt eine Komponente der Klasse A im geerbten Teil der Klasse B bei der Vererbung “ankommt“, hängt vom Zugriffsabschnitt der Komponente in der Klasse A und der *Vererbungsart* ab (es ist nicht zwangsläufig so, dass eine in A öffentliche Komponente — A-Zugriffsabschnitt **public** — auch im geerbten Teil in B im **public** Zugriffsabschnitt “landet“).

Es gibt folgende drei *Vererbungsarten*:

- **public**-Vererbung,
- **protected**-Vererbung und

- `private`-Vererbung.

Diese Vererbungsarten werden in folgenden drei Unterabschnitten erläutert:

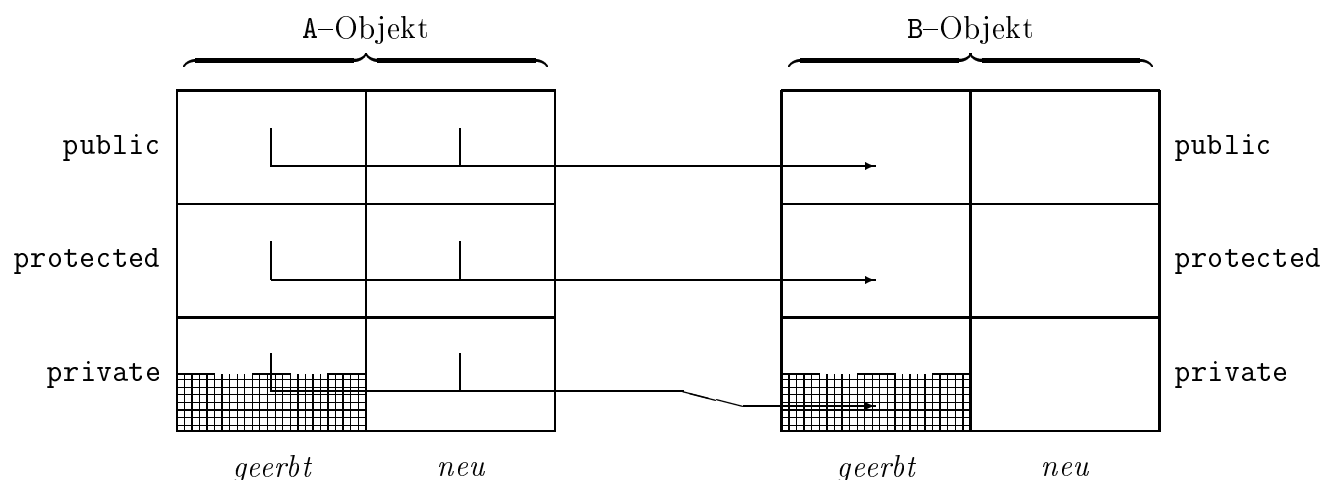
7.2.1 `public`-Vererbung

Die `public`-Vererbung (öffentliche Vererbung) ist die üblichste Vererbungsart:

```
class A { ... };

class B : public A { ... };
...
```

Wird eine Klasse `B` öffentlich von einer Klasse `A` abgeleitet (die Klasse `A` heißt dann auch *öffentliche* Basisklasse von `B`), so “landen“ die `public`-Komponenten von `A` im geerbten, `public` Zugriffsabschnitt von `B`, die die `protected`-Komponenten von `A` im geerbten, `protected` Zugriffsabschnitt von `B` und die `private`-Komponenten von `A` im geerbten, nicht zugreifbaren `private` Zugriffsabschnitt von `B`:



Ein `B`-Objekt hat zwar alle Komponenten, welche auch ein `A`-Objekt hat — auf die `private`-`A`-Komponenten (die ein `B`-Objekt auch geerbt hat) können die `B`-Funktionen (Member oder befreundete) jedoch nicht mehr zugreifen.

Diese Zuordnung:

- `public` `A`-Teil in den (für `B`-Funktionen zugänglichen) `public` `B`-Teil,
- `protected` `A`-Teil in den (für `B`-Funktionen zugänglichen) `protected` `B`-Teil und
- `private` `A`-Teil in den (für `B`-Funktionen nicht zugänglichen) `private` `B`-Teil

ist unabhängig davon, ob die `A`-Komponenten ggf. selbst — von einer weiteren Klasse, von der `A` abgeleitet wurde — geerbt sind oder nicht!

Nur bei dieser Vererbungsart gilt:

Ein `B`-Objekt ist ein `A`-Objekt.

D.h. ein B-Objekt kann auch überall dort verwendet werden, wo ein A-Objekt verwendet werden könnte (hierbei wird der in B *neue* — genauer: nicht von A geerbte — Teil “vergessen“):

```
class A {
    ...
    public:
        void A_fkt();
    ...
};

class B: public A {
    ...
    public:
        void B_fkt();
    ...
};

...
void f1(A);    // Parameter vom Typ A
void f2(A&);   // Parameter vom Typ A&
void f3(A*);   // Parameter vom Typ A*
...
B b;           // B-Objekt
A a(b);        // A-Objekt mit B-Objekt initialisieren
A *ap;         // Zeiger auf A
B *bp;         // Zeiger auf B
...
a = b;         // B-Objekt einem A-Objekt zuweisen
...
ap = &b;       // Adresse eines B-Objektes einem Zeiger auf A zuweisen
                // Zeiger auf A koennen auf B-Objekte zeigen!
...
b.A_fkt();     // A-Memberfunktion fuer B-Objekt aufrufen
...
f1(b);         // Funktion mit A-Parameter mit B-Argument aufrufen
...
f2(b);         // Funktion mit A&-Parameter mit B-Argument aufrufen
...
f3(&b);        // Funktion mit A*-Parameter mit Adresse eines
...           // B-Objektes aufrufen
```

Jedes B-Objekt ist auch ein A-Objekt, das Umgekehrte gilt jedoch nicht! So ist in obigem Umfeld (keine Typumwandlung von A nach B) folgendes nicht erlaubt:

```
b = a;         // FEHLER: kann einem B-Objekt kein A-Objekt zuweisen
...
bp = &a;       // FEHLER: Adresse eines A ist keine Adress eines B
```

```

...
ap = &b;          // OK!
ap->B_fkt();      // obwohl der A-Zeiger auf ein B-Objekt zeigt, kann
                  // kein Zugriff auf neue B-Komponente erfolgen
...
void g1(B);       // Funktion mit B-Argument
void g2(B&);      // Funktion mit B&-Argument
void g3(B*);      // Funktion mit B*-Argument
...
g1(a);           // FEHLER: a ist kein B-Objekt
...
g2(a);           // FEHLER: a ist kein B-Objekt
...
g3(&a);          // FEHLER: Adresse eines A ist keine Adresse eines B
...

```

Nur bei dieser öffentlichen Vererbung bleibt die öffentliche Schnittstelle (**public**-Teil) der Klasse A in der öffentlichen Schnittstelle der Klasse B (im **public**, geerbten Zugriffsabschnitt) erhalten — die Schnittstelle der Klasse B umfasst die komplette Schnittstelle der Klasse A, sie ist allenfalls größer!

Öffentliche Vererbung ist die Voreinstellung, wenn eine mittels **struct** definierte Klasse von einer anderen Klasse abgeleitet wird (andere Klasse kann mittels **struct** oder **class** definiert sein!) — bei **struct** kann also das Schlüsselwort **public** bei der Vererbung fortgelassen werden:

```

class A { ... };
// struct A { ... };  auch moeglich!

struct B: A { ... };
// entspricht:
// struct B: public A { ... };
...

```

7.2.2 protected-Vererbung

Bei der **protected**-Vererbung ist bei der Definition der abgeleiteten Klasse das Schlüsselwort **protected** vor der Basisklasse anzugeben:

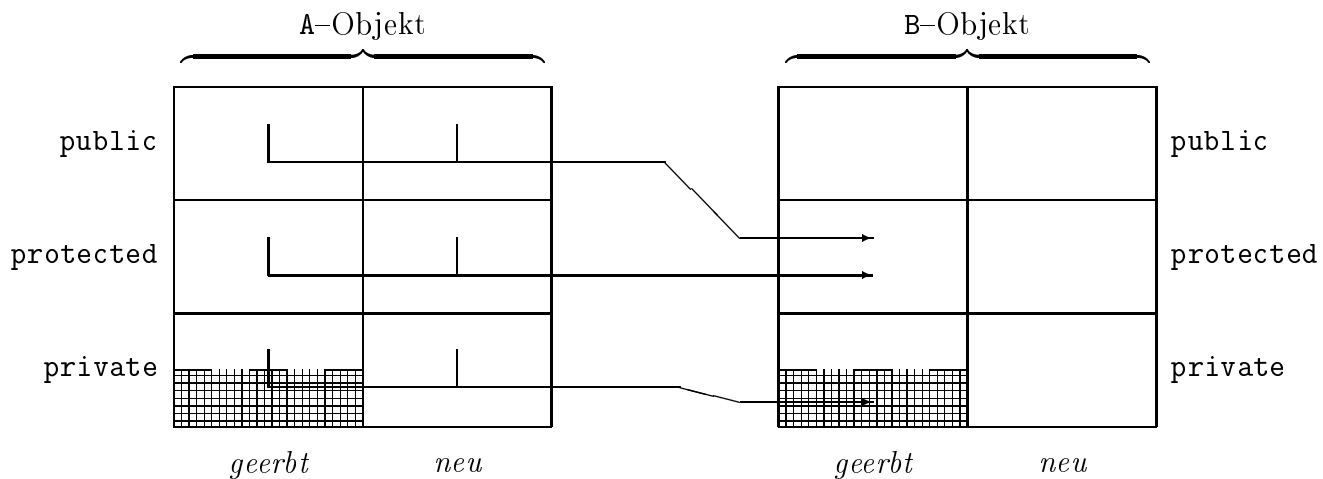
```

class A { ... };

class B : protected A { ... };
...

```

Der Unterschied der **protected**-Vererbung gegenüber der **public**-Vererbung ist der, dass die **public**-Elemente der Basisklasse im (geerbten) **protected**-Zugriffsabschnitt der abgeleiteten Klasse ankommen:



Obwohl jedes B-Objekt nach wie vor alle Komponenten hat, welche auch ein A-Objekt besitzt, ist hierdurch die A-Schnittstelle beim Übergang von der Klasse A zur Klasse B nicht mehr in der B-Schnittstelle vorhanden — entsprechend gilt bei dieser Vererbungsart:

Ein B-Objekt ist kein A-Objekt.

B-Objekte haben nur noch die “neue” Schnittstelle, die Klasse B wird mittels der Klasse A implementiert — die A-Komponenten stellen also ein *Implementierungsdetail* der Klasse B dar.

Die Klasse B ist mit Hilfe der Klasse A realisiert, die **public** und **protected** A-Komponenten sind im ererbten **protected** Zugriffsabschnitt von B und somit nur noch für B-Member- und zu B befreundete Funktionen zugreifbar.

7.2.3 private-Vererbung

Die restriktivste Vererbungsart ist die **private-Vererbung**.

Hier ist bei der Definition der abgeleiteten Klasse das Schlüsselwort **private** vor der Basisklasse anzugeben:

```
class A { ... };

class B : private A { ... };
...
```

Wird die abgeleitete Klasse (wie hier) mittels des Schlüsselwortes **class** (nicht mit **struct**) definiert, kann das Schlüsselwort **private** bei der **private-Vererbung** fortgelassen werden (unabhängig davon, ob die Basisklasse mit **class** oder **struct** definiert wurde):

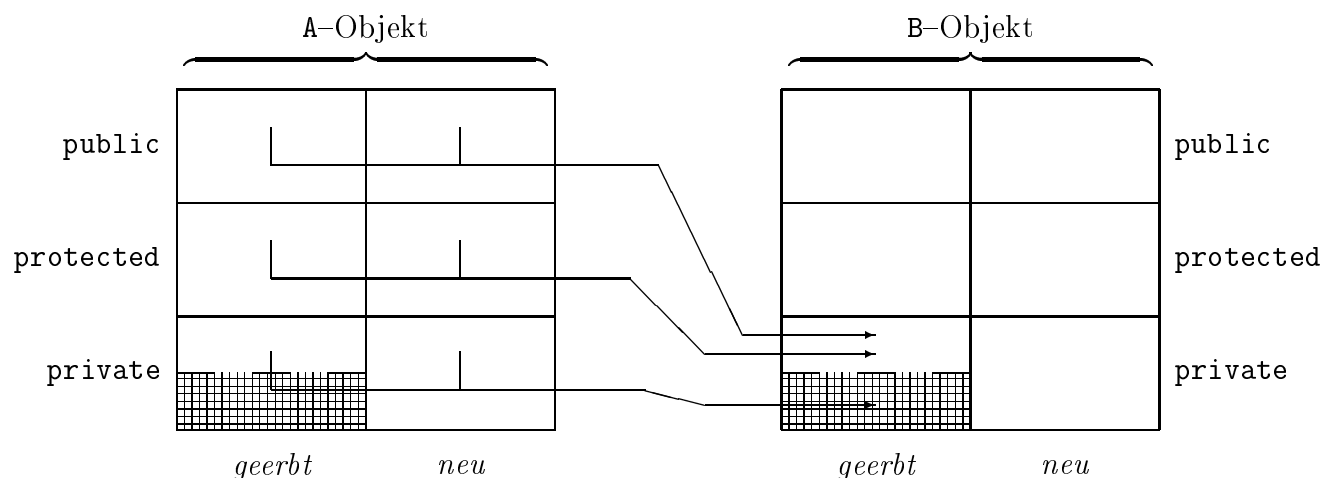
```
class A { ... };
// struct A { ... }; auch moeglich!

class B : A { ... };
// entspricht:
```



```
// class B : private A { ... };
...
```

Bei der **private**-Vererbung landen — wie bei jeder Vererbungsart — die **private** A-Komponenten im ererbten, nicht zugänglichen **private**-Teil von B — die **protected** und **public** A-Komponenten “landen” jedoch auch im ererbten **private**-Teil von B — und zwar im (für B-Member-Funktionen und zu B befreundete Funktionen) zugänglichen Teil:



d.h. wie bei der **protected** Vererbung sind diese nur noch für B-Member- und zu B befreundete Funktionen zugänglich.

Auch hier “erbt” ein B-Objekt zwar alle Komponenten von A, jedoch die Schnittstelle von A ist nicht mehr in der Schnittstelle von B enthalten! Aus diesem Grund gilt auch hier:

Ein B-Objekt ist kein A-Objekt.

Die Tatsache, dass die Klasse B von A abgeleitet ist, stellt wiederum ein *Implementierungsdetail* der Klasse B dar: B ist mit Hilfe von A realisiert (hätte aber auch ganz anders realisiert werden können!).

7.2.4 Anwenderschnittstelle und Vererbungsschnittstelle

Entwickelt man eine Klasse A, so ist der **public**-Teil die öffentliche Schnittstelle zur Klasse — auf die Komponenten dieses Zugriffsabschnitts kann jeder Anwender von Objekten der Klasse zugreifen. (Nochmals: üblicherweise stehen in der öffentlichen Schnittstelle nur Funktionskomponenten — Datenkomponenten sollte man immer im **private** bzw. im **protected** Zugriffsabschnitt unterbringen!)

Der **public**-Teil der Klasse A heißt *Anwenderschnittstelle von A*.

Neben der *gewöhnlichen* Anwendung von Objekten der Klasse A gibt es eine *spezielle* Anwendung der gesamten Klasse A, nämlich bei der Entwicklung einer neuen Klasse B kann man auf die Funktionalität der Klasse A aufbauen, indem man die Klasse B aus der Klasse A ableitet.

Wie in den letzten Abschnitten erläutert, können Memberfunktionen der Klasse B und zu B befreundete Funktionen auf die **public** A-Komponenten — zusätzlich aber auch

auf die **protected** A-Komponenten zugreifen — die Klasse B als *spezieller* „Anwender“ der Klasse A hat also gegenüber einem *normalen* Anwender erweiterte Zugriffsmöglichkeiten (nur der **private** A-Teil ist für B-Funktionen nicht verfügbar!).

Diese aus **public**- und **protected**-Teil von A bestehende, gegenüber der *normalen* Anwenderschnittstelle (**public**-Teil) erweiterte Schnittstelle heißt *Verbungsschnittstelle von A*.

Der Entwickler der Klasse B kann durch die von ihm gewählte Art der Vererbung steuern, in welchem Zugriffsabschnitt der Klasse B die von A geerbten **public** bzw. **protected** Komponenten „landen“ und damit, wie sich diese von A geerbten Komponenten bei weiterer Vererbung verhalten (wenn also aus der Klasse B eine weitere Klasse C abgeleitet wird).

7.2.5 using-Deklaration einzelner Komponenten

Bei der **public**-Ableitung einer Klasse B aus der Klasse A ist (aus B-Sicht) der weitreichendste Zugriff auf die A-Komponenten eines B-Objektes möglich:

- **public** A-Komponenten sind **public** (ererbte) B-Komponenten,
- **protected** A-Komponenten sind **protected** (ererbte) B-Komponenten,
- **private** A-Komponenten sind **private** (ererbte, aber unzugängliche) B-Komponenten.

Der Entwickler der Klasse B kann diese Zugriffsmöglichkeiten durch Wahl einer anderen Ableitungsart (**protected**- bzw. **private**-Vererbung) einschränken:

1. **protected**-Vererbung:

- **public** A-Komponenten sind **protected** (ererbte) B-Komponenten,
- **protected** A-Komponenten sind **protected** (ererbte) B-Komponenten,
- **private** A-Komponenten sind **private** (ererbte, aber unzugängliche) B-Komponenten.

2. **private**-Vererbung:

- **public** A-Komponenten sind **private** (ererbte, aber zugängliche) B-Komponenten,
- **protected** A-Komponenten sind **private** (ererbte, aber zugängliche) B-Komponenten,
- **private** A-Komponenten sind **private** (ererbte, aber unzugängliche) B-Komponenten.

Diese für alle Komponenten des entsprechenden Zugriffsabschnittes geltenden Einschränkungen können für einzelne Komponenten wieder aufgehoben werden, indem für diese A-Komponente in der Klasse B in dem Zugriffsabschnitt, in dem sie in der Klasse B „landen“ soll, eine **using**-Deklaration aufgeführt wird:

```

class A {
public:
    void f1_pub();
    void f2_pub();
    ...
protected:
    void f1_pro();
    void f2_pro();
    ...
private:
    ...
};

class B: private A {
public:
    using A::f1_pub;           // A::f1_pub jetzt public in B
                               // A::f2_pub immer noch private in B
    ...
protected:
    using A::f1_pro;           // A::f1_pro jetzt protected in B
                               // A::f2_pro immer noch private in B
    ...
private:
    ...
};

```

Eine A-Komponente, deren Zugriff durch die Vererbungsart in B eingeschränkt ist, kann auf diese Weise maximal in den Zugriffsabschnitt “gehoben” werden, in dem sie bei public-Vererbung “gelandert” wäre (in diesem Beispiel könnten die **protected** A-Komponenten maximal in den **protected** B-Teil “gehoben” werden — nicht jedoch in den **public**-Teil.)

7.3 Beispiel für (einfache) Vererbung

Als (einfaches) Beispiel für eine zur Ableitung geeignete Klasse soll folgende Klasse **Bruch** dienen (Member-Funktionen sind zur Vereinfachung implizit **inline**):

```

class Bruch {
protected:
    int zaehler;    // Komponenten protected, damit in Ableitung
                   // hierauf zugegriffen werden kann
    int nenner;
public:
    struct Bruchfehler {}; // Fehlerklasse

    // Konstruktor aus Zaehler und Nenner, auch
    // parameterlos verwendbar!
    Bruch( int z = 0, int n = 1) : zaehler(z), nenner(n)

```

```

{ if ( nenner == 0)
    throw Bruchfehler();

    // nenner garantiert positiv:
    if ( nenner < 0)
    { nenner *= -1;
      zaehler *= -1;
    }
}

// multiplikative Zuweisung:
const Bruch & operator*=(const Bruch &a)
{ zaehler *= a.zaehler;
  nenner   *= a.nenner;
  return *this;
}

// Ausgabe auf streams
void printOn(ostream &strm=cout)
{ strm << zaehler << '/' << nenner;
}

// ... weitere Member-Funktionen denkbar!

// Multiplikation als globale friend Funktion, damit
// Typumwandlung im ersten Argument moeglich wird!
friend Bruch operator*(const Bruch &a, const Bruch &b);

// ... weitere befreundete Funktionen denkbar!
};
...
// Funktionen, welche zur Klasse Bruch "geh hoeren":

// Multiplikation zweier Brueche
Bruch operator*( const Bruch &a, const Bruch &b)
{ Bruch tmp;
  tmp.zaehler = a.zaehler * b.zaehler;
  tmp.nenner  = a.nenner   * b.nenner;
  return tmp;
}

// Ausgabeoperator fuer Brueche:
ostream& operator<<(ostream &strm, const Bruch &b)
{ b.printOn(strm);
  return strm;
}
...

```

Aus dieser Klasse `Bruch` soll eine neue Bruch-Klasse `KBruch` abgeleitet werden, die der Tatsache Rechnung trägt, dass Brüche unkürzbar oder kürzbar sein können. Hierzu soll die Klasse `Bruch` um eine Komponente `bool kuerzbar`; erweitert werden, in der festgehalten ist, ob der Bruch unkürzbar (Wert von `kuerzbar` ist `false`) oder kürzbar (Wert von `kuerzbar` ist `true`) ist. Zusätzlich sollen neue Member-Funktionen `bool istKuerzbar()`; und `void kuerzen()`; eingeführt werden, die erste soll als Ergebnis liefern, ob der Bruch kürzbar ist oder nicht, und die zweite soll ggf. für das Kürzen des Bruches sorgen.

7.3.1 1. Versuch der Ableitung: nur neue Memberfunktionen definieren.

Hier ist der erste Versuch der Definition von `KBruch`. Gegenüber dem obigen Grobentwurf ist noch eine nicht öffentliche Memberfunktion `unsigned ggt() const`; eingeführt, welche den größten gemeinsamen Teiler von `zaehler` und `nenner` ermittelt und nur intern in den `KBruch`-Member-Funktionen verwendet wird und deshalb nicht in der öffentlichen Schnittstelle der Klasse steht:

```
// Deklaration der Klasse Bruch muss bekannt sein, ggf.
// Headerdatei der Klasse Bruch includen!
...
class KBruch: public Bruch { // KBruch oeffentlich von Bruch abgeleitet!
protected:
    // neue protected Elemente:
    bool kuerzbar;
    // Hilfsfunktion zur Berechnung des ggT von
    // zaehler und nenner, wird spaeter definiert!
    unsigned ggt() const;

public:
    // Neuer Konstruktor:
    KBruch(int z=0, int n=1)
        : Bruch(z,n)    // Bruch-Teil wird ueber Initialisierungsliste
                       // erzeugt!
    { // neue Komponente kuerzbar explizit gesetzt!
        kuerzbar = (ggt() > 1);
    }

    bool istKuerzbar() const
    { return kuerzbar;
    }

    void kuerzen()
    { if ( kuerzbar) // nur, falls kuerzbar
        { int teiler = ggt();

          zaehler /= teiler;
```

```

        nenner /= teiler;

        kuerzbar = false;
    }
}
};

// Implementation der ggT-Berechnung:
unsigned KBruch::ggT() const
{ if ( zaehler == 0)
    return nenner;

    // groesste Zahl ermitteln, die sowohl zaehler als
    // auch nenner ohne Rest teilt

    unsigned teiler = zaehler;
    if ( teiler < 0 )
        teiler *= -1;
    if ( nenner < teiler)
        teiler = nenner;
    // jetzt ist teiler das Minumim von abs(zaehler) und nenner!

    // finde groesste Zahl, durch welche zaehler und
    // nenner ohne Rest teilbar sind:
    while ( zaehler % teiler != 0 || nenner % teiler != 0)
        --teiler;

    return teiler;
}

```

Ein **KBruch** ist ein **Bruch** mit zusätzlichen Komponenten (Daten und Funktionen). Wird ein **KBruch** erzeugt, wird auch ein **Bruch** erzeugt (ein **KBruch** hat einen Teil, der ein **Bruch** ist).

Für das Erzeugen von Objekten sind Konstruktoren zuständig. Wird ein Konstruktor für einen **KBruch** aufgerufen, wird vom System automatisch auch ein **Bruch**-Konstruktor aufgerufen — wenn nichts anderes in der Initialisierungsliste des verwendeten **KBruch**-Konstruktors angegeben, der Standard-Konstruktor für die Klasse **Bruch**.

Bei unserem Konstruktor für die neue Klasse **KBruch** wird hier für den **Bruch**-Teil über die Initialisierungsliste der parameterbehaftete Konstruktoraufruf **Bruch(z,n)** aufgerufen. (Da der **Bruch**-Teil der neuen Klasse **KBruch** kein eigener Name verfügbar ist, wird der Konstruktor direkt über den Klassennamen **Bruch** der Basisklasse aufgerufen!)

Der **KBruch**-Konstruktor sorgt in seinem Anweisungsteil dafür, dass die neue **KBruch**-Komponente vernünftig vorbesetzt wird.

Die Definition der neuen Member-Funktionen **istKuerzbar** und **kuerzen** ist nahelegend, wobei die Funktion **kuerzen** und der Konstruktor die (mathematisch nicht

optimal implementierte) Funktion `ggt` zur Berechnung des größten gemeinsamen Teilers von `zaehler` und `nenner` benutzen.

Die zusätzlichen Komponenten sind sinnvoll definiert, die Klasse `KBruch` ist dennoch nicht ganz brauchbar, denn die neue Komponente `kuerzbar` wird durch die von der Klasse `Bruch` geerbte multiplikative Zuweisung `*` nicht richtig behandelt, wir folgendes Beispiel zeigt:

```
KBruch a(1,6);    // 1/6, nicht kuerzbar
KBruch b(3,2);    // 3/2, nicht kuerzbar
...
a *= b;           // multiplikative Zuweisung, Ergebnis ist 3/12,
                  // mathematisch kuerzbar trotzdem hat die
                  // Komponente kuerzbar der Wert false!
...
```

Der `KBruch`-Konstruktor sorgt dafür, dass bei der Initialisierung von `a` mit $\frac{1}{6}$ und der von `b` mit $\frac{3}{2}$ die jeweiligen Komponenten `kuerzbar` mit dem Wert `false` vorbesetzt sind — die Brüche sind unkürzbar!

Nach der multiplikativen Zuweisung, die mit der aus der Klasse `Bruch` geerbten Operatorfunktion `*` durchgeführt wird, hat der Bruch `a` den Wert $\frac{3}{12}$ und die Komponente `kuerzbar` des Objektes `a` hat immer noch den Wert `false`, da diese Komponente in der Klasse `Bruch` und somit auch in den `Bruch`-Member-Funktionen unbekannt ist und entsprechend auch nicht geändert wird.

Fazit: die für Objekte der Klasse `Bruch` sinnvoll definierte multiplikative Zuweisung `operator*` ist für ein Objekt der Klasse `KBruch` so nicht vernünftig definiert. Diese Zuweisung müsste für die neue Klasse `KBruch` passend neudefiniert werden!

Genau dies ist möglich:

Man kann in einer abgeleiteten Klasse eine (auch öffentliche) Memberfunktion der Basisklasse, welche für die abgeleitete Klasse nicht ausreicht, in der abgeleiteten Klasse (mit gleicher oder unterschiedlicher Signatur) neudefinieren und implementieren.

7.3.2 2. Versuch der Ableitung: zusätzlich einige schon vorhandene Memberfunktionen neudefinieren.

Wir wollen zunächst überlegen, welche der von der Klasse `Bruch` geerbten Funktionen in der Klasse `KBruch` neudefiniert werden müssen:

- Konstruktoren werden nicht vererbt — müssen also in der abgeleiteten Klasse neudefiniert werden. (Jeder Konstruktor der abgeleiteten Klasse ruft entweder implizit den parameterlosen Konstruktor oder explizit über die Initialisierungsliste einen anderen Konstruktor der Basisklasse auf!)
- die multiplikative Zuweisung (Memberfunktion der Klasse `Bruch`)

```
const Bruch & operator*=(const Bruch &);
```

muss, wie gesehen, in der Klasse `KBruch` neudefiniert werden, damit die in `KBruch` neue Komponente `kuerzbar` richtig behandelt wird.

Wir wollen in KBruch die multiplikative Zuweisung mit folgender Signatur realisieren:

```
const KBruch& operator*=( const Bruch &);
```

damit ein KBruch auch mit einem einfachen Bruch multipliziert werden kann.

Folgender Definitionsversuch dieser KBruch–Member–Funktion geht schief:

```
const KBruch & KBruch::operator*=(const Bruch &a)
{
    zaehler *= a.zaehler; // FEHLER: a.zaehler ist nicht verfuegbar!
    nenner  *= a.nenner;  // FEHLER: a.nenner  ist nicht verfuegbar!

    kuerzbar = (ggt() > 1); // Komponente kuerzbar separat berechnen

    return *this;
}
```

denn diese KBruch–Member–Funktion ...operator*=(...); ist bezüglich des Parameters `const Bruch &a` ein “normaler” Anwender der Klasse `Bruch` und somit sind die Zugriffe auf die `protected` Komponenten `zaehler` und `nenner` von `a` unzulässig! (Die Zugreifbarkeit auf die “Vererbungsschnittstelle” bezieht sich innerhalb einer Member–Funktion der abgeleiteten Klasse nur auf die geerbten `protected` Komponenten des *aktuellen Objektes*!)

Abhilfe bietet hier, in der KBruch multiplikativen Zuweisung explizit die `Bruch` multiplikative Zuweisung aufzurufen:

```
const KBruch & KBruch::operator*=(const Bruch &a)
{
    Bruch::operator*=(a);    // fuer das aktuelle Objekt die
                            // multiplikative Bruch-Zuweisung mit demselben
                            // Argument aufrufen!

    kuerzbar = (ggt() > 1); // Komponente kuerzbar separat
                            // berechnen

    return *this;
}
```

Diese ist jetzt eine vernünftige Neudefinition der multiplikativen Zuweisung:

```
KBruch a, b;
Bruch  x, y;
...    // Aufruf von:
x *= y; // const Bruch& Bruch::operator*=(const Bruch &);
x *= a; // const Bruch& Bruch::operator*=(const Bruch &);
        // KBruch a wird als Bruch aufgefasst!
```



```

...
b *= y;    // const KBruch& KBruch::operator*=(const Bruch &);
b *= a;    // const KBruch& KBruch::operator*=(const Bruch &);
           // KBruch a wird als Bruch aufgefasst!
...

```

- Die Ausgabefunktion `void printOn(ostream &strm);` könnte auch neudefiniert werden, um die Information ob kürzbar oder nicht auch in der Ausgabe zu sehen:

```

void KBruch::printOn(ostream &strm)
{
    strm << zaehler << '/' << nenner;
    strm << '(' << ( (kuerzbar) ? "" : "nicht ") << "kuerzbar)";
}

```

Eine Anwendung sieht wie folgt aus:

```

KBruch a(1,3);    // 1/3, nicht gekuerzbar
KBruch b(2,4);    // 2/4, kuerzbar
Bruch  c(1,5);    // 1/5
...
a.printOn(cout);  // Ausgabe:  1/3 (nicht kuerzbar)
b.printOn(cout);  // Ausgabe:  2/4 (kuerzbar)
c.printOn(cout);  // Ausgabe:  1/5
...

```

- Die zu `Bruch` befreundete Funktion:

```
Bruch operator*(const Bruch &, const Bruch &);
```

kann auch zur Multiplikation von `KBruch`-Objekten verwendet werden, allerdings ist das temporäre Ergebnis dieser Multiplikation ein `Bruch` (ohne die Komponente `kuerzbar`).

Ist man mit dieser Version nicht zufrieden, könnte man auch diese Funktion für `KBruch`-Objekte neudefinieren:

```
KBruch operator*(const KBruch &, const KBruch &);
```

Auf die (Neu-)Definition dieser Funktion wird hier verzichtet!

Wir wollen noch kurz überlegen, welche weiteren Funktionen vom System automatisch für die neue Klasse `KBruch` (und entsprechend für die alte `Bruch`) generiert werden und ob hier Anpassungen nötig sind:

- Vom System wird jeweils der Copy-Konstruktor erzeugt, für die Klasse `Bruch` der Konstruktor:

```
Bruch::Bruch(const Bruch &);
```

und für die Klasse `KBruch` der Konstruktor:

```
KBruch::KBruch(const KBruch &);
```

Da ein `KBruch` auch ein `Bruch` ist, kann ein `Bruch` mit einem `KBruch` erzeugt werden, das Umgekehrte geht jedoch nicht:

```
KBruch b;
Bruch a(b);    // OK!
KBruch c(a);   // FEHLER!
```

Diese hiermit definierte Umwandlung von `KBruch` nach `Bruch` wird implizit auch bei Funktionsaufrufen verwendet:

```
Bruch x;
KBruch y, z;
...
x = y * z;    // es wird die Operatorfunktion
               // Bruch operator*(const Bruch &a, const Bruch &b)
               // und hierbei aus dem KBruch y ein Bruch und aus
               // dem KBruch z ebenfalls ein Bruch erzeugt!
...
```

- Vom System wird ebenfalls jeweils der Zuweisungsoperator erzeugt, für die Klasse `Bruch`:

```
const Bruch & Bruch::operator=(const Bruch &);
```

und für die Klasse `KBruch`:

```
const KBruch & KBruch::operator=(const KBruch &);
```

Zusammen mit der automatischen Umwandlung von `KBruch` nach `Bruch` kann hiermit einem `Bruch` ein `KBruch` zugewiesen werden, umgekehrt geht das aber nicht:

```
Bruch x;
KBruch y;
...
x = y;        // OK!
y = x;        // FEHLER!
...
```

Möchte man auch einen `KBruch` mit einem `Bruch` initialisieren, müsste man eine Typumwandlung von `Bruch` nach `KBruch` (etwa in der Klasse `KBruch` einen `KBruch`-Konstruktor mit `Bruch`-Parameter oder in der Klasse `Bruch` einen Konversionsoperator von `Bruch` nach `KBruch`) definieren.

Dadurch hätte man sowohl eine Typumwandlung von `KBruch` nach `Bruch` und umgekehrt — hierdurch kommt es dann aber in der Anwendung leicht zu Mehrdeutigkeiten! Mit einer solchen Typumwandlung von `Bruch` nach `KBruch` könnte man einem `KBruch` auch einen `Bruch` zuweisen. Alternativ könnte man auch in `KBruch` einen Zuweisungsoperator:

```
const KBruch & KBruch::operator=(const Bruch &);
```

definieren.

7.3.3 Probleme mit der Neudefinition von Funktionen der Basisklasse in der abgeleiteten Klasse

Mit dieser Neudefinition der Funktionen

```
const KBruch& KBruch::operator*=(const Bruch &);
```

und

```
void KBruch::printOn(ostream&);
```

sind leider noch nicht alle Probleme beseitigt!

Der Grund hierfür ist, dass ein `KBruch` ein `Bruch` ist, ein `KBruch` wie ein `Bruch` verwendet werden kann und entsprechend kann für einen `KBruch` (indirekt über Zeiger oder Referenzen) eine für einen `KBruch` unpassende `Bruch`-Funktion aufgerufen werden, obwohl es eine für `KBruch` passende Version der Funktion gibt:

```
Bruch *p;           // Zeiger auf einen Bruch

void f(Bruch &c)     // Funktion mit Bruch-Referenz-Parameter
{
    Bruch tmp(5,2);  // temporärer Bruch 5/2
    c *= tmp;        // Multiplikation mit temp. Bruch
}

...
Bruch  a(1,3);       // 1/3
KBruch b(1,5);       // 1/5, nicht kürzbar
...
p = &a;              // OK! p zeigt auf a
p->printOn(cout);     // Ausgabe: 1/3
                      // es wird die Bruch-Funktion
                      // void printOn(ostream &) aufgerufen
p = &b;              // OK! p zeigt auf b, wobei dass, worauf
                      // p zeigt als Bruch aufgefasst wird!
b.printOn(cout);     // Ausgabe: 1/5 (nicht kürzbar)
p->printOn(cout);     // Ausgabe: 1/5 !!!
```

```

// obwohl p auf einen KBruch zeigt, wird die zur
// Klasse Bruch gehoerende Funktion
// printOn aufgerufen!
cout << b; // Ausgabe: 1/5 !!!
// es wird die Operatorfunktion
// ostream & operator<<(ostream &, Bruch &);
// aufgerufen und innerhalb dieser Funktion
// wird ueber die Referenz auf b
// void printOn(ostream &) aufgerufen!
...
a.printOn(cout); // Ausgabe: 1/3
f(a); // Funktionsaufruf
a.printOn(cout); // Ausgabe: 5/6
...
b.printOn(cout); // Ausgabe: 1/5 (nicht kuerzbar)
f(b); // Funktionsaufruf
b.printOn(cout); // Ausgabe: 5/10 (nicht kuerzbar) !!!
// innerhalb der Funktion f wird fuer b ueber die
// Referenz c der Operator *= aufgerufen. Da aber
// c eine Referenz auf Bruch ist, wird die zur
// Klasse Bruch gehoerende Operatorfunktion
// Bruch::operator*= aufgerufen, obwohl das
// tatsaechliche Objekt b ein KBruch ist!
...

```

Das Problem ist, dass zwar für ein KBruch-Objekt — aber über **Referenz auf Bruch** oder **Adresse eines Bruch** Funktionen aufgerufen werden und somit die entsprechenden Bruch-Funktionen und nicht die angepassten KBruch-Funktionen aufgerufen werden.

Es wird also nicht der tatsächliche Typ des Objektes zugrundegelegt, auf die der Zeiger bzw. die Referenz zeigt, sondern der formale Zeiger- bzw. Referenztyp (der Zeiger `p` zeigt tatsächlich auf einen KBruch, ist aber formal ein Zeiger auf Bruch, der Referenzparameter `c` der Funktion `f` ist in obigem Aufruf eine Referenz auf einen KBruch, ist aber formal eine Referenz auf einen Bruch).

Dies ist Konsequenz daraus, dass hier der Compiler anhand der formalen Typen der Zeiger bzw. Referenzen den Funktionsaufruf zur Compilierzeit umsetzen muss, ohne den erst zur Programmlaufzeit feststehenden tatsächlichen Typen der Objekte zu kennen!

Abhilfe bieten hier virtuelle Member-Funktionen, deren Aufruf, wenn er über eine Referenz oder Adresse erfolgt, erst zur Laufzeit der Programms — und dann für den tatsächlichen Typen des Objektes — umgesetzt wird (*späte Bindung*).

7.3.4 3. Versuch der Ableitung: Verwenden von virtuellen Funktionen.

Man kann in einer zur Ableitung vorgesehenen Klasse `A` einige Member-Funktionen als virtuell (Schlüsselwort `virtual`) definieren und diese Funktionen in einer abgeleiteten

Klassen B mit gleicher Signatur und mit gleichem Ergebnistyp neudefinieren:

```
class A {
    ...
    public:
        ...
        virtual void fkt(void) // virtuelle Funktion
        { ... }
        ...
};
class B: public A {
    ...
    public:
        ...
        virtual void fkt(void) // Neudefinition der
        { ... }                // virtuellen Funktion
        ...
};
```

Wird eine solche virtuelle Funktion über eine Referenz oder Adresse aufgerufen, so wird zur Programmlaufzeit anhand des tatsächlichen Types des Objektes entschieden, welche Version der Funktion aufzurufen ist:

```
A *p;
A a;
B b;
...
p = &a;
p->fkt();    // das Objekt, auf welches p zeigt, hat den
              // Typ A => Aufruf der A-Version von fkt
...
p = &b;
p->fkt();    // das Objekt, auf welches p zeigt, hat den
              // Typ B => Aufruf der B-Version von fkt
...

A &ar1 = a;  // A-Referenz ar1 ist Referenz auf A-Objekt a
ar1.fkt();  // => Aufruf der A-Version von fkt
...
A &ar2 = b;  // A-Referenz ar2 ist Referenz auf B-Objekt b
ar2.fkt();  // => Aufruf der B-Version von fkt
...
```

Die Neudefinition der virtuellen Funktion in der abgeleiteten Klasse B muss die gleiche Signatur und den gleichen Ergebnistyp wie die ursprüngliche Funktion in der Klasse A haben — als einzige Ausnahme kann, wenn in der Basisklasse A der Ergebnistyp Zeiger oder Referenz auf A war, in der abgeleiteten Klasse der Ergebnistyp Zeiger bzw. Referenz auf B sein.

In unserem Beispiel müssten also die in der Klasse KBruch neudefinierenden Funktionen `operator*=` und `printOn` bereits in der Basisklasse Bruch als virtuell vereinbart werden, die Klassendeklaration müsste also (bei gleicher Definition aller Funktionen) wie folgt aussehen:

```
class Bruch {
protected:
    int zaehler;    // Komponenten protected, damit in Ableitung
    int nenner;     // hierauf zugegriffen werden kann
public:
    struct Bruchfehler {}; // Fehlerklasse

    Bruch( int z = 0, int n = 1);

    // multiplikative Zuweisung:
    virtual const Bruch & operator*=(const Bruch &);

    // Ausgabe auf streams
    virtual void printOn(ostream &=cout);

    // ... weitere Member-Funktionen denkbar!

    // Multiplikation als globale friend Funktion, damit
    // Typumwandlung im ersten Argument moeglich wird!
    friend Bruch operator*(const Bruch &a, const Bruch &b);

    // ... weitere befreundete Funktionen denkbar!

};
...
// Funktionen, welche zur Klasse Bruch "gehören":

// Ausgabeoperator fuer Brueche:
ostream& operator<<(ostream &strm, const Bruch &b)
{ b.printOn(strm);
  return strm;
}
...
```

Die neue Klasse KBruch müsste dann wie folgt vereinbart sein:

```
class KBruch: public Bruch { // KBruch oeffentlich von Bruch abgeleitet!
protected:
    // neue protected Elemente:
    bool kuerzbar;
    // Hilfsfunktion zur Berechnung des ggT von
    // zaehler und nenner, wird spaeter definiert!
```

```

    unsigned ggt() const;

public:
    // Neuer Konstruktor:
    KBruch(int z=0, int n=1);

    bool istKuerzbar() const;

    void kuerzen();

    // Deklaration der neuzudefinierenden Funktionen:
    virtual void printOn(ostream &strm=cout);
    virtual const KBruch& operator*=(const Bruch &);
};

```

(Definition der Funktionen wie in den letzten Unterabschnitten!)
 Dieselbe Anwendung wie im letzten Unterabschnitt wird jetzt wie folgt umgesetzt:

```

Bruch *p;           // Zeiger auf einen Bruch

void f(Bruch &c)     // Funktion mit Bruch-Referenz-Parameter
{
    Bruch tmp(5,2);  // tmporaerer Bruch  5/2
    c *= tmp;        // Multiplikation mit temp. Bruch
}

...
Bruch  a(1,3);       // 1/3
KBruch b(1,5);       // 1/5, nicht kuerzbar
...
p = &a;              // OK! p zeigt auf a
p->printOn(cout);     // <---
                      // Ausgabe: 1/3
                      // es wird die Bruch-Funktion
                      // void printOn(ostream &) aufgerufen

...
p = &b;              // OK! p zeigt auf b, wobei dass, worauf
                      // p zeigt als Bruch aufgefasst wird!
b.printOn(cout);     // Ausgabe: 1/5 (nicht kuerzbar)
p->printOn(cout);     // <---
                      // Ausgabe: 1/5 (nicht kuerzbar)
                      // obwohl p ein Zeiger auf Bruch ist, wird hier,
                      // da p tatsaechlich auf einen KBruch zeigt,
                      // die KBruch Funktion printOn aufgerufen!
cout << b;           // Ausgabe: 1/5 (nicht kuerzbar)
                      // es wird die Operatorfunktion
                      // ostream & operator<<(ostream &strm, Bruch &a);

```

```

// aufgerufen und innerhalb dieser Funktion
// wird ueber die Referenz auf a
// void printOn(ostream &) aufgerufen!
// Obwohl a formal eine Referenz auf ein Bruch ist,
// wird trotzdem die KBruch printOn-Funktion
// aufgerufen, da das tatsaechliche Objekt b
// ein KBruch ist!

...
a.printOn(cout); // Ausgabe: 1/3
f(a);           // Funktionsaufruf
a.printOn(cout); // Ausgabe: 5/6
...
b.printOn(cout); // Ausgabe: 1/5 (nicht kuerzbar)
f(b);           // Funktionsaufruf
b.printOn(cout); // Ausgabe: 5/10 (kuerzbar) !!!
// innerhalb der Funktion f wird fuer b ueber die
// Referenz c der Operator *= aufgerufen. Obwohl
// c eine Referenz auf Bruch ist, wird trotzdem
// die zur Klasse KBruch gehoerende Operatorfunktion
// KBruch::operator*= aufgerufen, da das
// tatsaechliche Objekt b ein KBruch ist!

...

```

Zu beachten sind hier die beiden, für den Compiler völlig identischen Aufrufe der Funktion `printOn` (durch `<---` gekennzeichnet!). Diese syntaktisch gleichen Funktionsaufrufe werden vom System zur Laufzeit des Programms unterschiedlich umgesetzt, einmal durch die Funktion

```
void Bruch::printOn(ostream &);
```

(erster Aufruf) und einmal durch die Funktion

```
void KBruch::printOn(ostream &);
```

Mit solchen virtuellen Funktionen kann also folgendes realisiert werden:

Rufe für ein Objekt eine Funktion auf und das System (das Objekt selber) sorgt dafür, dass die zum Objekt passende richtige Funktionalität durchgeführt wird!

Dieses Verhalten wird *Polymorphie* genannt!.

7.4 Neudefinition, Überladung, virtuelle Funktionen

Die in diesem Abschnitt behandelten Techniken beziehen sich nur auf Member-Funktionen einer Klasse, nicht etwa auf `friend`-Funktionen einer Klasse oder globale Funktionen.

Insbesondere können nur Member-Funktionen virtuell sein!

Leitet man aus einer Klasse A eine neue Klasse B ab, so kann man, wie im letzten Abschnitt gesehen, eine bereits in Klasse A definierte Memberfunktion neu definieren.

Man muss unterscheiden, ob die Neudefinition in B die gleiche Signatur hat wie in A oder nicht.

7.4.1 Neudefinition mit unterschiedlicher Signatur

Beispiel:

```
class A {
    ...
    public:
        ...
        int fkt(int);
        ...
};

class B: public A {
    ...
    public:
        ...
        int fkt(double); // Neudefinition mit anderer Signatur
        ...
};
```

Formal ist diese Neudefinition der Funktion `int fkt(double);` in B eine Überladung der Funktion `int fkt(int);` aus A, trotzdem steht für Objekte der Klasse B nur noch die neue Funktion zur Verfügung:

```
B b;
int i;
double x;
...
b.fkt(x);    // rufe int B::fkt(double) auf!
...
b.fkt(i);    // rufe int B::fkt(double) auf, wandle
              // hierzu das int-Argument in double um!
...
```

Dies ergibt sich aus der in C++ eingebauten Regel:

Überladen bei Vererbung heißt überdecken!

D.h. die ursprüngliche Funktion der Basisklasse, die in der abgeleiteten Klasse mit neuer Signatur neudefiniert wurde, ist für Objekte der abgeleiteten Klasse durch die Neudefinition überdeckt — ist standardmäßig also nicht mehr verfügbar.

Diese Regel ist in C++ zur Sicherheit eingeführt worden, damit falsche Argumente beim Funktionsaufruf besser als Fehler erkannt werden können.

Soll für ein B-Objekt die A-Funktion aufgerufen werden, muss man

entweder die A-Funktion beim Aufruf explizit qualifizieren:

```

B b;
int i;
...
b.A::fkt(i);    // rufe fuer b die A-Funktion
                // int fkt(int) auf!
...

```

oder den Namen der in Klasse A definierten Funktion in B mittels des Schlüsselwortes `using` nochmals deklarieren:

```

class A {
    ...
public:
    ...
    int fkt(int);
    ...
};

class B: public A {
    ...
public:
    ...
    using A::fkt;    // Name fkt aus A auch in B bekannt machen!
    ...
    int fkt(double); // Neudefinition mit anderer Signatur
    ...
};

...
B b;
int i;
int x;
...
b.fkt(x);    // rufe int B::fkt(double) auf
...
b.fkt(i);    // rufe int A::fkt(double) auf
...

```

(Solche `using` Deklarationen funktionieren nicht bei unserem Gnu-C++-Compiler `gcc` (version 2.95.2), während der SUN-Workshop-Comiler dieses Sprachmittel bereits unterstützt!)

7.4.2 Neudefinition mit gleicher Signatur

Wird die Funktion in der Basisklasse A ohne das Schlüsselwort `virtual` definiert und in der abgeleiteten Klasse B mit gleicher Signatur (mit oder ohne Schlüsselwort `virtual`) neudefiniert, so handelt es sich hier auch um eine Überdeckung, d.h. bei

A-Objekten (auch über Zeiger oder Referenzen) sorgt der Compiler für den Aufruf der A-Funktion und bei B-Objekten wird die B-Funktion aufgerufen — für A-Objekte wird also keine *späte Bindung* bzgl. der Funktion durchgeführt. (Für ein B-Objekt `b` könnte wiederum über explizite Qualifikation `b.A : Funktionsname(Argumente)` die A-Funktion aufgerufen werden!)

Wird die Funktion in der Basisklasse `A` mit dem Schlüsselwort `virtual` definiert (genauer: das Schlüsselwort `virtual` ist bei der Funktionsdeklaration im Klassenrumpf angegeben!) und in der von `A` abgeleiteten Klasse `B` mit gleicher Signatur (Konstantheit gehört mit zur Signatur!) und gleichem Ergebnistyp neudefiniert, so wird der Aufruf der Funktion über Referenzen bzw. Adressen mittels *später Bindung* durchgeführt, d.h. nicht der Compiler, sondern das Laufzeitsystem entscheidet anhand des tatsächlichen Types des Objektes, auf welches die Referenz bzw. der Zeiger verweist, für den Aufruf der zum Objekt passenden Version der (virtuellen) Funktion.

Ist der Ergebnistyp der A-Funktion `A&` (Referenz auf `A`) oder `A*` (Adresse eines `A`), so darf der Ergebnistyp der B-Funktion `B&` (Referenz auf `B`) bzw. `B*` (Adresse eines `B`) sein — hierbei darf jeweils im Ergebnistyp die Qualifikation `const` (in beiden Funktionen gleichartig) auftreten.

Wird in der abgeleiteten Klasse `B` eine zu einer in der Basisklasse `A` als virtuell definierten Funktion gleichnamige Funktion mit anderer Signatur oder mit (wesentlich, s.o.) anderem Ergebnistyp (virtuell oder nicht) definiert, so handelt es sich nicht um die Neudefinition der virtuellen A-Funktion, sondern um eine Funktionsüberladung mit der Konsequenz, dass für B-Objekte die A-Funktion nicht mehr verfügbar ist!

Ist die Überladung in `B` selbst wieder virtuell, so haben wir es jetzt mit zwei in von `B` abgeleiteten Klassen neudefinierbaren virtuellen Funktionen zu tun!

Bei der Neudefinition einer virtuellen Funktion in der Klasse `B` (gleiche Signatur und im wesentlichen gleicher Rückgabetyt) kann das Schlüsselwort `virtual` fortgelassen werden! (*Einmal virtuell, immer virtuell!* Ich empfehle, das Schlüsselwort `virtual` trotzdem auch in der abgeleiteten Klasse anzugeben!)

Der Zugriffsabschnitt einer Neudefinition in der abgeleiteten Klasse kann ein anderer als in der Basisklasse sein (ich empfehle, eine solche Abweichung nur sehr gut überlegt einzusetzen).

7.4.3 Aufruf virtueller Funktionen

Der Aufruf einer virtuellen Funktion wird nur dann mittels *später Bindung* zur Laufzeit und nicht zur Compile-Zeit umgesetzt, wenn der Aufruf über die Adresse eines Objektes oder über eine Referenz auf ein Objekt erfolgt — ansonsten wird er *statisch* vom Compiler umgesetzt.

Erfolgt der Aufruf über explizite Qualifikation (`Klasse : Funktionsname(Argumente)`), so wird der Aufruf ebenfalls statisch umgesetzt.

Konstruktoren können niemals virtuell sein, da diese zur Erzeugung eines Objektes verwendet werden und der Typ des zu erzeugenden Objektes zur Compile-Zeit bereits für den Compiler feststehen muss.

Sollten in einem Konstruktor virtuelle Funktionen aufgerufen werden, so wird deren Aufruf ebenfalls vom Compiler *statisch* umgesetzt.

7.4.4 Virtuelle Funktionen und Defaultargumente

Virtuelle Funktionen können Defaultargumente besitzen, die wie immer bei der Funktionsdeklaration anzugeben sind.

Um die Defaultargumente kümmert sich jedoch der Compiler bei der Übersetzung und nicht das Laufzeitsystem beim Programmablauf!

Aus diesem Grunde sollte man bei der Neudeklaration einer virtuellen Funktion in einer abgeleiteten Klasse genau dieselben Defaultwerte vorsehen wie in der Basisklasse, sonst kann es vorkommen, dass zur Laufzeit zwar die richtige Funktion — aber mit (vermeindlich) falschen Defaultwerten aufgerufen wird:

```
class A {
    ...
public:
    ...
    // A-Funktion, Defaultwert 5
    virtual void fkt(int = 5);
    ...
};

class B : public A {
    ...
public:
    ...
    // Neudeklaration:
    // B-Funktion, Defaultwert 7
    virtual void fkt(int = 7);
    ...
};

...
B b;    // B-Objekt
A *p;   // A-Zeiger
...
p = &b;    // A-Zeiger p zeigt auf B-Objekt b!
p->fkt();  // Compiler setzt Defaultwert der A-Funktion,
           // also 5 ein, Laufzeitsystem sorgt aber fuer
           // den Aufruf der B-Funktion!!!
...
```

7.4.5 Virtuelle Destruktoren

Eine zur Ableitung geeignete Klasse A sollte immer einen (notfalls leeren) virtuellen Destruktor haben!

Der Grund ist der, dass aus der Klasse A eine Klasse B abgeleitet werden könnte, welche dynamische Komponenten hat und deshalb einen vernünftigen Destruktor braucht! Die naive Implementierung (ohne virtuellen Destruktor in A):

```
class A { ... };

class B: public A {
    ...
    public:
        ...
        ~B() { ... } // Destruktor in B notwendig
};
```

macht nämlich in Anwendungen folgender Art Probleme:

```
void f(void)
{
    A *p = new B;    // A-Zeiger zeigt auf B-Objekt,
                    // B-Objekt habe dynamische Komponenten
    ...

    delete p;        // hier wird der B-Destruktor NICHT aufgerufen,
                    // sondern nur der A-Destruktor!!!
}
```

In einer solchen Anwendung bleiben die für das B-Objekt (`new B`) erzeugten dynamischen Komponenten als “Speichermüll” zurück, da der zur “Entsorgung des Mülls” definierte B-Destruktor gar nicht aufgerufen wird!

Abhilfe ist hier, bereits in der Klasse A den Destruktor virtuell zu definieren:

```
class A {
    ...
    public:
        ...
        virtual ~A() {} // leerer Destruktor, aber virtuell!!
};

class B: public A {
    ...
    public:
        ...
        ~B() { ... } // Destruktor in B mit vernuenftiger Implementierung,
                    // da gleiche Signatur auch virtuell!!
        // Besser waere allerdings:
        // virtual ~B() { ... }
};
```

In der gleichen Anwendung wie oben:

```
void f(void)
{
    A *p = new B;    // A-Zeiger zeigt auf B-Objekt,
```

```

        // B-Objekt habe dynamische Komponenten
    ...

    delete p;        // hier wird der B-Destruktor aufgerufen!!!
}

```

wird dann, obwohl über einen A-Zeiger aufgerufen, der B-Destruktor durchgeführt!

7.4.6 Zuweisung virtuell?

Für jede Klasse A wird der Zuweisungsoperator

```
const A& operator=(const A&);
```

automatisch vom System erzeugt, wobei diese Operatorfunktion nicht virtuell ist.

Leitet man eine neue Klasse B von der Klasse A ab:

```

class B : public A {
    ...
};

```

so wird für B auch automatisch der (nicht virtuelle) Zuweisungsoperator

```
const B& operator=(const B&);
```

erzeugt, der für jede Komponente und den (die) geerbten Teil(e) von B die jeweilige Zuweisung aufruft.

Dieser für B erzeugte Zuweisungsoperator hat aber eine andere Signatur (`const B&`) als die A-Zuweisung (`const A&`).

Dies hat Folgendes zur Konsequenz:

```

A  a;    // A-Objekt
B  b;    // B-Objekt
A *p;    // A-Zeiger

a = b;    // OK, falls B oeffentlich von A abgeleitet:
           // aufgerufen wird  A::operator=(const A&),
           // wobei b als ein A "aufgefasst" wird.
b = a;    // FEHLER: falscher Argumenttyp fuer
           // B::operator=(const B&),
           // ein A-Objekt ist kein B-Objket!

p = &b;    // OK, ein B-Objekt ist ein A-Objekt

*p = a;    // OK, *p wird als A-Objekt aufgefasst, dem
           // mittels A::operator=(const A&) das
           // A-Objekt a zugewiesen wird!
           // p zeigt aber tatsaechlich auf B-Objekt b!

```

d.h. (über Standard-Zuweisungen) kann man einem B-Objekt nicht direkt ein A-Objekt zuweisen (2. Zuweisung in obigem Beispiel), über den Umweg eines Zeigers

auf **A** (oder auch einer Referenz auf ein **A**) ist das dennoch möglich (letzte Zuweisung in obigem Beispiel), wobei jedoch für das **B**-Objekt die **A**-Zuweisung ausgeführt wird. Möchte man, dass in diesem Fall (über einen Zeiger oder Referenz aus **A**) eine der Klasse **B** “angepasste” Zuweisung durchgeführt wird, muss man

1. bereits in der Klasse **A** die Zuweisung als virtuell definieren (wobei man die Funktionalität der Zuweisung vollständig selbst definieren muss — also alle Komponenten und evtl. geerbten Teile einander zuweisen):

```
class A {
    ...
    public:
        ...
        virtual const A& operator=(const A& a)
        {
            ... // Komponenten zuweisen
        }
        ...
};
```

und

2. in der Klasse **B** einen Zuweisungsoperator mit gleicher Signatur neudefinieren:

```
class B : public A {
    ...
    public:
        ...
        virtual const B& operator=(const A& a)
        // gleiche Signatur:      ~~~~~~
        {
            ...
        }
        ...
};
```

(Schlüsselwort **virtual** könnte fortgelassen werden — da Neudefinition einer virtuellen **A**-Funktion mit gleicher Signatur ist diese automatisch virtuell!), wobei man dann ebenfalls die Funktionalität dieser Operatorfunktion vollständig selbst definieren muss!

Zu beachten ist, dass (für **B**-Objekte) neben dieser neudefinierten Zuweisung:

```
virtual const B& operator=(const A&);
```

zusätzlich die vom System automatisch erzeugte (nicht virtuelle) Zuweisung

```
const B& operator=(const B&);
```

(hat andere Signatur!) vorhanden ist.

Die erstere kann auch über **A**-Zeiger oder **A**-Referenzen, wenn diese tatsächlich auf ein **B**-Objekt zeigen, aufgerufen werden (geht nur, weil in der Klasse **A** diese Zuweisung bereits als virtuell eingeführt wurde!), die zweite nicht!

Leitet man von **B** weitere Klassen ab, so muss man evtl. auch diese zweite Zuweisung virtuell vereinbaren!

7.5 Polymorphie

Mittels virtueller Funktionen hat man die Möglichkeit, “in Objekten der Basisklasse zu denken“ und virtuelle Funktionen (der Basisklasse) aufzurufen und das System selber entscheiden zu lassen, dass für konkrete Objekte (abgeleiteter Klassen) die richtige, zum tatsächlichen Objekt passende Version der virtuellen Funktion aufgerufen wird!

7.5.1 Beispiel für die Verwendung der Polymorphie

Dies soll an einem Beispiel erläutert werden:

Zunächst wird eine “universelle“ (noch ziemlich primitive) Basisklasse zur Verwaltung einer Linearen Liste entworfen:

```
#include <iostream>

class Link {
private:
    Link * next;
public:
    // Konstruktor
    Link() { next = 0; }

    // vorne einfüegen
    void insert( Link &a)
    { a.next = next;
      next = &a;
    }

    // vorne entfernen
    Link * exsert(void)
    { if ( next == 0)
        return 0;

      Link * tmp = next;
      next = next->next;
      tmp->next = 0;

      return tmp;
    }
}
```



```

    // virtueller Destruktor
    virtual ~Link() {}

    virtual void printOn(ostream & strm) const
    { strm << " () "; }
};

ostream & operator<< ( ostream& strm, Link &l)
{ l.printOn(strm);
  return strm;
}

```

Das Wesentliche bei allen Linearen Listen ist der Zeiger auf das nächste Element. Die Klasse `Link` hat nur diesen Zeiger mit Namen `next` und ein paar, gleich im Klassenrumpf definierte Memberfunktionen:

1. Den Konstruktor zur Vorbesetzung des Zeigers `next` mit dem Wert 0.
2. Die Funktion `insert` zum Einfügen eines Elementes vorne in der Liste.
3. Die Funktion `exsert` zum Entfernen des ersten Elementes aus der (nicht leeren) Liste. Die Funktion gibt die Adresse des entfernten Elementes bzw. den Adresswert 0 zurück.
4. Den leeren, virtuellen Destruktor.
5. Eine virtuelle Ausgabefunktion `printOn` zur Ausgabe des Inhalts des ersten Listenelementes. Da die Listenelemente (noch) keinen eigentlichen Inhalt haben, wird in dieser Funktion eine leeres Klammernpaar `()` ausgegeben!

Neben diesen Memberfunktionen ist der Ausgabeoperator

```
ostream & operator<< ( ostream& strm, Link &l);
```

definiert, der (mittels "Überkreuzung") für den Referenz-Parameter `l` die (virtuelle) Memberfunktion `printOn` aufruft. (Dies ist eine sehr rudimentäre Lineare Liste! Nur um das Beispiel nicht zu überfrachten, sind keine weiteren Funktionalitäten vorgesehen!)

Von dieser Basisklasse `Link` werden jetzt einige andere Klassen so abgeleitet, dass in den abgeleiteten Klassen die Listenelemente einen "Inhalt" bekommen:

1. Listenelemente der neuen Klasse `intLink` haben (zusätzlich) einen ganzzahligen Wert als Inhalt:

```

class intLink: public Link {
private:
    int wert;
public:
    intLink(int a = 0) : wert(a) {}
    virtual void printOn(ostream& strm) const
    { strm << " (" << wert << ") "; }
};

```

Im Konstruktor wird dass das ganzzahlige Argument (Defaultwert 0) in der neuen Komponente gespeichert.

In der (virtuellen) Ausgabefunktion wird der Wert der neuen, ganzzahligen Komponente (in runden Klammern) ausgegeben!

2. Listenelemente der neuen Klasse `doubleLink` haben (zusätzlich) einen `double` Wert als Inhalt:

```
class doubleLink: public Link {
private:
    double wert;
public:
    doubleLink(double a = 0.0) : wert(a) {}
    virtual void printOn(ostream& strm) const
    { strm << " (" << wert << ")" ";}
};
```

Im Konstruktor wird dass das `double` Argument (Defaultwert 0.0) in der neuen Komponente gespeichert.

In der (virtuellen) Ausgabefunktion wird der Wert der neuen, `double` Komponente (in runden Klammern) ausgegeben!

3. Listenelemente der neuen Klasse `stringLink` haben (zusätzlich) einen `char`-Zeiger als Inhalt.

```
#include <cstring> // wegen strcpy und strlen
```

```
class stringLink: public Link {
private:
    char *wert;
public:
    stringLink(const char *a = "")
    {
        wert = new char[strlen(a)+1];
        strcpy(wert,a);
    }
    virtual void printOn(ostream& strm) const
    { strm << " (" << wert << "\\") ";}

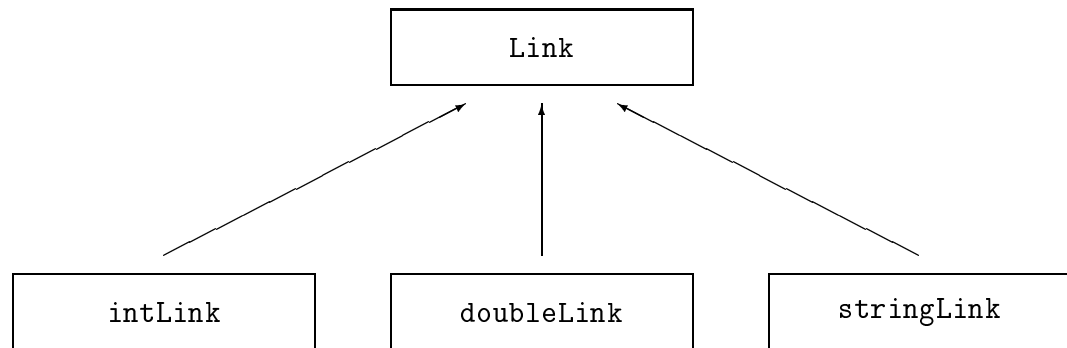
    virtual ~stringLink()
    {
        delete [] wert;
    }
};
```

Der Zeiger `wert` zeigt auf eine, bei der Konstruktion anzugebende, dynamisch abgespeicherte Zeichenkette (Defaultwert: leere Zeichenkette).

Die virtuelle Ausgabefunktion `printOn` ist entsprechend angepasst.

Da es sich hier um eine Klasse mit dynamischen Komponenten handelt, ist der Destruktor so definiert, dass die dynamische Komponente wieder freigegeben wird. (Auf Implementation des eigentlich auch erforderlichen Copy-Konstruktors und einer Zuweisung wurde hier verzichtet!)

Hierdurch ist jetze eine (kleine) “Klassenhierarchie“ aufgebaut, welche in folgendem Schaubild dargestellt ist:



In einer Anwendung kann die Klasse `Link` wie folgt verwendet werden:

```

void f(Link &);

int main(void)
{ // leere Lineare Liste
  Link anfang;

  // f füllt irgendwie die Lineare Liste
  f(anfang);

  // Elemente aus der Linearen Liste herausholen
  // und ausgeben:
  while ( ( tmp = anfang.exsert()) != 0 )
  { cout << *tmp;
    delete tmp;
  }
  cout << endl;

  return 0;
}

void f(Link &p)
{ // zunächst mal ein paar int-Werte abspeichern:
  for ( int i = 0; i < 5; ++i)
  { Link * tmp = new intLink(i);
    p.insert(*tmp);
  }
}
  
```

```

// jetzt ein paar double Werte abspeichern:
for ( int i = 0; i < 5; ++i)
{ Link * tmp = new doubleLink(double(i) + .5);
  p.insert(*tmp);
}

// jetzt noch zwei Strings abspeichern
Link * tmp = new stringLink("Hallo");
p.insert(*tmp);
tmp = new stringLink("Leute");
p.insert(*tmp);

return;
}

```

Im Hauptprogramm wird eine Funktion aufgerufen, welche die zunächst leere Lineare Liste irgendwie (dynamisch) mit Werten füllt (ganzzahlige, Gleitkomma und Zeichenketten). Anschließend wird im Hauptprogramm in eine Schleife das jeweils erste Element vom Typ `Link` aus der Liste herausgeholt und für das herausgeholte Objekt der Ausgabeoperator `<<` aufgerufen — wobei im Hauptprogramm gar nicht mehr klar ist, um was für konkrete Objekte (ein `Link`-, ein `intLink`-, ein `doubleLink`- oder ein `stringLink`-Objekt?) es sich handelt!

Die mittels der (im Ausgabeoperator `<<` aufgerufenen) virtuellen Ausgabefunktion `printOn` ermöglichte Polymorphie bewirkt, dass die zum Objekt passende Ausgabefunktion durchgeführt wird (Ausgabe von ganzzahligen bzw. von `double` Werten oder Zeichenketten, je nachdem, was für eine Art `Link` das aus der Liste herausgeholte Objekt tatsächlich ist!).

Die Ausgabe dieser Anwendung sieht so aus:

```
("Leute") ("Hallo") (4.5) (3.5) (2.5) (1.5) (0.5) (4) (3) (2) (1) (0)
```

7.5.2 Laufzeittypinformation

Wenn man (als Programmierer) virtuelle Funktionen sinnvoll einsetzt, braucht man im Allgemeinen den genauen Typ eines Objektes in einer Klassenhierarchie gar nicht zu kennen, um mit dem Objekt vernünftig umgehen zu können (siehe letztes Beispiel!). Trotzdem bietet der neue Standard mittels der Laufzeittypinformation (englisch: *RunTime Type Information*, abgekürzt *RTTI*) die Möglichkeit, zur Laufzeit des Programms abzufragen, ob ein Objekt einen gewissen Typ hat oder nicht!

Dies ist allerdings nur für Objekte solcher Typen (Klassen) möglich, in welchen mindestens eine virtuelle Funktion definiert ist (sog. *polymorphe* Typen/Klassen).

Hierzu steht der Operator:

```
dynamic_cast<Typ>(Argument)
```

zur Verfügung, welcher als Argument (in den runden Klammern) einen Zeiger oder eine Referenz auf ein polymorphes Objekt benötigt.

In den spitzen Klammern ist dann der entsprechende Zeiger- oder Referenztyp anzugeben, zu dem man wissen möchte, ob das Objekt von diesem Typ ist oder nicht (dieser "Zieltyp" muss nicht unbedingt polymorph sein!).

Bei einem Zeiger `p` ist die Anwendung wie folgt:

```
dynamic_cast<T *>(p)
```

zeigt jetzt `p` auf ein Objekt vom Typ `T`, d.h. `*p` ist wirklich ein `T`-Objekt oder `T` ist eine (eindeutige) öffentliche Basisklasse der tatsächlichen Klasse von `*p` (bei Mehrfachvererbung kann die "Eindeutigkeit" gestört sein!), so ist das Ergebnis dieses `dynamic_cast`-Operators die Adresse des `T`-Teils des Objektes `*p`.

Ist dies nicht der Fall (`*p` ist kein `T`-Objekt bzw. der Typ von `*p` ist nicht oder nicht eindeutig von `T` abgeleitet), so ist das Ergebnis dieses Operators der (ungültige) Adresswert 0.

Hat `p` selbst den Wert 0, so ist das Ergebnis des `dynamic_cast`'s ebenfalls 0.

Als Beispiel wollen wir in unserer Linearen-Listen-Anwendung den tatsächlichen Typen der aus der Linearen-Liste herausgeholtene Objekte erfahren:

```
void f(Link &);

int main(void)
{ Link anfang, *tmp;

  f(anfang);

  while ( ( tmp = anfang.exsert()) != 0 )
  { // Ist *tmp ein Link?
    if ( dynamic_cast<Link *>(tmp) != 0 )
      cout << "Link ";

    // ist *tmp ein intLink?
    if ( dynamic_cast<intLink*>(tmp) != 0 )
      cout << "intLink " ;

    // Ist *tmp ein doubleLink?
    if ( dynamic_cast<doubleLink *>(tmp) != 0 )
      cout << "doubleLink ";

    // Ist *tmp ein stringLink?
    if ( dynamic_cast<stringLink *>(tmp) != 0 )
      cout << "stringLink ";

    cout << *tmp << endl;
    delete tmp;
  }

  return 0;
}
```

(gleiche Definition der Typen und der Funktion `f` wie oben).

Als Ausgabe erhält man:

```
Link stringLink  ("Leute")
Link stringLink  ("Hallo")
Link doubleLink  (4.5)
Link doubleLink  (3.5)
Link doubleLink  (2.5)
Link doubleLink  (1.5)
Link doubleLink  (0.5)
Link intLink     (4)
Link intLink     (3)
Link intLink     (2)
Link intLink     (1)
Link intLink     (0)
```

(Natürlich ist jeder `stringLink`, `intLink` und `doubleLink` auch ein `Link`, deshalb sind jeweils zwei der `if`-Bedingungen wahr und deshalb jeweils zwei Typangaben!)

Die Verwendung von `dynamic_cast` mit Referenzen (`r` sei eine Referenz auf ein polymorphes Objekt):

```
dynamic_cast<T &>(r)
```

gestaltet sich insofern schwieriger, da eine Referenz an sich nicht “ungültig” sein kann (ungültige Zeiger haben den Wert 0!) und somit die Antwort auf die Frage *Typ ok oder nicht ok* nicht am Ergebnis dieses Operators ablesbar sein kann (das Ergebnis ist eine Referenz auf T)!

Ist `r` nun Referenz auf ein Objekt vom Typ `T` (oder auf ein Objekt eines Types mit “eindeutiger” öffentlicher Basisklasse `T`), so liefert dieser Operator eine Referenz auf den `T`-Teil von `r`.

Ist dies jedoch nicht der Fall (`r` ist kein `T`-Objekt bzw. der Typ von `r` ist nicht oder nicht eindeutig von `T` abgeleitet), so wirft dieses `dynamic_cast` eine Ausnahme vom Standard-Ausnahmetyp `bad_cast` aus und es muss mit den Mitteln der Ausnahmebehandlung auf diese Situation reagiert werden. (Zur Verwendung dieses Ausnahmetypes `bad_cast` müssen auf unserem System die Headerdateien `<stdexcept>` und `<typeinfo>` includet werden!)

Aus diesem Grund sollte man bei Referenzen nicht mittels `dynamic_cast` leichtfertig “mal eben nachfragen”, ob die Referenz einen gewissen Typ hat — wenn dies nämlich nicht der Fall ist, fängt man sich eine Ausnahme ein!

Die Verwendung eines solchen `dynamic_cast` mit Referenzen sollte nur dann eingesetzt werden, wenn es für den weiteren Programmablauf unabdingbar ist, dass die Referenz diesen gewissen Typ hat.

7.5.3 typeid

Zur Ermittlung des tatsächlichen Types eines Objektes (nicht den einer Basisklasse) stellt der Standard den Operator `typeid` zur Verfügung.

Er kann für einen Typen oder einen Ausdruck aufgerufen werden und liefert eine Referenz auf ein konstantes Objekt des in der Headerdatei `TypeInfo` definierten (polymorphen) Typs `TypeInfo`.

Die genaue Spezifikation dieses Typs ist teilweise implementierungsabhängig, mindestens folgende Funktionalität ist vom Standard vorgeschrieben:

```
class TypeInfo {
public:
    virtual ~TypeInfo();           // polymorpher Typ

    // Vergleich von TypeInfo-Objekten
    bool operator==(const TypeInfo &) const;
    bool operator!=(const TypeInfo &) const;

    // liefert einen (implementierungsabhängigen) String,
    // der den Typ beschreibt
    const char * name() const;

    ...

private:
    // Copy-Konstruktor verbieten
    TypeInfo(const TypeInfo &);

    // Zuweisung verbieten
    TypeInfo & operator=(const TypeInfo &);

    ...
};
```

Insbesondere kann man abfragen, ob ein Objekt einen gewissen (tatsächlichen) Typ hat oder nicht:

```
...
if ( typeid( *tmp ) == typeid( intLink ) )
    { ... }
else if ( typeid( *tmp ) == typeid( doubleLink ) )
    { ... }
...
```

Im Fehlerfall (mir ist es noch nicht gelungen, einen solchen zu provozieren) wirft der `typeid`-Operator eine Ausnahme des Typs `bad_typeid` aus (ggf. `<stdexcept>` includen).

7.6 Template-Klassen und Vererbung

Auch eine Template-Klasse kann von einer anderen Klasse abgeleitet werden (die Basisklasse selbst kann eine "normale" Klasse oder selbst wieder eine Template-Klasse sein).

7.6.1 Ableitung einer Template-Klasse von einer “normalen” Klasse

Es ist etwa möglich, aus obiger Klasse `Link` eine Template-Klasse `TLink` abzuleiten, so dass in der Linearen Liste Elemente (fast) beliebigen Types abgelegt werden können:

- “Normale” Basisklasse:

```
class Link {
private:
    Link * next;
public:
    // Konstruktor
    Link() { next = 0; }

    // vorne einfuegen
    void insert( Link &);

    // vorne entfernen
    Link * exsert(void);

    // virtuelle Ausgabefunktion
    virtual void printOn(ostream & strm) const;

    // virtueller Destruktor
    virtual ~Link() {}
};

// globaler Ausgabeoperator
ostream & operator<<( ostream& strm, Link &l);
```

- davon abgeleitete Template-Klasse:

```
template <class T>
class TLink: public Link {
private:
    T wert;
public:
    TLink(const T& a ) : wert(a) {}
    //          ~~~~~~
    virtual void printOn(ostream& strm) const
    { strm << " (" << wert << ") ";}
    //          ~~~~~~
};
```

In einer hiermit gebildeten Linearen Liste können beliebige Typen `T` gespeichert werden (Einschränkung: für den Typen `T` muss der Copy-Konstruktor und der Ausgabeoperator `<<` definiert sein, siehe markierte Stellen in obigem Quelltext!).

Anwendung:


```
void f(Link &p)
{
    Link *tmp;
    int i;

    // Abspeichern von int-Werten
    tmp = new TLink<int>(7);
    p.insert(*tmp);

    // Abspeichern von double-Werten
    tmp = new TLink<double>(7.5);
    p.insert(*tmp);

    // Abspeichern von int-Zeigern
    tmp = new TLink<int *>(&i);
    p.insert(*tmp);

    // Abspeichern eines Bruch:
    tmp = new TLink<Bruch>(i);
    p.insert(*tmp);

    // Abspeichern eines KBruch:
    tmp = new TLink<KBruch>(i+5);
    p.insert(*tmp);

    return;
}
```

7.6.2 Ableitung einer Template-Klasse von einer anderen Template-Klasse

Eine Template-Klasse kann von einer anderen Template-Klasse abgeleitet werden. Die Klasse `Vector` könnte etwa die Verallgemeinerung des Feldbegriffs sein, bei dem aber Feldunter- und Feldüberlauf abgefangen werden (ein Feld vom Typ `T` hat die bei der Konstruktion angegebene Länge `len`, der Indexoperator ist definiert, wirft aber bei negativem Index oder Index größer als `len` eine Ausnahme aus!):

```
template <class T>
class Vector {
protected:
    T *feld;
    int len;
public:
    // loakel Fehlerklasse:
    struct Feldzugriffsfehler {};

    // Konstruktor
```

```

Vector( int = 10);

// wegen dynamischer Komponente
// Copy-Konstruktor
Vector(const Vector<T> &);
// Zuweisung
const Vektor<T>& operator=(const Vector&);
// Destruktor
virtual ~Vector();

// Elementzugriff:
T& operator[](int) throw(Feldzugriffsfehler);
const T& operator[](int) const throw(Feldzugriffsfehler);
};

```

Von dieser Klasse könnte eine weitere Feld-Klasse `Vec` abgeleitet werden, bei der die Feldindizierung nicht unbedingt bei 0 anfängt, sondern an einem beliebigen ganzzahligen Wert (mögliche Indexwerte könnten z.B. -5 bis +5 oder von 1 bis 100 sein!):

```

template <class T>
class Vec: protected Vector<T> {
protected:
    int base;
public:
    // Konstruktor:
    // l ist Feldlaenge, b ist Index des ersten Feldelementes:
    Vec(int l, int b) : Vector<T>(l), base(b) {};

    T& operator[](int i) throw(Vector<T>::Feldzugriffsfehler)
    { return Vector<T>::operator[](i - base);
    }

    const T& operator[](int i) const throw(Vector<T>::Feldzugriffsfehler)
    { return Vector<T>::operator[](i - base);
    }
};

```

Ein manchmal nützlicher Trick ist es, der Template-Basisklasse, von der eine Template-Klasse abgeleitet wird, die abgeleitete Klasse als Typparameter zu übergeben:

```

template <class T> class alt { ... }

template <class T> class neu : public alt< neu<T> >
{ ... }

```

7.6.3 “Normale” Klassen von Template-Klassen ableiten

Eine normale Klasse kann nicht von einer Template-Klasse, sondern ggf. nur von einer instantiierten Template-Klasse abgeleitet werden:

```
// Template-Klasse:
template <class T>
class Vector { ... }

// normale Klasse:
class intvector : public Vector<int>
{ ... }
```

7.7 Konstruktoren und Vererbung

Wie bereits erwähnt wird, wenn ein Objekt einer abgeleiteten Klasse (wie immer durch einen Konstruktor der abgeleiteten Klasse) erzeugt wird, auch ein Konstruktor der Basisklasse aufgerufen.

Der Konstruktor der Basisklasse wird aufgerufen bevor der Anweisungsteil des Konstruktors der abgeleiteten Klasse ausgeführt wird. Im Anweisungsteil des Konstruktors der abgeleiteten Klasse kann also davon ausgegangen werden, dass der geerbte Teil bereits (durch den Konstruktor der Basisklasse) initialisiert wurde.

Dies soll an einem Beispiel verdeutlicht werden, in dem von einer Klasse A eine Klasse B und von der Klasse B eine Klasse C abgeleitet wird:

```
class A { ... }

class B : public A { ... }

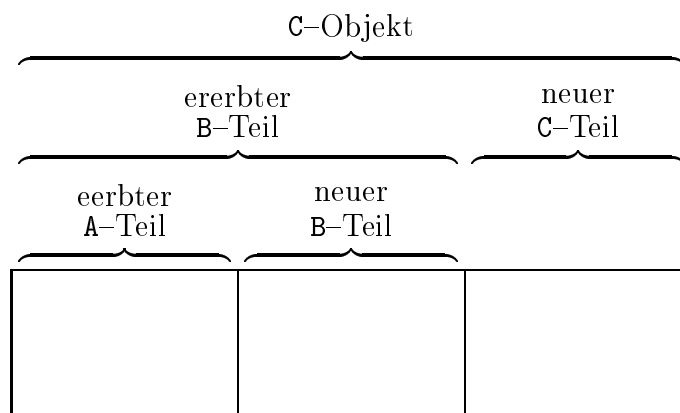
class C : public B { ... }
```

(Auf die Art der Vererbung kommt es hierbei nicht an!)

Ein C-Objekt hat seinen “neuen C-Teil” und den ererbten “B-Teil”.

Der ererbte B-Teil (des C-Objektes) hat wiederum den “in B neuen Teil” und den ererbten “A-Teil”.

Ein C-Objekt setzt sich also insgesamt aus folgenden Teilen zusammen:



Es werde ein **C**-Objekt erzeugt:

- Zunächst wird der für das ganze C-Objekt notwendige Speicher (uninitialisiert, "roh") zur Verfügung gestellt — auch für die ererbten **B**- und **A**-Teile!
- Bevor der Anweisungsteil des **C**-Konstruktors durchgeführt wird, wird der (ein) **B**-Konstruktor aufgerufen (ohne dass hierbei neuer Speicher für den **B**-Teil reserviert würde!).
- Bevor der Anweisungsteil des **B**-Konstruktors durchgeführt wird, wird der (ein) **A**-Konstruktor aufgerufen (ohne dass hierbei neuer Speicher für den **A**-Teil reserviert würde!).
- Der Anweisungsteil des **A**-Konstruktors wird ausgeführt. (Der **A**-Teil ist jetzt initialisiert!)
- Anschließend wird jetzt der Anweisungsteil des **B**-Konstruktors ausgeführt. (Der **B**-Teil ist jetzt initialisiert!)
- Schließlich wird der Anweisungsteil des **C**-Konstruktors ausgeführt. (Das ganze **C**-Objekt ist jetzt initialisiert!)

Die (Anweisungsteile der) Konstruktoren werden also in der Ableitungsreihenfolge der Klassen in der Klassenhierarchie (zuerst Basisklasse, dann abgeleitete Klasse usw.) ausgeführt.

Welcher **C**-Konstruktor zur Erzeugung des **C**-Objektes genommen wird, entscheidet der Compiler (nicht das Laufzeitsystem, Konstruktoren können nicht virtuell sein!) anhand des Kontextes, in dem die Erzeugung des **C**-Objektes anfällt (expliziter/impliziter Konstruktoraufruf).

Einfluss auf die Wahl des zu verwendenden **B**-Konstruktors kann man nur über eine Initialisierungsliste des verwendeten **C**-Konstruktors nehmen. (Wenn nicht anders in der Initialisierungsliste des **C**-Konstruktors spezifiziert, wird der parameterlose **B**-Konstruktor verwendet — dieser muss insbesondere verfügbar sein!).

Einfluss auf die Wahl des zu verwendenden **A**-Konstruktors kann man nur über eine Initialisierungsliste des verwendeten **B**-Konstruktors nehmen. (Wenn nicht anders in der Initialisierungsliste des **B**-Konstruktors spezifiziert, wird der parameterlose **A**-Konstruktor verwendet — dieser muss insbesondere verfügbar sein!).

Eine gewisse Ausnahme stellen hier Standard-Copy-Konstruktoren dar, die vom System für jede Klasse automatisch generiert werden:

Wird das **C**-Objekt mit dem automatisch generierten C-Copy-Konstruktor erzeugt, wird für den **B**-Teil auch der **B-Copy-Konstruktor** verwendet (unabhängig davon, ob dieser der automatisch erzeugte oder selbstdefiniert ist — er muss jedenfalls verfügbar sein!).

Ist jedoch für die Klasse **C** der Copy-Konstruktor selbst definiert, wird standardmäßig für den **B**-Teil der parameterlose B-Konstruktor — und nicht der **B-Copy-Konstruktor** — aufgerufen.

Soll bei der Erzeugung eines **C**-Objektes mit dem selbstdefinierten **C**-Copy-Konstruktor nicht der parameterlose **B**-Konstruktor, sondern ein anderer (etwa der **B**-Copy-Konstruktor) aufgerufen werden, so muss der gewünschte **B**-Konstruktor in der Initialisierungsliste des selbstdefinierten **C**-Copy-Konstruktors spezifiziert werden.

7.8 Destruktoren und Vererbung

Wird für ein Objekt einer in einer Klassenhierarchie ziemlich weit unten stehenden Klasse der Destruktor aufgerufen, so werden vom System natürlich auch für jeden (von höher stehenden Basisklassen) geerbten Teil des Objektes der entsprechende Destruktor aufgerufen.

Hierbei ist der Aufruf der einzelnen Destruktoren genau in umgekehrter Reihenfolge der entsprechenden Konstruktoraufrufe bei der Erzeugung des Objektes.

Als Beispiel soll wiederum folgende Klassenhierarchie erhalten:

```
class A { ... }

class B : public A { ... }

class C : public B { ... }
```

(Auch hier kommt es nicht auf die Vererbungsart an!)

Wird ein **C**-Objekt zerstört,

- wird zunächst der Anweisungsteil des **C**-Destruktors ausgeführt,
- anschließend der Anweisungsteil des **B**-Konstruktors und
- hiernach der Anweisungsteil des **A**-Destruktors.
- Schließlich wird (jetzt zum Schluss erst) der gesamte Speicherbereich für das **C**-Objekt (inklusive **B**- und **A**-Teil) freigegeben!

7.9 Vererbung und Klassen mit dynamischen Komponenten

Auch eine Klasse (Name sei **A**) mit dynamischen Komponenten ist zur Vererbung geeignet, wenn man die üblichen Regeln einhält:

- Einen vernünftigen Destruktor definieren. Dieser sollte allerdings — wie immer bei zur Vererbung vorgesehenen Klassen — virtuell sein.
- Einen vernünftigen Copy-Konstruktor definieren.
- Einen vernünftigen Zuweisungsoperator definieren.

Ist dies geschehen, kann man (relativ) problemlos von dieser Klasse eine neue Klasse **B** ableiten, denn:

- Der vom System automatisch erzeugte B-Copy-Konstruktor:

```
B::B(const B&);
```

“ruft“ für den A-Teil den A-Copy-Konstruktor auf, der ja in A passend zu den dynamischen Komponenten von A definiert wurde.

Zu beachten ist jedoch, dass, wenn in der Klasse B der Copy-Konstruktor aus irgendeinem Grund selbst neudefiniert werden muss (etwa, weil in der Klasse B neue dynamische Komponenten hinzukommen), in der Initialisierungsliste des B-Copy-Konstruktors der A-Copy-Konstruktor aufgerufen werden muss! (Ansonsten würde vom System der parameterlose A-Konstruktor aufgerufen. Im Allgemeinen ist dies nicht sinnvoll!)

- Die vom System automatisch erzeugte Zuweisung:

```
const B& B::operator=(const B&);
```

ruft für den A-Teil die A-Zuweisung auf, die ja in A passend zu den dynamischen Komponenten von A definiert wurde.

Muss aus irgendeinem Grund (etwa, weil in der Klasse B wiederum neue dynamische Komponenten hinzukommen) der Zuweisungsoperator selbst neudefiniert werden, so muss man die Funktionalität der B-Zuweisung vollständig selbst definieren, man kann aber für den A-Teil über explizite Qualifikation die (vernünftig definierte!) A-Zuweisung aufrufen!

(Die gleichen Aussagen treffen auch dann zu, wenn B nicht von A abgeleitet wird, sondern in B eine “normale“ Komponente vom Typ A vorhanden ist!)

7.10 Ausnahmen und Vererbung

Natürlich kann man auch Vererbung bei der Definition von Ausnahmeklassen verwenden.

Bei der Verwendung eines auf einem Feld beruhenden Stacks kann es zu einem Stackunterlauf (zu oft Funktion `pop` aufgerufen) oder zu einem Stacküberlauf (zu oft Funktion `pus` aufgerufen) kommen.

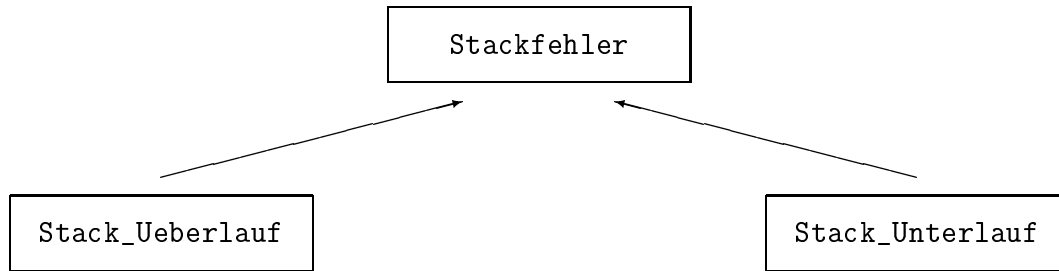
Damit man die einzelnen Fehler einerseits unterscheiden, andererseits aber auch gemeinsam behandeln kann, könnte man eine Klassenhierarchie von Fehlerklassen im Zusammenhang mit Stacks entwerfen:

```
// allgemeiner Fehler bei Behandlung eines Stacks
class Stackfehler { ... };
```

```
// Stackunterlauf, von Stackfehler abgeleitet
class Stack_Unterlauf: public Stackfehler { ... };
```

```
// Stackueberlauf, von Stackfehler abgeleitet
class Stack_Ueberlauf: public Stackfehler { ... };
```

Schaubild:



Gleichzeitig könnte man auf die Idee kommen, bei einem Stacküberlauf den Wert noch zu sichern, der nicht mehr in den Stack aufgenommen werden konnte. Diesen Wert kann man über einen entsprechenden Konstruktor in eine geeignete Komponente der Klasse `Stack_Ueberlauf` unterbringen:

```

class intStack {
private:
    int feld[10];
    int sp;

public:
    // eingebettete Fehlerklassen:
    // allgemeiner Fehler bei Behandlung eines Stacks
    class Stackfehler { };

    // Stackunterlauf, von Stackfehler abgeleitet
    class Stack_Unterlauf: public Stackfehler { };

    // Stackueberlauf, von Stackfehler abgeleitet
    class Stack_Ueberlauf: public Stackfehler {
    public:
        // nicht mehr in den Stack aufgenommener Wert:
        int wert;
        // Konstruktor:
        Stack_Ueberlauf(int a) : wert(a) {}
    };

    // Konstruktor:
    intStack() : sp(0) {}

    // uebliche Stack-Funktionen
    void push(int a)
    { if ( sp >= 10 )
        throw Stack_Ueberlauf(a);

        feld[sp++] = a;
    }
}
  
```

```

    int pop()
    { if ( sp == 0)
        throw Stack_Unterlauf();

        return feld[--sp];
    }
};

```

Einen solchen `intStack` kann man wie folgt verwenden:

```

...
intStack stack;
...
try
{ ...
    stack.push(...);
    ...
    ... = stack.pop();
    ...
}
catch (intStack::Stackfehler a)
{ // Faengt jeden Stackfehler
    ...
}
...

```

Bei diesem `catch` wird nicht zwischen `Stack_Unterlauf` und `Stack_Ueberlauf` unterschieden, sondern jedweder (von `Stackfehler` abgeleiteter) Stackfehlertyp (gleich) behandelt!

Will man Stackfehler vom Typen her unterscheiden, kann man wie folgt vorgehen:

```

...
intStack stack;
...
try
{ ...
    stack.push(...);
    ...
    ... = stack.pop();
    ...
}
catch (intStack::Stack_Unterlauf a)
{ // Faengt Unterlauffehler ab!
    ...
}
catch (intStack::Stack_Ueberlauf a)
{ // Faengt Ueberlauffehler ab, wobei ueber

```



```

// die Komponente wert des Fehlerobjektes
// auf den nicht mehr im Stack untergebrachten Wert
// zugegriffen werden kann:
cerr << "Wert: " << a.wert ;
cerr << " konnte nicht mehr im Stack gespeichert werden!" << endl;
...
}
...

```

7.11 Rein virtuelle Funktionen, abstrakte Basisklassen

Man kann in einer Klasse virtuelle Funktionen einführen, ohne eine Implementierung dieser Funktionen anzugeben — allerdings muss dann hinter der Deklaration, vor dem Semikolon der Zusatz `= 0` (bedeutet: nicht definiert) stehen, Beispiel:

```

class intStack {
public:
    virtual void push(int) = 0;
    virtual int pop() = 0;
};

```

Solche virtuellen Funktionen ohne Implementierung heißen *rein virtuelle Funktionen*. Die Namen und Signaturen dieser virtuellen Funktionen sind bereits eingeführt — die Funktionen selbst sind aber noch nicht definiert. Aufgrund der virtuellen Funktionen sind die Klasse und alle von dieser Klasse abgeleiteten Klassen somit *polymorph*.

Von einer solchen Klasse kann kein konkretes Objekt erzeugt werden, da die Klasse nicht vollständig definiert ist (es fehlt die Definition der rein virtuellen Funktionen) — Zeiger und Referenzen auf diese Klasse können allerdings sehr wohl erzeugt werden:

```

intStack a;           // FEHLER: kann keinen intStack erzeugen!
intStack *p;          // OK, Zeiger auf intStack geht!

void f(intStack);      // FEHLER: intStack kann kein Parameter sein!
void g(intStack &);    // OK, Referenz auf intStack geht!
void h(intStack *);    // OK, Zeiger auf intStack geht!

```

Eine solche Klasse mit mindestens einer rein virtuellen Funktion heißt *abstrakte Basisklasse*, sie stellt ein *abstraktes* Konzept ohne (vollständige) Implementierung dar (hier im Beispiel: abstrakter Typ `intStack` zur Aufnahme ganzzahliger Werte mit der üblichen Schnittstelle `push` und `pop`).

In abgeleiteten Klassen kann man dann die in der abstrakten Basisklasse noch nicht definierten virtuellen Funktionen definieren und somit den abstrakten Typ “realisieren“, wobei man den abstrakten Typ auf unterschiedliche Art realisieren kann, etwa:

- Realisierung des Stacks über ein Feld:

```
class Feld_intStack: public intStack {
protected:
    int feld[100];
    int sp;
public:
    // Konstruktor
    Feld_intStack() { }
    // Realisation der Funktion push
    virtual void push(int wert)
    { ... }
    // Realisation der Funktion pop
    virtual int pop()
    { ... }
};
```

Die Realisierungen der virtuellen Funktionen `push` und `pop` sind an die Feld-Implementierung des Stacks anzupassen. Dadurch, dass die vormalig rein virtuellen Funktionen in dieser Klasse jetzt implementiert sind, ist die Klasse `Feld_intStack` eine konkrete (nicht mehr abstrakte) Klasse und es können Objekte dieser Klasse erzeugt werden.

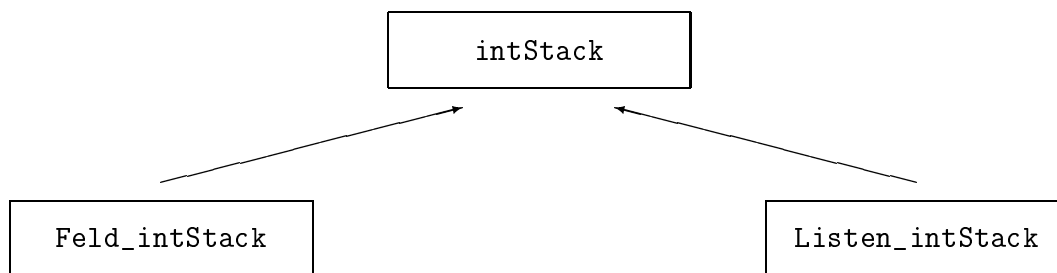
- Realisierung des Stack über eine Lineare Liste:

```
class Listen_intStack: public intStack {
protected:
    struct listel {
        int wert;
        listel *next;
    } *p;

public:
    // Konstruktor
    Listen_intStack() { }
    // Realisation der Funktion push
    virtual void push(int wert)
    { ... }
    // Realisation der Funktion pop
    virtual int pop()
    { ... }
};
```

Die Realisierungen der virtuellen Funktionen `push` und `pop` sind an die Listen-Implementierung des Stacks anzupassen. Dadurch, dass die vormalig rein virtuellen Funktionen in dieser Klasse jetzt implementiert sind, ist die Klasse `Listen_intStack` eine konkrete (nicht mehr abstrakte) Klasse und es können Objekte dieser Klasse erzeugt werden.

Wir haben jetzt folgende Klassenhierarchie, wobei die Basisklasse `intStack` abstrakt ist:



Mittels Polymorphie kann man nun Anwendungen schreiben, in denen “irgendein” `intStack` (per Referenz oder über Zeiger) verwendet wird und es kommt dabei nicht darauf an, welche der “konkreten” `intStack`-Realisierungen tatsächlich dahintersteckt:

```

Feld_intStack fst;    // ist auch ein intStack!
Listen_intStack lst;  // ist auch ein intStack!

void f(intStack &);   // Funktion mit intStack-Referenz-Parameter
...
f(fst);              // rufe Funktion f mit Feld_intStack als Argument auf!
...
f(lst);              // rufe Funktion f mit Listen_intStack als Argument auf!
...

// Definition der Funktion f:
void f( intStack &stack)
{ // verwende stack als intStack, gleichgültig
  // welcher konkrete Stack hinter der
  // Referenz steckt!
  ...
  stack.push(7);
  ...
  erg = stack.pop();
  ...
}
  
```

Eine Abstrakte Basisklasse, in der außer einigen rein virtuellen Funktionen nichts anderes deklariert ist (wie die Klasse `intStack` in obigem Beispiel) heißt auch *reine Schnittstellen-Deklaration* oder *Schnittstellenklasse*.

Hierdurch wird frei von jeder Implementierung “nur” eine Schnittstelle eingeführt, hier im Beispiel die universelle Schnittstelle für einen Stack (für `int`-Werte): für Stacks gibt es die Funktionen `push` und `pop`.

Ein Objekt einer öffentlich von dieser Schnittstellenklasse `intStack` abgeleiteten konkreten Klasse (d.h. die virtuellen Funktionen müssen in der konkreten Klasse komplett implementiert sein) hat (u.a.) diese Schnittstelle und “ist ein” `intStack` und kann, entsprechend der Schnittstelle, wie ein `intStack` verwendet werden.

7.12 Mehrfachvererbung

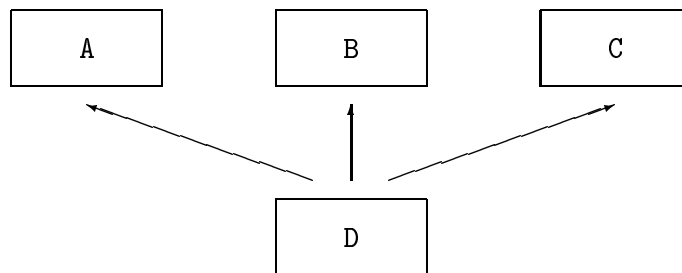
Wie bereits erwähnt, unterstützt C++ Mehrfachvererbung (*multiple Inheritance*) — man leitet aus zwei oder mehreren Basisklassen (direkt) eine neue, abgeleitete Klasse ab:

```
class A { ... };
class B { ... };
class C { ... };

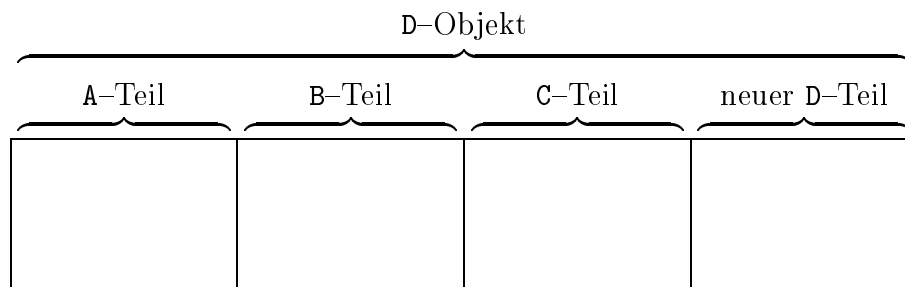
class D : public A, private B, protected C {
...
};
```

Die Klasse D hat die drei direkten Basisklassen A, B und C, wobei hier in diesem Beispiel die Vererbungsarten unterschiedlich sind. (Die gewünschte Vererbungsart muss für jede Basisklasse einzeln angegeben werden — ist keine angegeben, so ist die Vererbungsart nach Voreinstellung **private**, falls die abgeleitete Klasse mit dem Schlüsselwort **class** definiert wird, bzw. **public**, falls die abgeleitete Klasse mit dem Schlüsselwort **struct** definiert wird!)

Schaubild:



Ein D-Objekt hat alle Merkmale (Daten und Funktionen) wie ein A-Objekt, ein B-Objekt und ein C-Objekt — darüberhinaus hat das D-Objekt die neuen Merkmale, welche erst im Klassenrumpf von D deklariert sind:



Durch die Vererbungsart wird wieder gesteuert, in welchem Zugriffsabschnitt in D die geerbten Komponenten der jeweiligen Zugriffsabschnitte der jeweiligen Basisklasse “landen“, siehe Abschnitt 7.2.

Wird ein D-Objekt (durch einen D-Konstruktor) erzeugt, sind natürlich auch ein A-, ein B- und ein C-Konstruktor “fällig“. Diese werden (in dieser Reihenfolge) vor dem Anweisungsteil des D-Konstruktors aufgerufen — und, wenn in der Initialisierungsliste

des D-Konstruktors nichts anderes ausgesagt ist, werden die parameterlosen A-, B- bzw. C-Konstruktoren genommen.

Ist eine der Klassen A, B oder C (oder mehrere von diesen) selbst wieder aus anderen Klassen abgeleitet, so sorgen deren Konstruktoren wieder für den Aufruf der Konstruktoren der jeweiligen Basisklassen!

7.12.1 Beispiel für Mehrfachvererbung

Wie wollen wiederum einen Stack für `int`-Werte implementieren.

Damit wir diesen Stack universell einsetzen können, wollen wir ihn von der abstrakten Basisklasse:

```
class intStack {
public:
    virtual void push(int) = 0;
    virtual int pop() = 0;
};
```

ableiten.

Gleichzeitig wollen wir den Stack mittels der Template-Instantiierung `Vector<int>` realisieren, denn in dieser ist das Stacküberlauf- und Stackunterlaufproblem über Ausnahmen schon so gut wie gelöst:

```
template <class T>
class Vector {
protected:
    T *feld;
    int len;
public:
    // lokale Fehlerklasse:
    struct Feldzugriffsfehler {};

    // Konstruktor
    Vector( int = 10);
    // Copy-Konstruktor (wegen dynamischer Komponente)
    // Vector(const Vector<T> &);
    // Zuweisung (wegen dynamischer Komponente)
    // const Vektor<T>& operator=(const Vector&);
    // Destruktor (wegen dynamischer Komponente)
    virtual ~Vector();

    // Elementzugriff:
    T& operator[](int) throw(Feldzugriffsfehler);
    const T& operator[](int) const throw(Feldzugriffsfehler);
};
```

Mittels dieser beiden Klassen lässt sich der Stack für `int`-Werte sehr einfach realisieren:

```

class intvectorstack: public intStack, protected Vector<int>
{
    int sp;

    public:
        intvectorstack() : sp(0), Vector<int>(100) {}

        virtual void push(int i) { (*this)[sp++] = i;}
        virtual int pop(){ return (*this)[--sp];}
};

```

Erläuterungen:

- Die Klasse `intStack` ist öffentliche Basisklasse, die Template-Instantiierung `Vector<int>` jedoch `protected`, da es sich hierbei um ein Implementationsdetail handelt.
- Im Konstruktor wird der Vektor mit der Länge 100 vereinbart und der Stack-pointer `sp` erhält zunächst den Wert 0.
- Die Stack-Funktionen `push` und `pop` werden über den Indexoperator `operator[]` der Klasse `Vector<int>` realisiert, welche beim Über- bzw. Unterlauf des Vektors eine Ausnahme (`Vector<int>::Feldzugriffsfehler`) auswerfen.

(Diese Implementierung eines Stacks ist in mancherlei Hinsicht noch verbesserungswürdig!)

7.12.2 Namenskonflikte

Das Hauptproblem bei Mehrfachvererbung sind Namenskonflikte: eine Klasse kann in einer Klassenhierarchie mit Mehrfachvererbung einen Namen für eine Komponente über unterschiedliche Vererbungslinien erhalten:

```

class A {
    ...
    public:
        void f(int);
    ...
};

class B {
    ...
    public:
        int f(double);
    ...
};

class C: public A, public B {
    ...
};

```

Wird jetzt für ein Objekt der Klasse `C` die Funktion `f` aufgerufen, steht der Compiler vor einem Problem: ist die `A`-Funktion oder die `B`-Funktion gemeint:

```
C c;
int i;
double x;
...
c.f(i);    // A::f(int) oder B::f(double) ???
c.f(x);    // A::f(int) oder B::f(double) ???
...
```

Hier entscheidet der Compiler nicht anhand der Argumente und den Regeln der Funktionsüberladung, sondern er meldet den Fehler, dass hier zwei gleichwertige Namen vorliegen.

Abhilfe bietet hier nur explizite Qualifikation der Funktion:

```
C c;
int i;
double x;
...
c.A::f(i); // rufe A::f(int) auf
c.B::f(x); // rufe B::f(double) auf
c.A::f(x); // rufe A::f(int) auf, wandle hierzu
           // den double-Wert von x nach int um!
...
```

7.12.3 Mehrfache Basisklassen

Wird eine Klasse `B` von einer Klasse `A` abgeleitet, eine Klasse `C` ebenfalls und schließlich eine Klasse `D` gleichzeitig von `B` und `C`:

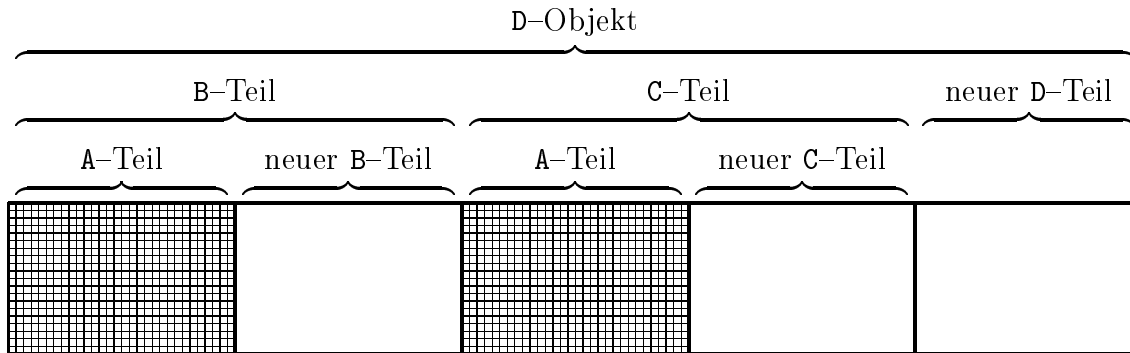
```
class A {
    ...
public:
    void f(void);
    ...
};

class B: public A { ... };
class C: public A { ... };

class D: public B, public C { ... };
```

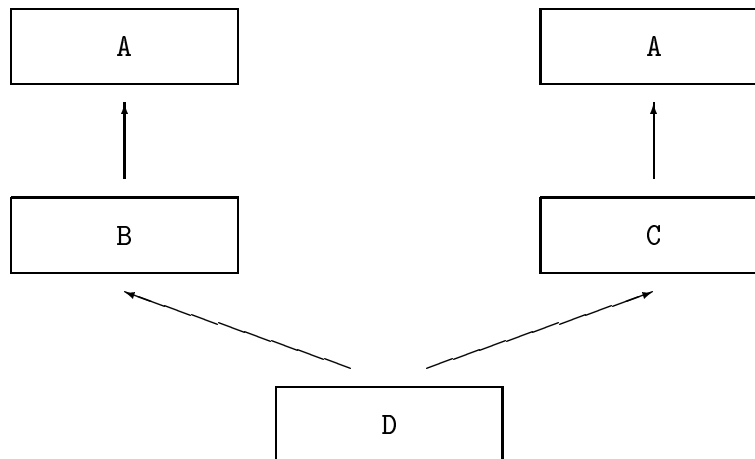
so hat ein `D`-Objekt den geerbten `B`-Teil, in diesem `B`-Teil steckt der in `B` geerbte `A`-Teil, und das `D`-Objekt hat den geerbten `C`-Teil, in dem aber auch ein `A`-Teil vorhanden ist.

Ein D-Objekt verfügt somit über zwei geerbte A-Teile, den einen über B ererbten und den anderen über C ererbten (die beiden A-Teile des D-Objektes sind in folgendem Schaubild schraffiert):



Eine solche mehrfach in einer abgeleiteten Klasse vorhandene Basisklasse heißt *mehrfache Basisklasse* oder *replizierte Basisklasse*.

Zur Verdeutlichung, dass die replizierte Basisklasse mehrfach in einem Objekt der abgeleiteten Klasse vorhanden ist, wird in Schaubildern die Basisklasse ebenfalls mehrfach aufgeführt:



Jede Komponente der mehrfachen Basisklasse ist dann mehrfach in einem Objekt der abgeleiteten Klasse vorhanden — entsprechend muss bei der Erzeugung eines Objektes der abgeleiteten Klasse für jeden mehrfach vorhandenen Basisklassenteil jeweils ein Konstruktor aufgerufen werden (hier im Beispiel wird bei der Erzeugung eines D-Objekts zweimal ein A-Konstruktor aufgerufen!).

Beim Zugriff auf geerbte Komponenten der mehrfachen Basisklasse über das Objekt der abgeleiteten Klasse muss über explizite Qualifikation klargemacht werden, auf welche, über welchem Vererbungsweig geerbte Komponente zugegriffen werden soll:

```
D d;
...
d.B::f();    // ueber B geerbte A-Funktion f aufrufen!
d.C::f();    // ueber C geerbte A-Funktion f aufrufen!
...
```


Beispiel für die Verwendung einer replizierten Basisklasse

Eine replizierte Basisklasse kann durchaus sinnvoll eingesetzt werden:

Im Abschnitt 7.5 über Polymorphie haben wir eine Klasse `Link` zur Realisation einer Linearen Liste definiert (wobei die Listenelemente noch keinen eigentlichen Inhalt haben):

```
class Link {
private:
    Link * next;
public:
    // Konstruktor:
    Link();
    // vorne einfuegen:
    void insert( Link &);

    // vorne entfernen:
    Link * exsert(void);

    // virtuelle Ausgabefunktion:
    virtual void printOn(ostream & ) const;
    // virtueller Destruktor
    virtual ~Link();
};
// globaler Ausgabeoperator:
ostream & operator<<( ostream& , Link &);
```

und hieraus die beiden Klassen `intLink` und `doubleLink` abgeleitet, deren Listenelemente einen `int` bzw. einen `double`-Wert zum Inhalt hatten:

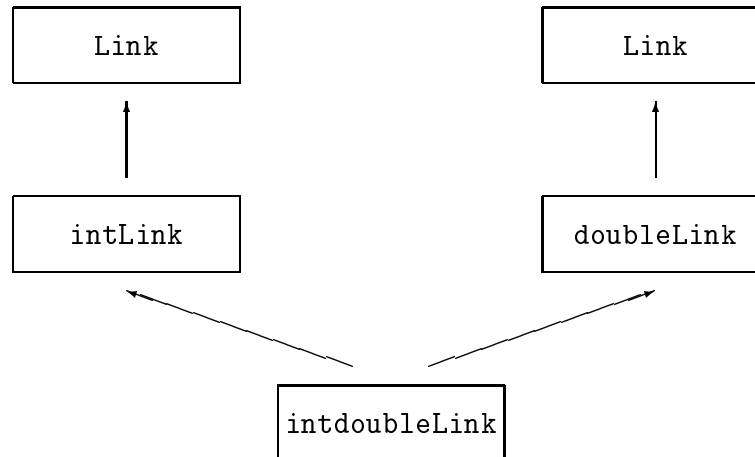
```
class intLink: public Link {
private:
    int wert;
public:
    intLink(int = 0);
    virtual void printOn(ostream& ) const;
};

class doubleLink: public Link {
private:
    double wert;
public:
    doubleLink(double = 0);
    virtual void printOn(ostream& strm) const;
};
```

Wenn wir nun aus diesen beiden Klassen eine neue Klasse `intdoubleLink` ableiten:

```
class intdoubleLink: protected intLink, protected doubleLink { ... };
```

erhalten wir folgende Klassenhierarchie:



Ein `intdoubleLink`-Objekt hat somit zwei `Link`-Teile, kann also zwei Lineare Listen verwalten!

Wir wollen hiermit eine Art von Doppelliste entwickeln, in die sowohl `int`-Werte als auch `double`-Werte eingefügt werden können, wobei die `int`-Werte in einer der Linearen Listen stehen und die `double`-Werte in der anderen, so dass `int` und `double` automatisch getrennt werden.

```

class intdoubleLink: protected intLink, protected doubleLink {
public:
    // int-Werte in die ueber intLink geerbte Liste einfuegen
    void insert(intLink &il)
    { intLink::insert(il); }

    // double-Werte in die ueber doubleLink geerbte Liste einfuegen
    void insert(doubleLink &dl)
    { doubleLink::insert(dl); }

    // Element aus intLink-Liste entfernen
    Link * i_exsert(void)
    { return intLink::exsert(); }

    // Element aus doubleLink-Liste entfernen
    Link * d_exsert(void)
    { return doubleLink::exsert(); }
};
  
```

Dieser kann nun wie folgt verwendet werden:

```
intdoubleLink idl;
```

```
Link *itmp;
Link *dtmp;
```

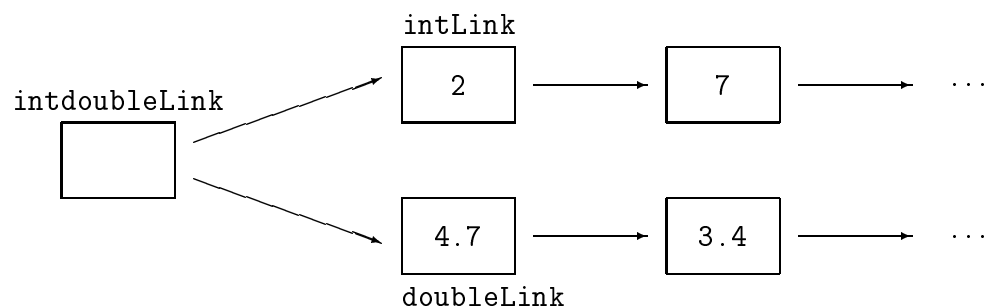
```

Link *tmp;

// IntLink einfuegen
itmp = new intLink(7);
idl.insert(*itmp);
...
// doubleLink einfuegen
dtmp = new doubleLink(3.4);
idl.insert(*dtmp);
...
// intLink herausholen
tmp = idl.i_exsert();
...
// doubleLink herausholen
tmp = idl.d_exsert();
...

```

Veranschaulichung:



(Der Typ `intdoubleLink` ist nicht ganz sauber definiert, da das Objekt `idl` neben den Zeigern `next` auf die Lineare Listen noch zwei weitere, aber nicht verwendete Komponenten `wert` besitzt. Darüberhinaus geht das Einsortieren in die beiden Listen nur statisch über die Typen `intLink` bzw. `doubleLink`, während beim Herausholen ein Zeiger auf ein `Link`-Objekt zurückgeliefert wird!)

7.12.4 Virtuelle Basisklassen

Schauen wir uns die Vererbung

```

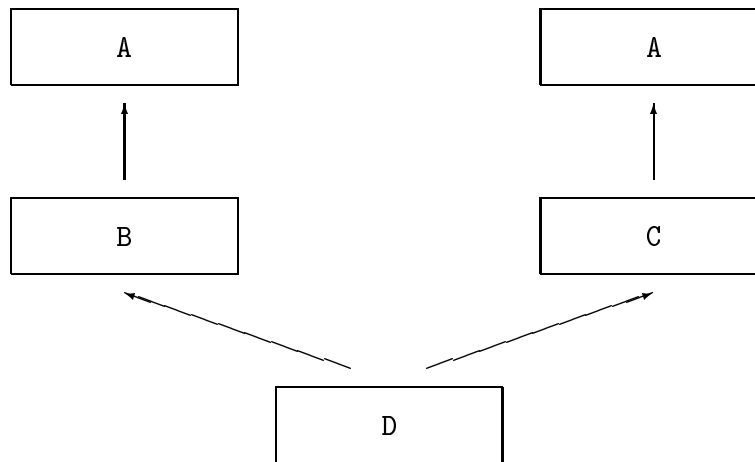
class A { ... };

class B: public A { ... };
class C: public A { ... };

class D: public B, public C { ... };

```

im Schaubild



nochmal an, so ist **A** eine mehrfache Basisklasse in **D** und in **D**-Objekten ist der **A**-Teil mehrfach vorhanden.

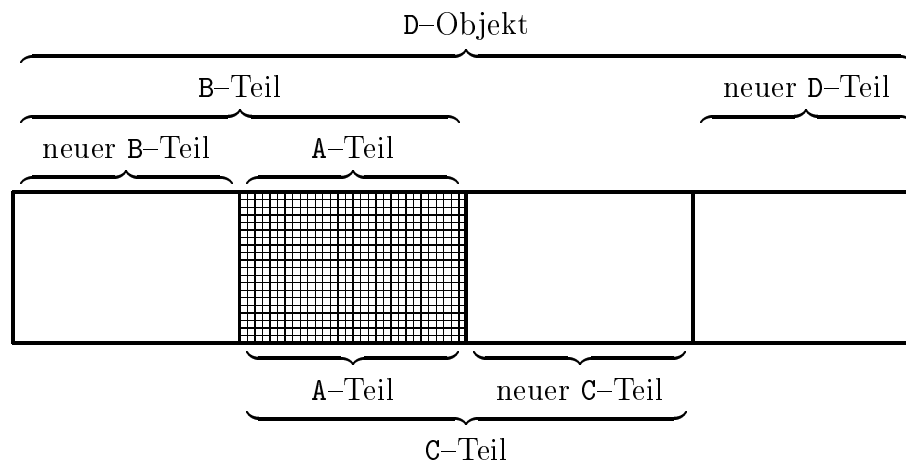
In vielen Anwendungen ist es nicht wünschenswert, dass der **A**-Teil mehrfach in **D**-Objekten vorhanden ist, vielmehr sollten die geerbten **A**-Teile miteinander “identifiziert” werden, d.h. der **A**-Teil ist zwar auf unterschiedlichen Wegen geerbt worden, aber nur einmal in der Klasse **D** vorhanden.

Dies kann dadurch erreicht werden, dass die Klasse **A** *virtuell* an die Klassen **B** und **C** vererbt wird:

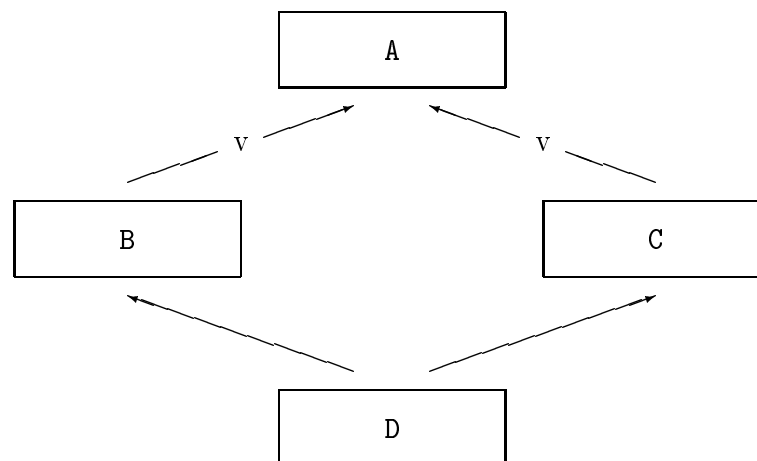
```
class A { ... };  
  
class B: virtual public A { ... };  
class C: virtual public A { ... };  
  
class D: public B, public C { ... };
```

Hierzu ist das Schlüsselwort **virtual** zusätzlich zur Vererbungsart bei der Vererbung anzugeben.

Die Bedeutung ist die, dass, wenn jetzt aus der Klasse **B** bzw. der Klasse **C** weitere Klassen abgeleitet werden, die *virtuell* geerbten **A**-Teile miteinander identifiziert werden:



Im Schaubild der Klassenhierarchie sieht das dann so aus:



Die virtuelle Vererbung wird durch den Buchstaben v in den Pfeilen kenntlich gemacht.

Beispiel für die Verwendung einer virtuellen Basisklasse

Auch die Verwendung virtueller Basisklassen wollen wir an unserem Link-Beispiel demonstrieren:

Wir können aus der Klasse `Link` die beiden Klassen `intLink` und `doubleLink` *virtuell* ableiten:

```
class Link { ... };

class intLink: virtual public Link { ... };
//          ~~~~~

class doubleLink: virtual public Link { ... };
//          ~~~~~
```

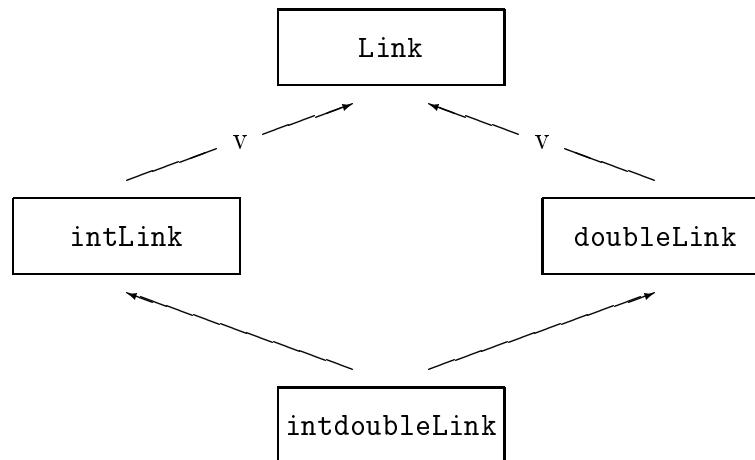
(Definition der Klassen wie oben!) und dann aus diesen beiden Klassen die neue Klasse `intdoubleLink`:

```
class intdoubleLink: public intLink, public doubleLink {
public:
    intdoubleLink(int i=0, double x=0.0) : intLink(i), doubleLink(x) {}
    virtual void printOn(ostream & strm) const
    { strm << " (" << intLink::wert << ',' << doubleLink::wert << ") ";
    }
};
```

Ein `intdoubleLink` ist ein `intLink`, kann also einen `int`-Wert aufnehmen, und ein `doubleLink`, kann also einen `double`-Wert aufnehmen.

In einem `intdoubleLink` gibt es aber nur einen `Link`-Teil, so dass auch nur eine Lineare Liste verwaltet werden kann. Ein `intdoubleLink` stellt also ein Element einer Linearen Liste zur Aufnahme eines `int` und eines `double`-Wertes dar.

Schaubild:



Eine Anwendung könnte wie folgt aussehen:

```
void f(Link &p)
{ Link * tmp = new intLink(7);
  p.insert(*tmp);

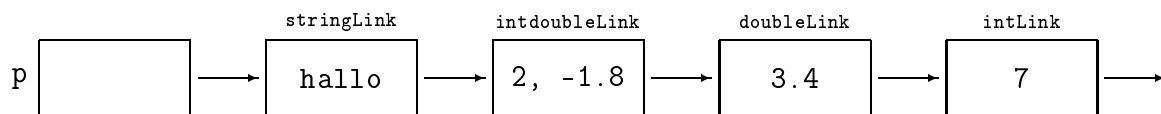
  tmp = new doubleLink(3.4);
  p.insert(*tmp);

  tmp = new intdoubleLink( 2, -1.8);
  p.insert(*tmp);

  tmp = new stringLink("hallo");
  p.insert(*tmp);

  return;
}
```

Am Ende dieser Funktion sollte die Lineare Liste, deren erstes Element die Referenz `p` ist, in etwa wie folgt aussehen:



(Zur Verdeutlichung sind über den einzelnen Listenelementen deren tatsächliche Typen vermerkt!)

7.12.5 Schlussbemerkungen zu mehrfachen und virtuellen Basisklassen

1. Eine Klasse `A` kann nicht mehrfache direkte Basisklasse einer Klasse `B` sein:

```

class A { ... };

class B: public A, public A { ... }; // FEHLER: mehrfache
...                               // direkte Basisklasse!!
  
```

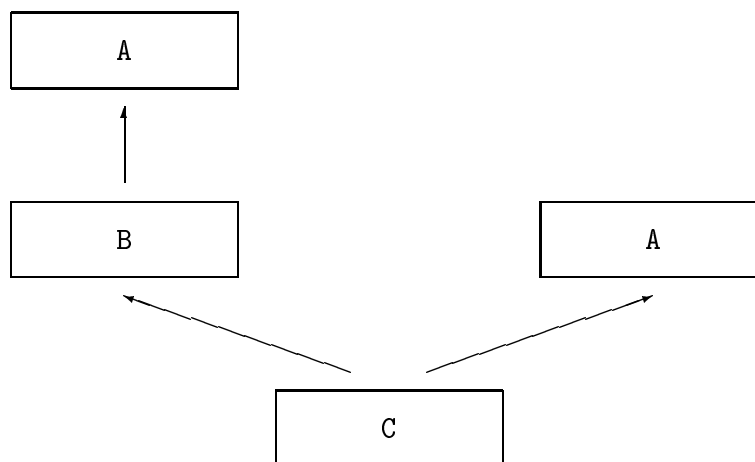
2. Es ist zwar unüblich, aber erlaubt, dass eine Klasse `A` gleichzeitig direkte und indirekte Basisklasse einer anderen Klasse ist:

```

class A { ... };
class B : public A { ... };

class C : public B, public A { ... };
...
  
```

Schaubild:



(Unsere Compiler geben hier allerdings eine Warnung aus!)

3. Hat eine Klasse D eine Klasse A (ggf. auf mehreren Wegen) virtuell geerbt, so muss ein A-Konstruktor in der Initialisierungsliste des D-Konstruktors aufgeführt sein (falls nicht der parameterlose A-Konstruktor genommen werden soll!):

```

class A {
public:
    A() { cerr << "parameterloser A-Konstruktor" << endl;}
    A(int i) { cerr << "int A-Konstruktor" << endl;}
    A(double x) { cerr << "double A-Konstruktor" << endl;}
    A(int i, double x) {cerr << "int-double A-Konstruktor << endl;}
    void f() { cerr << "A-Funktion f aufgerufen" << endl; }
};

class B: virtual public A {
public:
    B() : A(1) {}    // parameterloser B-Konstruktor ruft
                    // int-A-Konstruktor auf!
    void f() { cerr << "B-Funktion f aufgerufen" << endl; }
};

class C: virtual public A {
public:
    C() : A(3.4) {} // parameterloser C-Konstruktor ruft
                    // double-A-Konstruktor auf!
    void f() { cerr << "C-Funktion f aufgerufen" << endl; }
};

class D: public B, public C {
public:
    D() {}
    D(int i): A(i, 3.4) {}
};

int main(void)
{ D d1;    // Ausgabe: parameterloser A-Konstruktor !!!
  D d2(1); // Ausgabe: int-double A-Konstruktor      !!!

  return 0;
}

```

Hier wird bei der Erzeugung des D-Objektes d1 der parameterlose D-Konstruktor aufgerufen und dieser ruft (implizit, da nichts anderes angegeben) die parameterlosen B- und C-Konstruktoren auf! Für die Erzeugung des (virtuellen) A-Teils sind aber nicht die B- und C-Konstruktoren zuständig (diese könnten, wie hier im Beispiel, widersprüchliche Angaben zum A-Konstruktor enthalten), sondern der D-Konstruktor selbst. Da in der Initialisierungsliste des parameterlosen

D-Konstruktors nichts anderes angegeben ist, wird somit der parameterlose A-Konstruktor zur Initialisierung des (virtuellen) A-Teils genommen!

Entsprechend wird bei der Erzeugung des D-Objektes d2 der virtuelle A-Teil mit dem Konstruktor A(int,double) erzeugt, da dieser in der Initialisierungsliste des zur Erzeugung von d2 verwendeten D-Konstruktors D(int) steht!

Ebenso kann auf eine Komponente, etwa eine Member-Funktion, des virtuellen A-Teils (falls explizite Qualifikation aufgrund von Mehrdeutigkeiten erforderlich ist) über den Namen der virtuellen Klasse direkt zugegriffen werden:

```
D d;

d.f();           // FEHLER: Mehrdeutig
d.B::f();        // Funktion void B::f() wird aufgerufen.

// folgende drei Aufrufe sind voellig gleichwertig,
// es wird jeweils die Funktion void A::f() aufgerufen
// und diese arbeitet mit dem virtuellen A-Teil von d:

d.B::A::f();     // Funktion void A::f() wird aufgerufen
d.A::f();        // Funktion void A::f() wird aufgerufen
d.C::A::f();     // Funktion void A::f() wird aufgerufen
...
```

4. Eine Klasse A kann nicht gleichzeitig virtuelle und direkte Basisklasse einer anderen Klasse sein:

```
class A { ... };

class B : virtual public A { ... };
class C : virtual public A { ... };

class D : public B, public C, public A { ... }; // FEHLER:
// gleichzeitig direkte und virtuelle Basisklasse geht nicht!!!
...
```

5. Es ist jedoch wohl möglich, dass eine Klasse A gleichzeitig virtuelle und nicht virtuelle, aber indirekte Basisklasse einer anderen Klasse ist:

```
class A {
    ...
    public:
        void f(void);
    ...
};
```

```

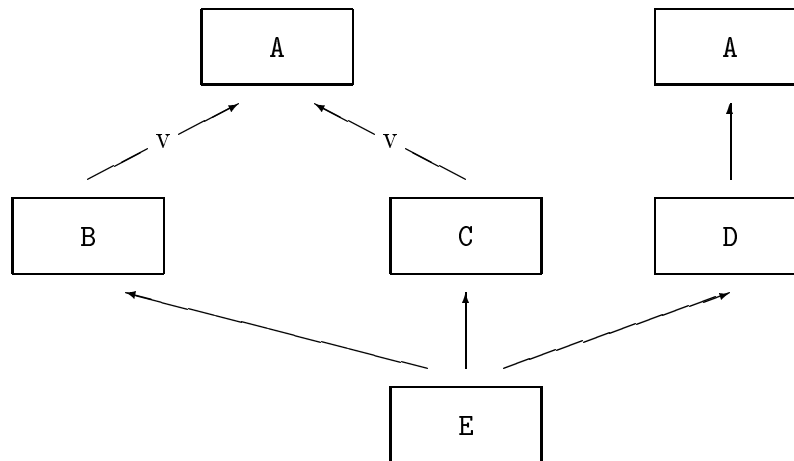
class B: virtual public A { ... };
class C: virtual public A { ... };

class D: public A { ... };

class E: public B, public C, public D { ... };
...

```

Im Schaubild sieht das so aus:



Ein D-Objekt hat zwei A-Teile, der erste A-Teil wird gemeinsam von den B und C-Teilen verwendet, und der zweite A-Teil ist der über D geerbte.

Für die Initialisierung des nicht virtuellen, über D geerbten A-Teils ist — wie immer bei nicht virtuellen Basisklassen — der D-Konstruktor zuständig.

Für die Initialisierung des virtuellen A-Teils ist der E-Konstruktor selbst zuständig.

Die Funktion `void A::f();` muss für ein E-Objekt mittels expliziter Qualifikation aufgerufen werden, da diese zweimal vorhanden ist, einmal für den virtuellen A-Teil und einmal für den nicht virtuellen, über D geerbten A-Teil.

Bei dem nicht virtuellen Teil muss der ganze Vererbungsweig mit angegeben werden, für den virtuellen Teil reicht die Angabe der Klasse A:

```

E e;
e.D::A::f(); // bei diesem Aufruf arbeitet die Funktion f mit dem
              // nicht virtuellen, ueber D geerbten A-Teil von e

// folgende drei Aufrufe sind gleichwertig, die Funktion f arbeitet
// jeweils mit dem virtuellen, ueber B und C geerbten A-Teil von e
e.A::f();
e.B::A::f();
e.C::A::f();
...

```

7.13 Navigieren in Klassenhierarchien

Virtuelle Funktionen und Polymorphie bieten auch im Zusammenhang mit Mehrfachvererbung ein geeignetes Sprachmittel, um mit Objekten vernünftig arbeiten zu können, ohne den genauen, tatsächlichen Typ des Objektes zu kennen.

Mittels der Cast-Operatoren

```
static_cast<Typ>(Ausdruck)
```

```
dynamic_cast<Typ>(Ausdruck)
```

kann man (unter gewissen) Umständen ein Objekt (oder einen Zeiger/eine Referenz auf ein Objekt) eines Types in der Klassenhierarchie in einen andern Typen der Klassenhierarchie “umwandeln”.

Bei allen Arten des Castens sind drei Typen beteiligt:

- der formale Typ des Objektes, welches umgewandelt werden soll (*formaler umzuwandelnder Typ* von *Ausdruck*),
- der (formale) Typ, in den umgewandelt werden soll (*Zieltyp*, steht in den spitzen Klammern),
- und dem tatsächlichen Typ des Objektes, welches umgewandelt werden soll (*tatsächlicher umzuwandelnder Typ*)!

Ist der Zieltyp ein Adresstyp (T^*), so muss der formale, umzuwandelnde Typ ebenfalls ein Adresstyp sein.

Ist der Zieltyp eine Referenztyp ($T\&$) oder ein gewöhnlicher Typ (T), so muss der formale, umzuwandelnde Typ auch ein Referenz oder ein normaler Typ sein.

Bei Adressen und Referenzen brauchen der formale Typ und der tatsächliche Typ des umzuwandelnden Objektes nicht übereinzustimmen:

```
class A { ... };
class B : public A { };
```

```
B b;
A &ar = b;
```

```
dynamic_cast<...>(ar); // formaler Typ von ar ist: A-Referenz
                      // tatsächlicher Typ ist: B-Objekt
```

Bei einem `static_cast` werden nur der formale, umzuwandelnde Typ und der Zieltyp berücksichtigt, bei einem `dynamic_cast` auch der tatsächliche, umzuwandelnde Typ — aber nur, falls der formale, umzuwandelnde Typ ein polymorpher Typ ist (für den also Typen mindestens eine virtuelle Funktion definiert ist) und der Zieltyp ein Adress- oder Referenztyp ist.

Man unterscheidet folgende Arten des “Castens” (Typangaben beziehen sich auf formalen, umzuwandelnden und Zieltyp):

- *Upcast*:
Casten von einer abgeleiteten zu einer (möglicherweise auch indirekten) Basis-klasse (in der Klassenhierarchie von *unten* nach *oben*),

- *Downcast*:

Casten von einer Klasse zu einer (möglicherweise auch indirekt) von dieser Klasse abgeleiteten Klasse (in der Klassenhierarchie von *oben* nach *unten*), und

- *Crosscast*:

Cast zwischen einer Klasse A und einer Klasse B, die weder direkt noch indirekt voneinander abgeleitet sind.

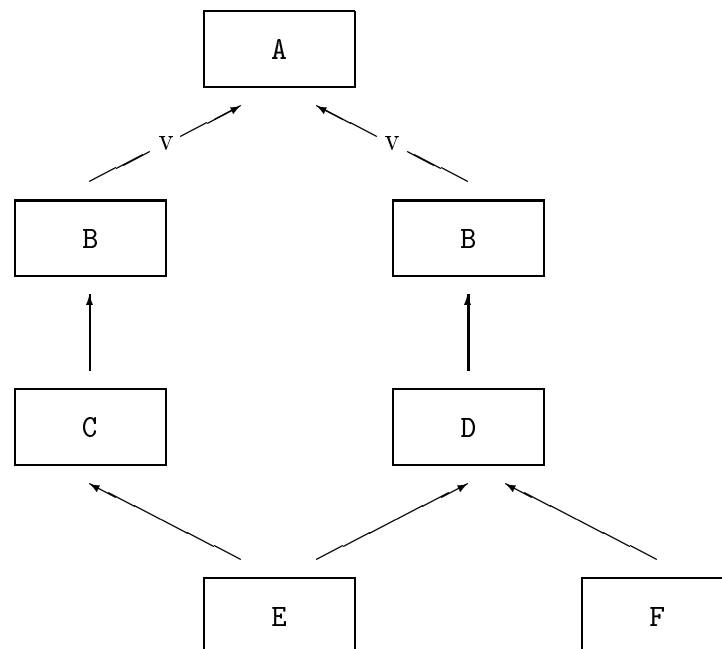
Ein *Crosscast* ist nur mittels `dynamic_cast` möglich und nur dann erfolgreich, wenn es eine Klasse C gibt, die sowohl von A als auch von B (möglicherweise auch indirekt) abgeleitet ist und das tatsächliche umzuwandelnde Objekt diesen Typ C hat.

Ob und in welchem Umfang solche Umwandlungen möglich sind, hängt von der Vererbungsart in der Klassenhierarchie und von ggf. separat definierten Typumwandlungen (über zusätzliche Konstruktoren oder Konversionsoperatoren) ab.

Zu den Beispielen legen wir folgende Klassenhierarchie zugrunde, in der alle Ableitungen `public` sind und keine weiteren benutzerdefinierten Typumwandlungen definiert sind:

```
class A { public: virtual ~A() {...} ... }; // polymorphe Klasse
class B : virtual public A { ... };
class C : public B { ... };
class D : public B { ... };
class E : public C, public D { ... };
class F : public D { ... };
```

Im Schaubild:



Ein paar Regeln (Liste ist nicht unbedingt vollständig!):

1. Ein *Upcast* wird immer (auch bei `dynamic_cast`) vom Compiler statisch umgesetzt und ist immer dann möglich, wenn der formale umzuwandelnde Typ genau einen mit dem Zieltypen übereinstimmenden Teil besitzt:

```
void f( E& e)  // e ist Referenz auf E
{ const C &c1 = static_cast<C>(e);    // OK, E hat genau einen C-Teil
  const B &b1 = static_cast<B>(e);    // FEHLER: E hat zwei B-Teile
  const A &a1 = static_cast<A>(e);    // OK: E hat einen A-Teil

  const C &c2 = dynamic_cast<C>(e);  // OK, E hat genau einen C-Teil
  const B &b2 = dynamic_cast<B>(e);  // FEHLER: E hat zwei B-Teile
  const A &a2 = dynamic_cast<A>(e);  // OK: E hat ein A-Teil
}
```

2. Ist der Zieltyp kein Adresstyp und kein Referenztyp, so ist nur ein *Upcast* möglich!
3. Ist bei einem statischen *Downcast* (`static_cast`) der formale, umzuwandelnde Typ eine virtuelle Basisklasse des Zieltypes, so wird dies vom Compiler als Fehler gemeldet!
4. Ein statisches *Downcast* (`static_cast`) mit einem Referenz- oder Adress-Zieltyp ist nur dann (formal) möglich, wenn der Zieltyp genau einen Teil besitzt, der den formalen, umzuwandelnden Typ hat, ansonsten erhält man eine Compilerfehlermeldung.

```
void f( B& b)  // b ist Referenz auf B
{ const E &e1 = static_cast<E>(b);  // Fehler: B in E nicht
}                                     // eindeutig
```

Diese Typumwandlung ist vom System ungeprüft, d.h. der Programmierer ist selber dafür verantwortlich, dass die umzuwandelnde Referenz oder Adresse zur Laufzeit tatsächlich auf ein Objekt des Zieltypes zeigt, ansonsten ist das Laufzeitverhalten nicht definiert:

```
void f( B& b, C& c)  // b ist Referenz auf B
                   // c ist Referenz auf C
{ B & br = c;

  // Downcast formal ok und liefert zur Laufzeit
  // keine Probleme, da br tatsaechlich auf ein C zeigt!
  const C &cr1 = static_cast<C>(br);

  // Downcast formal ok und liefert zur Laufzeit
  // aber Probleme, da br moeglicherweise nicht
  // auf ein C zeigt!
  const C &cr2 = static_cast<C>(b);
}
```

5. Bei einem dynamischen Cast (`dynamic_cast`) muss der Zieltyp ein Referenz- oder Adresstyp und der formale umzuwandelnde Typ ein polymorpher Typ sein. Hat dann der tatsächliche umzuwandelnde Typ genau einen Teil, dessen Typ mit dem Zieltyp übereinstimmt, geht die Umwandlung in Ordnung, ansonsten nicht.

Falls die Umwandlung nicht “in Ordnung geht”,

- wird bei einem *Upcast* vom Compiler eine Fehlermeldung ausgegeben,
- ansonsten, falls der Zieltyp ein Referenztyp ist, eine `bad_cast`-Ausnahme ausgeworfen,
- ansonsten (Zieltyp ist dann ein Adresstyp) die Zieltyp-Adresse 0 geliefert.

Beispiel:

```
int main(void)
{
    E e;
    C c;

    A *ap = &e;
    B *bp = dynamic_cast<B*> (ap); // mehrfacher B-Teil!
        // Ergebnis 0!

    F *fp = dynamic_cast<F*> (&e); // kein F-Teil
        // Ergebnis 0!

    C *cp = &e;
    D *dp = dynamic_cast<D*>(cp); // funktionierendes Cross-Cast
        // Ergebnis: umgewandelte Adresse!
    cp = &c;
    D *dp = dynamic_cast<D*>(cp); // nicht funktionierendes
        // Cross-Cast, Ergebnis 0!

    return 0;
}
```

7.14 Abstrakte Dienste mittels konkreter Klassen realisieren

Häufig ist es so, dass ein abstrakter Dienst (etwa: ein **Stack**, *Kellerspeicher*) mittels einer konkreten Klasse (etwa einer *Linearen Liste* **LListe**) implementiert wird. (Die Realisierung des **Stack**-Konzeptes mittels einer Linearen Liste führe zur Klasse **LLStack**.)

Meistens könnte der abstrakte Dienst auch mit einer völlig anderen konkreten Klasse (etwa einer Klasse **Vector** für Vektoren) realisiert werden. (Die Realisierung des **Stack**-Konzeptes mittels eines Vektors führe zur Klasse **VStack**.)

Bei der Erzeugung eines konkreten Stacks muss dann eine der Realisierungsmöglichkeiten ausgewählt werden — es muss dann entweder ein `LLStack`- oder ein `VStack`-Objekt erzeugt werden.

Wie schreibt man nun (komplexere) Funktionen, in denen mehrere Stacks benötigt werden und (innerhalb der Funktion) erzeugt werden müssen, wobei erst zur Laufzeit (anhand von konkreten Typen) entschieden wird, ob die Stacks als `VStack`'s oder `LLStack`'s zu erzeugen sind? (Man bedenke: Konstruktoren können nicht virtuell sein!) Dem Beispiel, mit dem wir die hierzu mögliche Technik demonstrieren möchten, liegen folgende beiden konkreten Template-Klassen zugrunde:

1. Klasse `LListe` zu (einfachen) Verwaltung einer Linearen Liste, wobei die einzelnen Listenelemente neben dem Zeiger aufs nächste Element einen eigentlichen Inhalt vom Typ `T` haben:

```
template <class T>
class LListe {
private:
    struct LLelement {
        T wert;
        LLelement * next;
    } *p;

public:
    // eingebettete Fehlerklasse
    struct listenfehler{};

    // Konstruktor parameterlos
    LListe() : p(0) {}

    // Aufgrund dynamischer Komponenten:
    // Copy-Konstruktor
    LListe( const LListe<T> &);
    // Zuweisung
    const LListe<T>& operator=(const LListe<T>& );
    // Destruktor
    virtual ~LListe();

    // vorne eine Element einfuegen
    void insert( const T&);

    // vorne ein Element entfernen und Element zurueckgeben,
    // heirbnei kann bei leerer Liste eine listenfehler-Ausnahme
    // ausgeworfen werden:
    T exsert();
};
```

(Die Realisierung der Funktionen dürfte naheliegend sein!)

2. Klasse `Vector` zur (einfachen) Verwaltung eines Vektors mit Indexüberprüfung (eigentliche Vektorelemente haben den Typen `T`):

```
template <class T>
class Vector {
private:
    T *feld;
    int laenge;
public:
    // eingebette Fehlerklassen
    struct index_zu_klein {};
    struct index_zu_gross {};

    // Konstruktor - auch parameterlos
    Vector(int = 100);

    // Wegen dynamischer Komponenten:
    // Copy-Konstruktor
    Vector( const Vector<T> &);
    // Zuweisung
    const Vector<T>& operator=(const Vector<T>& );
    // Destruktor
    virtual ~Vector();

    // Indexoperatoren, einmal fuer variable und einmal fuer
    // konstante Vektoren.
    // Werfen ggf. Ausnahmen vom Typ index_zu_klein oder
    // index_zu_gross aus:
    T& operator[](int);
    const T& operator[](int) const;
};
```

(Auch hier dürfte die Implementierung der Funktionen naheliegend sein!)

Das abstrakte Konzept des Kellerspeichers für Objekte vom Typ `T` werde durch folgende abstrakte Basisklasse (Schnittstellenklasse) eingeführt:

```
template <class T>
class Stack {
public:
    // eingebettete Fehlerklassen
    struct stackunterlauf {};
    struct stackueberlauf {};

    // Funktionen sind rein virtuell, werden erst in
    // abgeleiteten Klassen konkretisiert:
    virtual void push(const T&) = 0;
```



```

    virtual const T pop() = 0;

    // virtueller Destruktor, da zur Ableitung geeignet:
    virtual ~Stack() {}
};

```

Die Erzeugung eines Stacks wird in folgende, ebenfalls abstrakte Basisklasse “gekapselt“, sie enthält nur eine rein virtuelle Funktion `make_Stack`, welche einen Zeiger auf einen neuen Stack mit Elementtyp `T` zurückgibt:

```

template <class T>
class Stack_maker {
public:
    virtual Stack<T> * make_Stack() = 0;
};

```

Da es bis jetzt noch keinen konkreten Stacktyp gibt, kann es bis jetzt auch noch kein konkretes `Stack_maker`-Objekt geben.

Zwei konkrete `Stack`-Typen werden jetze mit Hilfe der Klassen `LListe` und `Vector` definiert. Da die Realisierung des `Stack`-Konzeptes durch diese konkreten Klassen nur ein Implementationsdetail ist, werden die Klassen `LListe` bzw. `Vector` nur `protected` vererbt. Die Basisklasse `Stack` ist jeweils öffentlich, da die abgeleiteten Klassen `LLStack` und `VStack` jeweils `Stack`'s sein sollen:

1. Aus der (abstrakten) Klasse `Stack` und aus der konkreten Klasse `LListe` wird die (konkrete) Stackklasse `LLStack` abgeleitet und hierbei werden die für `Stack` notwendigen Funktionen auf `LListen`-Operationen zurückführen:

```

template <class T>
class LLStack : public Stack<T>, protected LListe<T> {
public:
    virtual void push( const T& a)
    { insert(a); }

    virtual const T pop()
    { try
      { return exsert();
      }
      catch ( LListe<T>::listenfehler a)
      { // evtl. aufgetretener Listenfehler wird
        // auf passenden Stackfehler abgebildet:
        throw stackunterlauf();
      }
    }
};

```

Zusätzlich wird aus der abstrakten Klasse `Stack_maker` die konkrete Klasse `LLStack_maker` abgeleitet, deren (jetzt nicht mehr virtuelle) Funktion `make_Stack` die Adresse eines konkreten, neuen `LLStack`'s zurückgibt:

```

template <class T>
class LLStack_maker: public Stack_maker<T> {
public:
    Stack<T> * make_Stack() { return new LLStack<T>; }
};

```

2. Analog kann aus der (abstrakten) Basisklasse **Stack** und der konkreten Klasse **Vector** die (konkrete) Stackklasse **VStack** abgeleitet und hierbei die für **Stack** notwendigen Funktionen auf **Vector**-Operationen zurückgeführt werden:

```

template <class T>
class VStack : public Stack<T>, protected Vector<T> {
private:
    int sp;
public:
    VStack(): sp(0), Vector<T>(10) {}

    virtual void push( const T& a)
    { try
      { operator[](sp++) = a;
      }
      catch ( Vector<T>::index_zu_gross a)
      { // evt. aufgetretenen Indexfehler
        // auf passenden Stackfehler abbilden!
        throw stackueberlauf();
      }
    }

    virtual const T pop()
    { try
      { return operator[--sp];
      }
      catch ( Vector<T>::index_zu_klein a)
      { // evt. aufgetretenen Indexfehler
        // auf passenden Stackfehler abbilden!
        throw stackunterlauf();
      }
    }
};

```

Auch hier wird zu dieser konkreten **Stack**-Klasse **VStack** eine konkrete **Stack_maker**-Klasse **VStack_maker** erzeugt, deren (jetzt nicht mehr virtuelle) Funktion **make_Stack** die Adresse eines konkreten, neuen **VStack**'s zurückgibt:

```
template <class T>
class VStack_maker: public Stack_maker<T> {
public:
    Stack<T> * make_Stack() { return new VStack<T>; }
};
```

Sei nun `f` eine Funktion, in welcher eine komplexere Aufgabenstellung gelöst wird und zu deren Lösung mehrere Stacks mit abgespeicherten `int`-Werten benötigt, so kann man dieser Funktion ein `Stack_maker`-Objekt als Argument übergeben und innerhalb der Funktion kann dann mittels dieses Objektes jeweils ein neuer Stack generiert werden:

```
void f(Stack_maker<int> & stackmaker)
{ // erzeuge ueber das stackmaker-Objekt einen
  // neuen int-Stak:
  Stack<int> *stack = stackmaker.make_Stack();

  // verwende den int-Stack:

  try
  {
      ...
      stack->push(i);
      ...
      stack->pop();
      ...
  }
  // Fange moegliche Stack-Fehler ab:
  catch( Stack<int>::stackunterlauf a)
  {
      cerr << "Stackunterlauf" << endl;
  }
  catch( Stack<int>::stackueberlauf a)
  {
      cerr << "Stackueberlauf" << endl;
  }
  ...

  // Stack freigeben:
  delete stack;

  return;
}
```

Beim Aufruf dieser Funktion kann über das konkrete Argument gesteuert werden, ob innerhalb der Funktion mit `VStack`'s oder `LLStack`'s gearbeitet werden soll:

```
...  
// Erzeuge VStack_maker-Objekt:  
VStack_maker<int> vm;  
  
// arbeite in f mit VStack's:  
f(vm);  
...  
// Erzeuge LLStack_maker-Objekt:  
LLStack_maker<int> lm;  
  
// arbeite in f mit LLStack's:  
f(lm);  
...
```

Zu beachten ist, dass zur Zeit des Aufrufs der Funktion `f` noch kein einziger Stack vorhanden ist, sondern diese erst beim Ablauf der Funktion über das `Stack_maker`-Objekt erzeugt werden.

Mittels Polymorphie wird dann anhand des tatsächlichen Types des `Stack_maker`-Objektes (hier sind die Typen `VStack_maker` oder `LLStack_maker` möglich) der (oder die) entsprechend(e) Stack('s) erzeugt.

Eine solche Klasse (wie hier die Klasse `Stack_maker`), deren einzige Aufgabe es ist, Polymorphie bei der Erzeugung von konkreten Objekten zu ermöglichen (anhand des konkreten Types des `Stack_maker`-Objektes wird zur Laufzeit entschieden, mit welcher Art Stack's gearbeitet werden soll) wird manchmal auch *Fabrik* (engl.: *Factory*) genannt.

Teil III

Die Standardbibliothek

Kapitel 8

Ein– Ausgabe

Die I/O-Bibliothek in C++ dient zur Ein– bzw. Ausgabe einzelner *Zeichen* und aus einzelnen Zeichen zusammengesetzter *Zeichenfolgen*.

Heutzutage ist im Zusammenhang mit Internationalen Zeichensätzen nicht mehr ganz so klar, was eigentlich ein *Zeichen* ist.

Standardmäßig gibt es in C++ die beiden Zeichentypen `char` und `wchar_t`, wobei (auf unserem System) für ein `char` genau ein Byte Speicher verwendet wird und für ein `wchar_t` sind es bereits 4 Byte.

Um dem Anwender die Möglichkeit zu geben, eigene Zeichentypen zu entwickeln und diese mit der Standardbibliothek genau so komfortabel verwenden zu können, wie eingebaute Zeichentypen, wird im Standard festgelegt, welche Eigenschaften ein *Zeichentyp* haben muss und die Bibliothek kann dann im Wesentlichen mit jedem Typ gleichartig umgehen, der eben diese Eigenschaften besitzt.

8.1 Der Template-Typ `char_traits<T>`

Jeder Typ `T`, zu dem es eine Spezialisierung des (im Standard definierten) Templates

```
template <class T>
struct char_traits { };    // wirklich leer!
```

gibt, kann von der Standardbibliothek als *Zeichentyp* verwendet werden.

Für die Standard-Zeichentypen `char` und `wchar_t` gibt es in der Standardbibliothek bereits entsprechende Spezialisierungen:

```
template <>
struct char_traits<char> {
... // hier steht eine ganze Menge drin
};

template <>
struct char_traits<wchar_t> {
... // hier steht auch eine ganze Menge drin
};
```

und in den Spezialisierungen sind im Klassenrumpf Komponenten aufzuführen, welche die Gemeinsamkeiten aller *Zeichentypen* ausmachen.

Gemeinsamkeiten aller Zeichentypen sind (u.a.):

- Der Zeichentyp korrespondiert zu einem ganzzahligen Typen und es gibt eine Umwandlung vom Zeichentyp in diesen Ganzzahltypen und umgekehrt.
- Zeichen können verglichen und einander zugewiesen werden.
- Es gibt rudimentäre Operationen für aus Zeichen zusammengesetzte Felder.
- Es gibt rudimentäre Typen im Zusammenhang mit dem Zeichentyp und Streams.
- Es gibt einen Wert im zugehörigen Ganzzahltyp, der nicht zu einem Zeichen des Zeichentyps korrespondiert (Verallgemeinerung von EOF) und Funktionen zur Abfrage dieses Wertes.

Will man einen eigenen Typen, der Name sei CH, als Zeichentyp vereinbaren, muss man für diesen Typen das Template `char_traits` wie folgt spezialisieren:

```
template <>
struct char_traits<CH>    // Spezialisierung fuer Typen CH
{
    typedef CH char_type;    // anderer Name fuer CH
    typedef ... int_type;    // zugehoeriger Ganzzahltyp, anstelle ...
                            // muss der konkrete Ganzzahltyp eingesetzt werden!

    // Zuweisen eines Zeichens b an ein Zeichen a:
    static void assign( char_type & a, const char_type &b){ ... }

    // Umwandlungen vom Zeichentyp zum Ganzzahltyp und umgekehrt:
    static char_type to_char_type( const int_type &i) { ... }
    static int_type to_int_type ( const char_type &c) { ... }

    // Vergleiche:
    static bool eq_int_type( const int_type &i, const int_type &j)
    { ... }
    static bool eq ( const char_type &a, const char_type &b) // gleich
    { ... }
    static bool lt ( const char_type &a, const char_type &b) // kleiner
    { ... }

    // Operationen fuer Felder:

    // verschiebt n Zeichen vom Feld s2 ind Feld s,
    // Felder duerfen ueberlappen:
    static char_type* move (char_type *s, const char_type *s2, size_t n)
    { ... }
```



```

// kopiert n Zeichen vom Feld s2 ind Feld s,
// Felder duerfen nicht ueberlappen:
static char_type* copy (char_type *s, const char_type *s2, size_t n)
{ ... }

// setzt ersten n Zeichen im Feld s auf Zeichen a:
static char_type* assign ( char_type *s, size_t n, char_type a)
{ ... }

// vergleicht die ersten n Zeichen in den Feldern s und s2:
// Ergebnis 0, falls gleich, < 0 falls s lexikographisch vor s2,
// sonst > 0:
static int compare(const char_type *s, const char_type *s2, size_t n)
{ ... }

// gibt Laenge des Feldes zurueck:
static size_t length ( const char_type *s) { ... }

// sucht in den ersten n Zeichen von s nach dem ersten Auftreten
// des Zeichens c, gibt Adresse des Treffers oder 0 zurueck:
static const char_type * find ( const char_type *s, size_t n,
                               const char_type &c)
{ ... }

// Typen fuer Streams:
typedef ... off_type; // Offset in Streams,
                      // fuer ... muss konkreter Typ eingesetzt werden!
typedef ... pos_type; // Position in Streams,
                      // fuer ... muss konkreter Typ eingesetzt werden!
typedef ... stat_type; // Zustand eines Streams,
                      // fuer ... muss konkreter Typ eingesetzt werden!

// EOF:
// liefert Wert von EOF:
static int_type eof() { ... }

// falls i nicht dem EOF entspricht (bezuglich obiger Funktion
// eq_int_type), wird Wert von i zurueckgeliefert, ansonsten ein
// Wert, der bezueglich eq_int_type von i verschieden ist:
static int_type not_eof(const int_type &i) { ... }
};

```

(Natürlich müssen alle Funktionen vernünftig implementiert werden, entweder, wie hier mittels { ... } angedeutet, innerhalb des “Klassenrumpfes, oder außerhalb des Klassenrumpfes dann aber aus Performance-Gründen tunlichst explizit inline!)

Die “Member-Funktionen“ müssen alle als `static` vereinbart werden, damit diese ohne ein konkretes Objekt der Klasse `char_traits<CH>` aufrufbar sind!

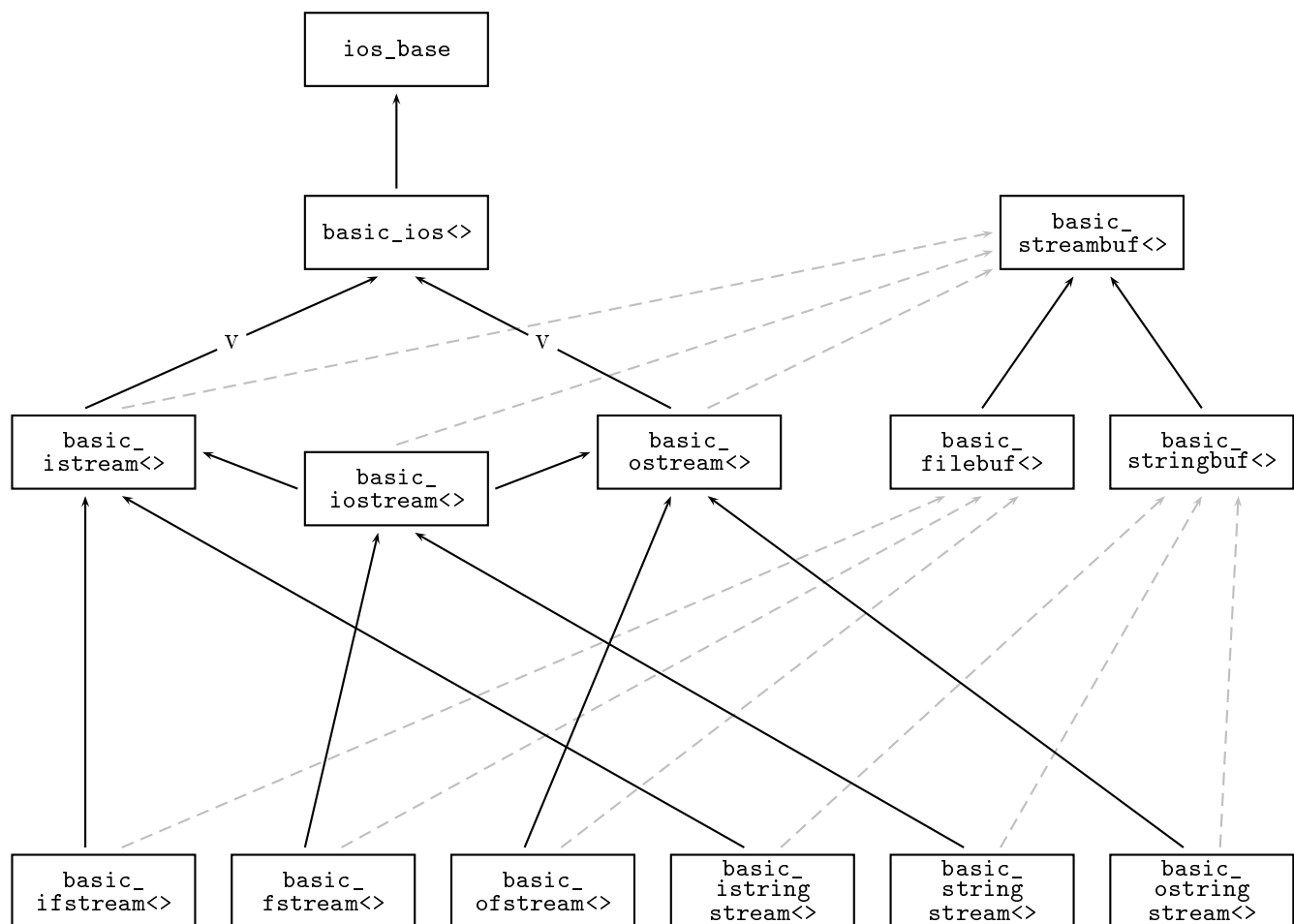
Diese `char_traits` stellen auch gar keine “richtigen“ Klassen dar, sondern sie beschreiben nur die “allgemeinen Merkmale von Zeichen“ für den konkreten Typ `CH`.

Jeder konkrete Typ `CH`, zu dem es die Spezialisierung `char_traits<CH>` gibt, kann “ist“ ein Zeichentyp und kann wie ein Zeichentyp verwendet werden!

8.2 Hintergrund zur Ein- Ausgabe in C++

Das C++-Ein-Ausgabekonzept bedient sich (ziemlich massiv) der durch Objektorientierung (Vererbung und Polymorphie) und generische Programmierung (Templates) möglichen Abstraktionsmechanismen.

Hierzu sieht der Standard eine Reihe von (Template-)Klassen vor, welche in folgender Skizze dargestellt werden. Die durchgehenden schwarzen Pfeile stellen jeweils eine Vererbungsbeziehung dar (Klasse am Ausgangspunkt des Pfeils ist von der Klasse an der Pfeilspitze abgeleitet, virtuelle Ableitung ist wiederum durch ein `v` im Pfeil kenntlich gemacht), die gestrichelten grauen Pfeile besagen, dass die Klasse am Ausgangspunkt des Pfeils ein Objekt der Klasse am Ziel des Pfeils (als Komponente) verwendet. Mit Ausnahme der obersten Klasse `ios_base` sind alle anderen Klassen Templates. Die Template-Klassen haben alle den Präfix `basic_` in ihrem Namen und die Tatsache, dass es sich um Templates handelt, ist durch die spitzen Klammern `<>` im Anschluss an den Namen kenntlich gemacht:



In der Klasse `ios_base` sind Eigenschaften definiert, welche für alle Stream-Klassen prinzipiell gleichartig vorhanden sind, etwa Zustände und Fehlerzustände sowie Möglichkeiten, diese abzufragen und zu beeinflussen. Diese Statusinformationen sind unabhängig von der Zeichenart, die durch den Stream verarbeitet werden soll.

Alle anderen Klassen hängen vom zu verarbeitenden Zeichentyp ab — es sind Template-Klassen, welche durch den entsprechenden Zeichentyp und dessen Eigenschaften (in zug. `char_traits` definiert) parametrisiert sind, alle folgenden Klassen sind entsprechend in folgender Art definiert:

```
template <class CH, class traits = char_traits<CH> >
class ...
```

D.h. durch den ersten Template-Parameter `CH` kann man den zu verarbeitenden Zeichentyp festlegen und durch den zweiten Template-Parameter `traits` (der den Defaultwert `char_traits<CH>` hat) die geforderten Zeicheneigenschaften. (Für ein- und denselben Zeichentyp `CH` könnte man also auch andere, von den Standardeigenschaften `char_traits<CH>` abweichende Eigenschaften — `traits` — verwenden!)

Die "Buffer-Klasse" `basic_streambuf<>` (und die von `basic_streambuf<>` abgeleiteten Klassen `basic_stringbuf<>` und `basic_filebuf<>`) ist für die eigentliche Ein-/Ausgabe auf das entsprechende Medium verantwortlich. Sie umfasst im Wesentlichen einen Puffer für Zeichen (des entsprechenden Zeichentypes) und Informationen

darüber, in wie weit dieser Puffer gefüllt ist. Falls erforderlich wird der Puffer automatisch gefüllt (bei lesenden Zugriffen) bzw. geleert (bei schreibenden Zugriffen). Diese Buffer-Klassen und zugehörigen Funktionen stellen also das Bindeglied zwischen den Streamklassen und den (physikalischen) Ein-/Ausgabemedien her und die Streamklassen “schreiben“ und “lesen“ nur von diesem “Puffer“.

In der Klasse `basic_ios<>` sind die Eigenschaften eines Streams festgelegt, welche spezifisch vom Zeichentyp abhängen, aber davon unabhängig sind, ob der Stream eine Ein- oder Ausgabestream ist.

In der Klasse `basic_istream<>` werden zusätzlich die entsprechenden Eingabe-Operationen definiert und in der Klasse `basic_ostream<>` die Ausgabe-Operationen.

In der Klasse `basic_iostream<>`, welche von `basic_istream<>` und `basic_ostream<>` abgeleitet ist, stehen somit Ein- und Ausgabeoperationen zur Verfügung. (Aufgrund der Tatsache, dass `basic_ios<>` eine virtuelle Basisklasse ist, sind insbesondere Zustandsinformationen — `ios_base` — und allgemeine Formatinformationen — `basic_ios<>` — nur einmal in `basic_iostream<>` enthalten. Darüberhinaus arbeiten in einem `iostream` sowohl die Eingabe- als auch die Ausgabeoperationen mit einem Buffer, so dass auf das durch den Buffer angesprochene Medium gleichzeitig gelesen und geschrieben werden kann!)

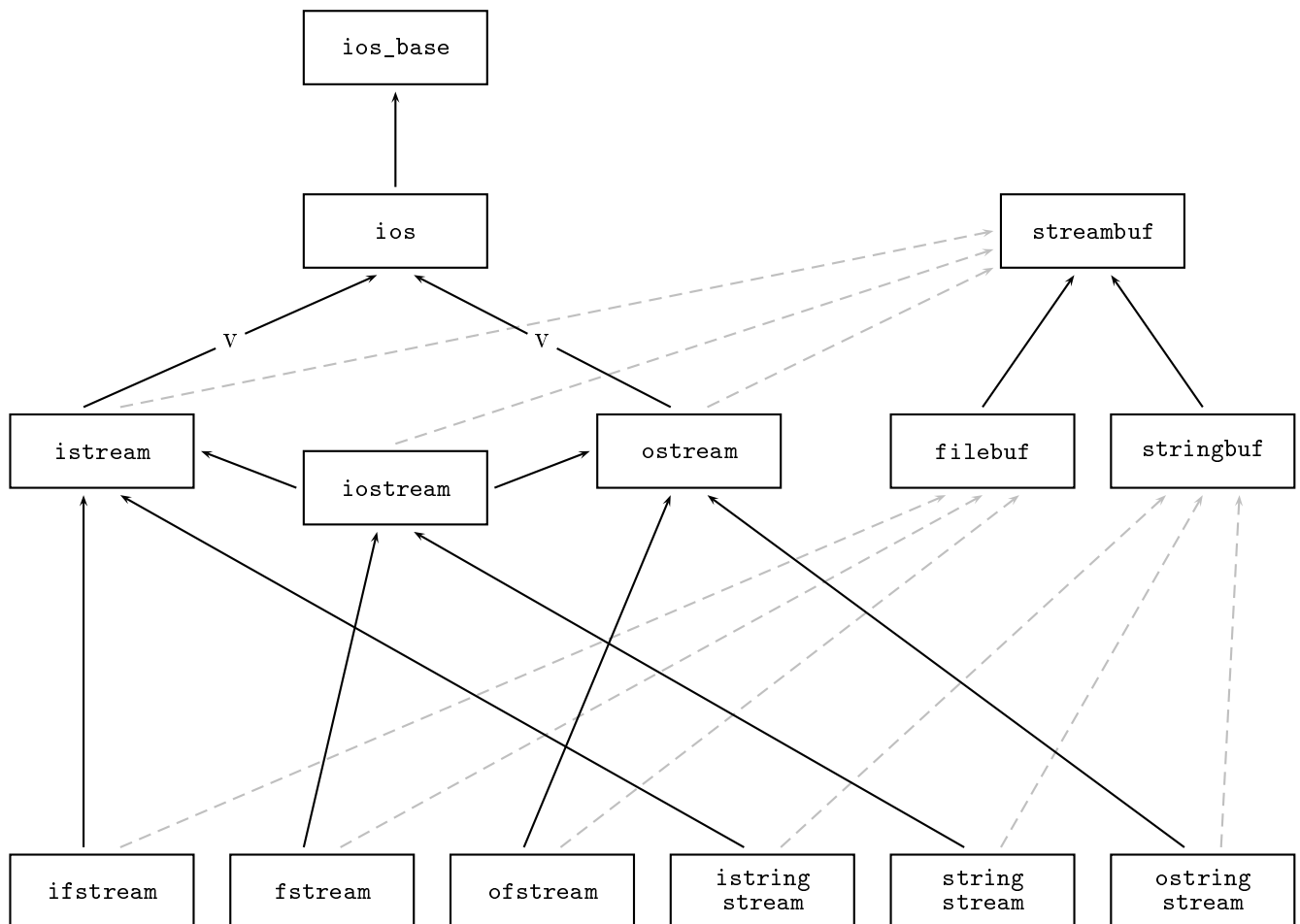
In den File-Stream-Klassen `basic_ifstream<>` (Eingabe), `basic_ofstream<>` (Ausgabe) und `basic_fstream<>` (Ein- und Ausgabe) kommen dann zusätzlich Eigenschaften hinzu, die speziell zur der Dateibehandlung erforderlich sind! Diese verwenden den ebenfalls zur Dateibehandlung spezialisierten, von `basic_streambuf<>` abgeleiteten Buffertyp `basic_filebuf<>`.

Die String-Stream-Klassen `basic_istringstream<>` (Lesen aus Strings), `basic_ostringstream<>` (Schreiben in Strings) und `basic_stringstream<>` (Lesen und Schreiben aus/in Strings) sind speziell auf Strings (Zeichenfelder) zugeschnitten und bedienen sich des ebenfalls für Strings spezialisierten, von `basic_streambuf<>` abgeleiteten Buffertyps `basic_stringbuf<>`. (Da beim Lesen und Schreiben aus/in Strings ggf. zusätzlich dynamische Speicherverwaltung — etwa Vergrößern des Feldes, auf welches ausgegeben wird — erforderlich wird, kann man bei diesen String-Stream-Klassen und der String-Buffer-Klasse neben Zeichentyp und Zeicheneigenschaften (`char_traits`) ein zusätzliches Template-Argument — `allocator` genannt — angeben, über das die dynamische Speicherverwaltung abgewickelt wird. Standardmäßig wird ansonsten hier die dynamische Speicherverwaltung mittels `new`, `new[]`, `delete` und `delete[]` verwendet!)

Zu all diesen Template-Klassen sind im Standard bereits für die Standardzeichentypen `char` und `wchar_t` Instantiierungen vereinbart.

Die Namen dieser für den Zeichentyp `char` und den Eigenschaften `char_traits<char>` aus den Template-Klassen instanziierten Klassen können aus den Namen der entsprechenden Templates gewonnen werden, indem das Namenspräfix `basic_` fortgelassen wird.

Die für `char` vorhandenen konkreten Klassen sind in folgendem Schaubild aufgeführt (die Bedeutung der Pfeile ist wie in der letzten Abbildung):



Im Standard sind bereits einige konkrete Objekte dieser Klassen definiert:

- `istream cin;`
 Standardeingabekanal, Lesen von der Tastatur (entspricht dem `stdin` in C),
- `ostream cout;`
 Standardausgabekanal, Schreiben auf den Bildschirm (i. Allg. gepuffert, entspricht dem `stdout` in C),
- `ostream cerr;`
 Standardfehlerkanal, geschrieben wird (standardmäßig) auch auf den Bildschirm — aber ungepuffert — (entspricht dem `stderr` in C), und
- `ostream clog;`
 Standardprotokollkanal zur Ausgabe von Protokollmeldungen während der Programmlaufs (gibt es in C noch nicht, ist aber i. Allg. gepuffert).

Die für den Zeichentyp `wchar_t` vorhandenen Klassen und Standardobjekte sind völlig analog, ihren Namen ist (von der Klasse `ios_base` abgesehen) jeweils der Buchstabe `w` voranzustellen (also etwa `wistream` für Eingabestrom, `wostream` für Ausgabestrom usw.).

Insbesondere sind die konkreten Objekte `wcout`, `wcin`, `wcerr` und `wclog` zur Standard-ein-/Ausgabe von Tastatur und Bildschirm mit dem Zeichentyp `wchar_t` vorgesehen. Zur Verwendung der Standard- und File-Streamklassen und Objekte muss die Headerdatei `<iostream>` includet werden. Sollen auch die String-Stringklassen gebraucht werden, muss zusätzlich die Headerdatei `<sstream>` eingebunden werden.

Natürlich gehören alle vorgestellten Klassen und Objekte zum Namensbereich `std`, so dass zu deren Verwendung ggf. dieser Namenbereich immer explizit angegeben werden muss (etwa: `std::cout`) oder dieser Namensbereich mittels der `using`-Direktive:

```
using namespace std;
```

generell “einzuschalteten“ ist.

8.3 Fähigkeiten unserer Compiler

Unsere Compiler sind noch nicht ganz so weit mit der Umsetzung dieser vom Standard vorgesehenen Konzepte, insbesondere werden die Klassen noch nicht als Templates realisiert, die für die unterschiedlichen Zeichentypen instantiiert werden können.

Allerdings gibt es aber die für den Zeichentyp `char` vorgesehenen Klassen

- `ios`
Grundlegende Streameinstellungen, Fehlerzustände,
- `istream`
gegenüber `ios` zusätzlich Eingabe-Operationen,
- `ostream`
gegenüber `ios` zusätzlich Ausgabe-Operationen,
- `iostream`
Ein- und Ausgabeoperationen,
- `ifstream`
Lesen von Dateien,
- `ofstream`
Ausgabe auf Dateien,
- `fstream`
Ein- und Ausgabe von bzw. auf eine Datei,
- `istrstream` (anstelle des vom Standard vorgesehenen `istringstream`)
Lesen von Strings,
- `ostrstream` (anstelle des vom Standard vorgesehenen `ostreamstream`)
Schreiben auf Strings,
- `iostrstream` (anstelle des vom Standard vorgesehenen `stringstream`)
Lesen und Schreiben in und auf Strings.

Die Funktionalität dieser Klassen selbst entspricht (weitestgehend) dem Standard, zur Verwendung der String-Streamklassen muss beim GNU-C++-Compiler allerdings (noch) die Headerdatei `<strstream>` (anstelle der vom Standard vorgesehenen Headerdatei `<sstream>`) includet werden.

Der Einfachheit halber beziehe ich mich im Folgenden nur auf die für den Zeichentyp `char` vorgesehenen Klassen. (Die anderen werden von unserem System noch nicht unterstützt, die Funktionalität sollte aber völlig gleich sein!)

Einige der im Folgenden beschriebenen Funktionen, Typen und Werte sind nach Standard bereits in (der auf unseren Systemen noch nicht vorhandenen Klasse) `ios_base` definiert, von der bekanntlich auch die auch auf unseren Systemen vorhandene Klasse `ios` abzuleiten wäre.

Bei der Beschreibung der entsprechenden Dinge habe ich meistens die Klasse `ios_base::` mit aufgeführt, auf unserem System könnte (müsste) `ios_base` entsprechend durch `ios` ersetzt werden!

Bereits in `ios_base` (oder in der hiervon über das Template `basic_ios<>` abgeleiteten Klasse `ios`) definierte Stream-Member-Funktionen können i. Allg. für alle Stream-Objekte aufgerufen werden, hier braucht man in der Anwendung die Funktion nicht explizit über `ios::base` bzw. `ios::` zu qualifizieren, möchte man aber in `ios_base` definierte Konstanten verwenden, so müssen diese explizit (über `ios_base::` bzw. `ios::`) qualifiziert werden!

8.4 Ausgabe

In `iostream` sind der Datentyp (Klasse) `ostream` (Ausgabestrom) und die globale Objekte `cout` (Standardausgabe, entspricht dem `stdout` in C) sowie `cerr` (Standardfehlerausgabe, entspricht also dem `stderr` in C) und `clog` (Standardprotokollausgabe) definiert.

8.4.1 Einfache Ausgabe

Die Funktionalität des Ausgabeoperators `<<` für einen `ostream` haben wir bereits in Abschnitt 3.1.2 kennengelernt.

Neben diesem universellen Ausgabeoperator `<<` gibt es zur Klasse `ostream` noch die Member-Funktionen:

```
- ostream& ostream::put(char);
```

welche wie folgt aufgerufen:

```
cout.put(c);
```

das als Argument angegebene Zeichen (`char c`) auf den angegebenen Stream (`cout`) ausgibt. Das Funktionsergebnis ist wiederum eine Referenz auf den Stream, auf den ausgegeben wird.

```
- ostream& ostream::write ( const char *p, streamsize n);
```

Hierbei ist `streamsize` ein implementierungsabhängiger, ganzzahliger Typ, welcher die natürliche Länge von Bytefolgen in Zusammenhang mit Strömen repräsentiert! Der Datentyp `streamsize` wird bereits in der Klasse `ios` definiert, von welcher alle anderen Ein-/Ausgabeklassen abgeleitet sind. (In den meisten Implementierungen dürfte dieser Typ `streamsize` mit dem Typen `size_t` übereinstimmen!)

Die Funktion `write` gibt von der angegebenen Zeichenkette `p` (unabhängig von einem Stringendezeichen `'\0'`!) genau `n` Zeichen hintereinander aus, etwa:

```
char *s;
int n;
...
cout.write(s, n);
...
```

Die Funktionsaufruf von `write` gibt als Funktionsergebnis wieder (eine Referenz auf) den Stream zurück, für welche sie aufgerufen wurde!

8.4.2 Pufferung der Ausgabe, Manipulatoren

Wie bereits im Abschnitt 3.1.4 erläutert ist in C++ wie in C ist die Standardausgabe (Ausgabe auf `cout`) gepuffert (es wird immer erst dann ausgegeben, wenn ein ganzer Block — systemabhängig 512 Byte o. ä. — an Ausgabe zustandekommen ist).

`cerr` bezieht sich auf die Standardfehlerausgabe und ist nicht gepuffert — d.h. jedes auf `cerr` ausgegebene Zeichen erscheint direkt auf dem Bildschirm.

Der Ausgabepuffer zu einem gepufferten Ausgabestrom (etwa `cout` oder `clog`) kann mit der (zur Klasse `ostream` gehörenden Member-) Funktion

```
ostream& ostream::flush();
```

geleert werden (d.h. der Pufferinhalt wird ausgegeben und anschließend der Puffer geleert!):

```
...
cout << "Hallo\n"; // irgendwelche Ausgabe auf cout, ggf. in Puffer
cout.flush();      // Ausgabepuffer fuer cout ausgeben und leeren
cerr << "Hallo\n"; // ungepufferte Ausgabe auf Standardfehlerkanal
clog << "Hallo\n"; // gepufferte Ausgabe auf Standardfehlerkanal
clog.flush();      // Ausgabepuffer fuer clog ausgeben und leeren
...
```

Zur Ausgabe und Leerung eines Ausgabepuffers gibt es den in Abschnitt 3.1.4 bereits erwähnten, gleichwertigen *Ausgabe-Manipulator* mit demselben Namen `flush`.

Für `ostream`'s gibt es standardmäßig noch

- den `endl`-Manipulator (*endline*),
er bewirkt die Ausgabe eines Zeilenvorschubzeichens `'\n'` und anschließende Leerung des Ausgabepuffers, Beispiel:


```
cout << i << endl << j << endl;
```

Diese Ausgabe entspräche somit:

```
cout << i << '\n' << flush << j << '\n' << flush;
```

- den `ends`-Manipulator (*endstring*),
er bewirkt die Ausgabe eines Stringendezeichens `'\0'` und anschließende Leerung des Ausgabepuffers.

Man kann eigene Manipulatoren definieren, mehr dazu in Abschnitt 8.7.

8.4.3 Formatierung der Ausgabe

Die Ausgabe mittels des Ausgabeoperators `<<` ist i. Allg. unformatiert, d.h.:

- ist `i` eine `int`-Variable, so entspricht die Ausgabe mit

```
cout << i;
```

in C dem Funktionsaufruf:

```
printf("%d",i);
```

- ist `x` eine `double`-Variable, so entspricht die Ausgabe mit

```
cout << x;
```

in C dem Funktionsaufruf:

```
printf("%g",x);
```

In C hat man die Möglichkeit, durch Steuerzeichen und –Flaggen bei einer Formatangabe (u.a.) eine Feldbreite (Mindestanzahl auszugebender Zeichen), Präzision (Anzahl der Nachkommastellen bei Gleitkommawerten), Ausrichtung (rechtsbündig oder linksbündig), Füllzeichen, ..., festzulegen, etwa:

Format	Bedeutung
<code>%+d</code>	Ausgabe eines ganzzahligen Werten, wobei das Vorzeichen immer mit ausgegeben wird.
<code>%10d</code>	Ausgabe eines ganzzahligen Wertes mit mindestens 10 Zeichen (Feldbreite). Bei kleineren Werten wird mit Leerzeichen aufgefüllt (rechtsbündige Ausgabe), bei größeren Werten werden so viele Zeichen wie erforderlich ausgegeben.
<code>%010d</code>	wie <code>%10d</code> , Füllzeichen ist jedoch <code>'0'</code> .
<code>%-15s</code>	linksbündige Ausgabe einer Zeichenkette in einem Feld der Breite 15.
<code>%15.10e</code>	“wissenschaftliche“ Ausgabe eines Gleitpunktwertes mit mindestens 15 Zeichen (Feldbreite), davon 10 Nachkommastellen (Präzision).
<code>:</code>	<code>:</code>

In C++ wird zu einem Ein-/Ausgabestrom dessen “Zustand“ verwaltet, in dem man (u.a.) auch derartige Formatierungen, Präzision, Füllzeichen, ..., dauerhaft einstellen kann.

(In den Beispielen wird hier jeweils der Strom `cout` angegeben. Es könnte aber auch jeder anderer Strom, etwa `cerr` oder `clog` oder auch ein mit einer Datei verknüpfter, selbstdefinierter Ausgabestrom — siehe Abschnitt 8.8 — angegeben werden!)

Die Feldbreite kann hier auch eingestellt werden — die Feldbreite bezieht sich jedoch immer nur auf die **nächste Ausgabe** eines numerischen Wertes oder einer Zeichenkette.

– Die Feldbreite kann wie folgt erfragt und eingestellt werden:

- Erfragen der Feldbreite durch Aufruf der Member-Funktion:

```
streamsize ios_base::width();
```

also etwa:

```
streamsize i = cout.width();
```

- Setzen der Feldbreite durch Aufruf der Member-Funktion:

```
streamsize ios_base::width(streamsize n);
```

also etwa auf den Wert 10 mit dem Aufruf:

```
cout.width(10);
```

Funktionsergebnis ist der bisher gültige Wert. Das Setzen der Feldbreite gilt nur für die nächste Ausgabe!

(Funktionsüberladung: die Funktion `width` kann zur Abfrage der Feldbreite ohne Argument aufgerufen werden und zum Setzen der Feldbreite mit einem Argument vom Typ `streamsize`.)

- Setzen der Feldbreite mit dem Manipulator `setw(streamsize)` (da dieser Manipulator aber ein Argument hat, muss die Standard-Headerdatei `<iomanip>` includet werden, siehe Abschnitt 8.7):

```
cout << setw(10) << i << setw(20) << j << endl;
```

(Auch solche Feldbreiteneinstellungen mittels Manipulatoren beziehen sich nur auf die jeweils nächste Ausgabe!)

- Setzen der Feldbreite auf den Standardwert (soviel Zeichen wie notwendig) durch Aufruf der Funktion (oder des Manipulators) mit Argument 0:

```
cout.width(0);
```

oder

```
cout << setw(0);
```

– Die Präzision (Voreinstellung ist 6) kann mittels der Member-Funktion

```
streamsize ios_base::precision(streamsize);
```

erfragt und (dauerhaft, nicht nur für die nächste Ausgabe) eingestellt werden:

- Erfragen der Präzision durch Aufruf ohne Argument:

```
streamsize i = cout.precision();
```

- Setzen der Präzision (etwa auf den Wert 10):

```
cout.precision(10);
```

(Funktionsergebnis ist der bisher gültige Wert.)

- Alternativ:

Setzen der Präzision mit dem Manipulator `setprecision(streamsize)` (da dieser Manipulator auch ein Argument hat, muss wiederum die Standard-Headerdatei `<iomanip>` includet werden, siehe auch Abschnitt 8.7):

```
cout << setprecision(10) << 1./3. << endl;
```

- Das Füllzeichen kann mittels der Member-Funktion

```
char ios::fill(char);
```

(erst in `ios` bzw. `basic_ios<>` definiert!) wie folgt erfragt und (dauerhaft, nicht nur für die nächste Ausgabe) eingestellt werden:

- Erfragen der Füllzeichens durch Aufruf ohne Argument:

```
char c = cout.fill();
```

- Setzen des Füllzeichens (etwa auf das Zeichen Unterstrich-Zeichen `'_'`):

```
cout.fill('_');
```

- Alternativ:

Setzen des Füllzeichens mit dem Manipulator `setfill(char)` (da dieser Manipulator auch ein Argument hat, muss wiederum die Standard-Headerdatei `<iomanip>` includet werden, siehe auch Abschnitt 8.7):

```
cout << setfill('_') << setw(10) << 3 << endl;
```

Die weiteren (Ein-/Ausgabe-)Eigenschaften eines Stroms werden in einem zum betreffenden Strom gehörenden Objekt vom (systemabhängigen, i. Allg. aber ganzzahligen, bereits in `ios_base` definierten) Datentyp `fmtflags` abgespeichert.

Von diesem Datentyp `fmtflags` sind in `ios_base` einige Konstanten (“Flaggen“) definiert, welche einzeln oder in Gruppen einen Zustand repräsentieren.

Zur Einstellung kann eine der Funktionen

```
fmtflags ios_base::setf(fmtflags);
fmtflags ios_base::setf(fmtflags, fmtflags);
void ios_base::unsetf(fmtflags);
```

verwendet werden.

Die Funktion `setf` dient zum zusätzlichen Setzen von Flaggen — Funktionsergebnis ist der vorherige Zustand des Streams.

Die Funktion `unsetf` dient zum Zurücksetzen von Flaggen.

Es können folgende Zustände beeinflusst werden:

- Ausrichtung innerhalb der Feldbreite.
Hier gibt es drei mögliche Einstellungen:

- Rechtsbündig.
Dieser (voreingestellte) Zustand kann durch die Memberfunktion `setf` mit folgenden Argumenten eingestellt werden:

```
cout.setf(ios_base::right, ios_base::adjustfield);
```

Alternativ ist diese Einstellung auch über den Manipulator `right` möglich:

```
cout << right;
```

- Linksbündig.
Dieser Zustand kann wie folgt durch `setf`:

```
cout.setf(ios_base::left, ios_base::adjustfield);
```

geschehen. Alternativ ist auch hier die Einstellung über einen Manipulator möglich:

```
cout << left;
```

- Vorzeichen linksbündig, Wert rechtsbündig (dazwischen Füllzeichen).
Dieser Zustand kann wie folgt durch `setf`:

```
cout.setf(ios_base::internal, ios_base::adjustfield);
```

geschehen. Alternativ ist auch hier die Einstellung über einen Manipulator möglich:

```
cout << internal;
```

Die drei “Flaggen“ `ios_base::left`, `ios_base::right` und `ios_base::internal` gehören zu einer “Gruppe“ von Flaggen, von denen i. Allg. genau eine “gesetzt“ sein muss. Diese Gruppe hat den Namen `ios_base::adjustfield` und muss bei `setf` als zweites Argument angegeben werden, damit etwa beim “Setzen“ der Flagge `ios_base::internal` die bislang gesetzte Flagge (`ios_base::right` oder `ios_base::left`) “zurückgesetzt“ wird.

- Art der Ausgabe ganzzahliger Werte.
Auch hier gibt es drei mögliche Einstellungen:

- Dezimal.
Dieser (voreingestellte) Zustand kann durch die Memberfunktion `setf` mit folgenden Argumenten eingestellt werden:

```
cout.setf(ios_base::dec, ios_base::basefield);
```

Alternativ über Manipulator:

```
cout << dec;
```

- Oktal.
Dieser Zustand kann durch die Memberfunktion `setf` wie folgt:

```
cout.setf(ios_base::oct, ios_base::basefield);
```

oder alternativ über Manipulator:

```
cout << oct;
```

eingestellt werden.

- Hexadezimal.

Dieser Zustand kann durch die Memberfunktion `setf` wie folgt:

```
cout.setf(ios_base::hex,ios_base::basefield);
```

oder alternativ über Manipulator:

```
cout << hex;
```

eingestellt werden.

Die drei “Flaggen” `ios_base::dec`, `ios_base::oct` und `ios_base::hex` gehören zu einer “Gruppe” von Flaggen, von denen i. Allg. wiederum genau eine “gesetzt” sein muss. Diese Gruppe hat den Namen `ios_base::basefield` und muss bei `setf` als zweites Argument angegeben werden, damit etwa beim “Setzen” der Flagge `ios_base::hex` die bislang gesetzte Flagge (`ios_base::dec` oder `ios_base::oct`) “zurückgesetzt” wird.

Die Einstellung der Basis der Ausgabe kann auch über den Manipulator `setbase` erfolgen:

```
cout << setbase(n);
```

wobei `n` die gewünschte Basis (also 10, 8 oder 16) ist. (Zur Verwendung dieses Manipulators mit einem Argument muss wiederum die Headerdatei `<iomanip>` includet werden!)

- Art der Ausgabe von Gleitkommazahlen.

Es gibt zwei Arten der Ausgabe von Gleitkommazahlen:

- “Wissenschaftlich”, d.h. mit Exponent, etwa `1.2345678E4` (für `1234.5678`). Diese Ausgabeform kann durch `setf`:

```
cout.setf(ios_base::scientific,ios_base::floatfield);
```

oder über den Manipulator:

```
cout << scientific;
```

eingestellt werden.

- Festkommaschreibweise, d.h. ohne Exponent, etwa `1234.5678` (für `1234.5678`). Diese Ausgabeform kann durch `setf`:

```
cout.setf(ios_base::fixed,ios_base::floatfield);
```

oder über den Manipulator:

```
cout << fixed;
```

eingestellt werden.

Die zwei “Flaggen” `ios_base::scientific` und `ios_base::fixed` gehören zu einer “Gruppe” von Flaggen, von denen maximal eine “gesetzt” sein kann. Diese Gruppe hat den Namen `ios_base::floatfield` und muss bei `setf` als zweites Argument angegeben werden, damit etwa beim “Setzen” der Flagge `ios_base::scientific` die evtl. bislang gesetzte Flagge (etwa `ios_base::fixed`) “zurückgesetzt” wird.

Bei der Ausgabe von Gleitkommawerten ist die Voreinstellung die, dass bei “kleinen” Gleitpunktzahlen (etwa zwischen 10^{-4} und 10^{+5}) die Festkomm Schreibweise gewählt wird, und ansonsten die “wissenschaftliche” Schreibweise! (D.h. standardmäßig ist weder `ios_base::fixed` noch `ios_base::scientific` gesetzt!)

Die Rückeinstellung auf diesen Standard kann durch:

```
cout.unsetf(ios_base::floatfield);
```

erfolgen.

- Bei der Ausgabe von oktalen oder hexadezimalen Werten kann die Ausgabe mit führenden 0 (oktal) bzw. 0x (hexadezimal) erfolgen oder nicht.

Die Voreinstellung ist die, dass diese führenden Zeichen nicht ausgegeben werden!

- Einschalten von “*Führendes 0 bzw. 0x ausgeben*”:

```
cout.setf(ios_base::showbase);
```

oder über Manipulator:

```
cout << showbase;
```

- Ausschalten von “*Führendes 0 bzw. 0x ausgeben*”:

```
cout.unsetf(ios_base::showbase);
```

oder über Manipulator:

```
cout << noshowbase;
```

- Bei der Ausgabe von Gleitpunktwerten kann die Ausgabe eines Dezimalpunktes erzwungen werden (auch bei zufälligerweise ganzzahligen Gleitpunktwerten)! Voreinstellung ist, dass der Dezimalpunkt nicht zwangsweise ausgegeben wird.

- Einschalten von “*Dezimalpunkt immer ausgeben*”:

```
cout.setf(ios_base::showpoint);
```

oder über Manipulator:

```
cout << showpoint;
```

- Ausschalten von “*Dezimalpunkt immer ausgeben*”:

```
cout.unsetf(ios_base::showpoint);
```

oder über Manipulator:

```
cout << noshowpoint;
```

- Bei der Ausgabe von numerischen Werten kann die Ausgabe eines Vorzeichens auch bei positiven Werten erzwungen werden. Voreinstellung ist, dass das Vorzeichen nicht zwangsweise ausgegeben wird.

- Einschalten von “*Vorzeichen immer ausgeben*”:

```
cout.setf(ios_base::showpos);
```

oder über Manipulator:

```
cout << showpos;
```

- Ausschalten von “*Vorzeichen immer ausgeben*“:

```
cout.unsetf(ios_base::showpos);
```

oder über Manipulator:

```
cout << noshowpos;
```

- Bei der Ausgabe von numerischen Werten werden ggf. auftretende Buchstaben (Exponent oder hexadezimale Ziffern oder `0x` in Zusammenhang mit hexadezimaler Ausgabe und `ios_base::showbase`) mit kleinen Buchstaben ausgegeben. Hier kann man auf Großbuchstaben umstellen.

- Umstellen auf Großbuchstaben:

```
cout.setf(ios_base::uppercase);
```

oder über Manipulator:

```
cout << uppercase;
```

- Zurückstellen auf Kleinbuchstaben:

```
cout.unsetf(ios_base::uppercase);
```

oder über Manipulator:

```
cout << nouppercase;
```

- Auch das Pufferungsverhalten des Stromes kann man einstellen, durch:

```
cout.setf(ios_base::unitbuf);
```

wird eingestellt, dass der zu `cout` gehörende Ausgabepuffer nach jedem Ausgabebefehl ausgegeben und geleert wird (ungepufferte Ausgabe).

Durch:

```
cout.unsetf(ios_base::unitbuf);
```

wird die Pufferung wieder eingeschaltet!

(Bei `cout` und `clog` ist standardmäßig die Pufferung eingeschaltet, d.h. die Flagge `ios_base::unitbuf` ist hier standardmäßig nicht gesetzt — und bei `cerr` ist standardmäßig die Pufferung ausgeschaltet, d.h. hier ist `ios_base::unitbuf` standardmäßig gesetzt!)

Der gesamte Ein-/Ausgabezustand eines Stroms ist wie gesagt in einem Objekt vom in der Klasse `ios_base` definierten (ganzzahligen) Typ `fmtflags` abgelegt.

Den Wert dieses Objektes kann man als Ganzes mit der Element-Funktion

```
fmtflags ios_base::flags(fmtflags);
```

in Erfahrung bringen (oder auch setzen):

```

...
// Einstellungen sichern durch Aufruf von
// flags ohne Argument:
ios_base::fmtflags zustand = cout.flags();
...
cout.setf(...);          // Einstellungen mittels setf oder
cout << hex;              // Manipulatoren aendern
...
cout << oct;
...
// urspruengliche Einstellungen restaurieren durch Aufruf
// von flags mit dem Argument vom Typ fmtflags, in dem
// der urspruengliche Zustand abgespeichert wurde:
cout.flags(zustand);
...

```

Da der Typ `ios_base::fmtflags` ganzzahlig ist, kann man mittels Bit-Operationen (vornehmlich `|`, `~` und `&`) die Einstellung auch über diese Funktion `flags` vornehmen, etwa:

```

...
ios_base::fmtflags alt, neu;
// bisherige Einstellungen sichern
alt = cout.flags();
// neuen Zustand zusammenbasteln:
neu = (alt | ios_base::hex) & ~ios_base::dec;    // hex hinzu und
                                                // dec entfernen
neu = (neu | ios_base::left) & ~ios_base::right; // linksbuendig hinzu
                                                // und rechtsbuendig entfernen
neu = neu | ios_base::showbase;                // Basis des Zahlensystems ausgeben
// neuen Zustand einstellen
cout.flags(neu);
...
// urspruengliche Einstellungen restaurieren
cout.flags(alt);
...

```

Auch mit `setf` und `unsetf` kann man durch Bit-Operationen gleich mehrere Einstellungen vornehmen (bei `setf` “Flaggen“ hinzufügen und bei `unsetf` “Flaggen“ löschen):

```

// Flaggen hinzufuegen
cout.setf(ios_base::showbase | ios_base::uppercase);
...
// Flaggen loeschen
cout.unsetf(ios_base::showbase | ios_base::uppercase);
...

```


Doch bei solchen Bit-Manipulationen in Zusammenhang mit den Funktionen `flags`, `setf` bzw. `unsetf` muss man bei “Gruppen” von “Flaggen” (`ios_base::adjustfield`, `ios_base::basefield` und `ios_base::floatfield`) vorsichtig sein! Hier muss man selbst dafür sorgen, dass von den beteiligten Flaggen jeweils höchstens (`floatfield`) bzw. genau eine (`basefield` und `adjustfield`) “gesetzt” ist!

Ist `zustand` ein Wert vom Typ `ios_base::fmtflags`, so entspricht der Manipulator

```
cout << setiosflags( zustand);
```

dem Funktionsaufruf

```
cout.setf( zustand);
```

und der Manipulator

```
cout << resetiosflags( zustand);
```

dem Funktionsaufruf

```
cout.unsetf( zustand);
```

(Aufgrund der Argumente ist die Headerdatei `<iomanip>` zu includen!)

8.5 Eingabe

In `iostream` ist der Datentyp (Klasse) `istream` (Eingabestrom) und das globale Objekt `cin` (entspricht dem `stdin` aus C) definiert.

8.5.1 Elementfunktionen zur Eingabe

Neben dem in Abschnitt 3.1.3 kennengelernten universellen Eingabe-Operator `>>` stehen zum Einlesen von einzelnen Zeichen oder Zeichenketten weitere Elementfunktionen der Klasse `istream` zur Verfügung:

```
int istream::get();
```

liefert (wie `getc` in C) den Wert des nächsten gelesenen Zeichens oder EOF beim Ende der Eingabe. Zwischenraumzeichen werden nicht überlesen! (Es handelt sich um eine Elementfunktion und somit sieht der Aufruf wie folgt aus: `int i = cin.get();`.)

```
istream& istream::get(char &c);
```

(Funktionsüberladung) Liest ein Zeichen und speichert es im (Referenz-) Argument `c` ab! Als Funktionsergebnis wird — wie bei den meisten folgenden Funktionen — eine Referenz auf den Stream, für welchen die Funktion aufgerufen wurde, zurückgegeben. An diesem Ergebnis kann man erkennen, ob das Lesen geklappt hat (siehe Abschnitt 8.6)!

```
istream& istream::get( char *p, streamsize n );
```

liest (höchstens) `n-1` Zeichen und speichert sie im Feld `p` ab. Sollte innerhalb der nächsten `n-1` Zeichen ein Zeilenvorschubzeichen `'\n'` sein, so wird der Lesevorgang vor diesem `'\n'` abgebrochen — dieses `'\n'` wird also nicht mehr gelesen! In jedem Fall wird aber noch ein `'\0'` angehängt.

```
istream& istream::get( char *p, streamsize n, char ende);
```

wie `get(char *p, streamsize n);`; nur dass das als dritte Argument angegebene Zeichen `ende` die Rolle des `'\n'` übernimmt!

`istream& istream::getline (char *p, streamsize n);`

wie `get(char *p, streamsize n);`; nur dass ein Zeilenendezeichen `'\n'` ggf. gelesen, aber nicht im Feld `p` abgespeichert wird! (Wird durch das `'\0'` überspeichert!)

`istream& istream::getline (char *p, streamsize n, char ende);`

wie `getline(char *p, streamsize n);`; nur dass das als drittes Argument angegebene Zeichen die Rolle des Zeilenendezeichens `'\n'` übernimmt!

`istream& istream::read (char *p, streamsize n);`

liest `n` Zeichen (auch über Zeilenenden hinweg), hängt kein `'\0'` an. (Beim Ende der Eingabe können auch weniger Zeichen gelesen werden!)

Der Standard sieht (u.a.) noch folgende weiteren Elementfunktionen vor, mit denen man die Eingabe beeinflussen kann:

`int istream::peek();`

Wie `int get();` liefert `peek` den Wert des nächsten in der Eingabe stehenden Zeichens (oder EOF), aber im Gegensatz zu `get` bleibt dieses Zeichen auf der Eingabe stehen — wird also strenggenommen gar nicht gelesen!

`istream& istream::unget();`

Schreibt das zuletzt gelesene Zeichen zurück in die Eingabe (es muss vorher mindestens ein Zeichen gelesen worden sein!).

`istream& istream::putback(char c);`

Schreibt das Zeichen `c` auf die Eingabe zurück! (Der Standard sagt, dass dieses Zeichen wirklich das zuletzt gelesene Zeichen sein muss!)

Die Funktion

`istream& istream::ignore();` bzw.

`istream& istream::ignore(streamsize n);`

liest (falls nicht EOF) ein bzw. `n` Zeichen vom Eingabestrom, ohne diese abzuspeichern.

Zur Leerung des Eingabepuffers sieht der Standard die Funktion

`int istream::sync();`

vor. Funktionsergebnis ist 0, falls der Aufruf erfolgreich war, falls nicht, wird `-1` zurückgegeben und im Zustand des Stroms die Flagge `badbit` (siehe Abschnitt 8.6) gesetzt. (Diese Funktion wird durch unseren Compiler noch nicht unterstützt!)

8.5.2 Formatierung der Eingabe

Für einen Eingabestrom kann man wiederum eine Feldbreite setzen:

`cin.width(n);`

oder mit Manipulator:

`cin >> setw(n);`

(Member-Funktion `width` bzw. Manipulator `setw`, siehe Abschnitt 8.4.3, sind ebenfalls für `istream`'s definiert!)

Eine auf den Wert `n` (vom Typ `streamsize`) gesetzte Feldbreite hat zur Folge, dass bei der nächsten Eingabe einer Zeichenkette (also `char*`) mit dem Eingabeoperator `>>` höchstens `n-1` Zeichen eingelesen werden (`'\0'` wird noch angehängt!).

Ebenfalls kann mittels der Flaggen `ios_base::dec`, `ios_base::oct` und `ios_base::hex` der Gruppe `ios_base::basefield` (Einzustellen mittels der Funktionen `setf`, `flags` oder über die Manipulatoren `dec`, `oct` bzw. `hex`) eingestellt werden, ob bei der Eingabe ganzzahliger Werte

diese dezimale (Standard), oktale oder hexadezimale Eingaben akzeptiert werden:

```
int i;

cin >> i;           // dezimal

cin.setf(ios_base::hex, ios_base::basefield);
cin >> i;           // hexadezimal

cin >> oct >> i;    // oktal
cin >> i;           // immer noch oktal
```

Standardmäßig wird bei `>>` führender Zwischenraum überlesen. Hierzu gibt es in `ios_base` ebenfalls eine "Flagge" `ios_base::skipws`, die standardmäßig "gesetzt" ist. Durch:

```
cin.unsetf(ios_base::skipws);
```

kann man diese Flagge löschen und im Folgenden wird Zwischenraum nicht mehr überlesen. Mit:

```
cin.setf(ios_base::skipws);
```

kann das Überlesen wieder eingeschaltet werden!

Alternativ ist diese Einstellung (laut Standard, leider noch nicht beim GCC-Compiler) auch mittels Manipulatoren möglich:

Ausschalten des Überlesens: `cin >> noskipws;`

Einschalten des Überlesens: `cin >> skipws;`

Unabhängig davon, ob die Flagge `ios_base::skipws` gesetzt ist oder nicht, kann mit dem Manipulator `ws` (*white space*) explizit Zwischenraum überlesen werden:

```
char w[100];
cin >> ws >> w; // erst Zwischenraum ueber-, dann Zeichenkette lesen
```

(Durch `cin >> ws;` werden alle in der Eingabe anliegenden Zwischenraumzeichen gelesen — aber es wird nichts zugewiesen!)

8.6 Fehlerzustände von Strömen

Analog zu den Ein-/Ausgabeeinstellungen eines Streams (siehe Abschnitt 8.4.3) ist der Fehlerzustand eines Streams in einer Variablen vom systemabhängigen (aber

ganzzahligen), in der Klasse `ios_base` definierten Typen `iostate` abgelegt.

Von diesem Typ sind in `ios_base` bereits einige Konstanten (‘‘Flaggen’’) definiert:

Flagge	Bedeutung
<code>ios_base::goodbit</code>	Stream ist in Ordnung (<code>goodbit= 0</code>).
<code>ios_base::eofbit</code>	Ende der Eingabe (Dateiende) erreicht.
<code>ios_base::failbit</code>	Letzte Operation mit dem Stream hat nicht geklappt — der Stream ist aber im Prinzip in Ordnung. (Sollte das Einlesen aufgrund von EOF nicht geklappt haben, ist zusätzlich <code>ios_base::eofbit</code> gesetzt!)
<code>ios_base::badbit</code>	Stream ist nicht mehr verwendbar.

Der Fehlerzustand eines Streams wird (bei Ein–Ausgabe von Standardtypen) weitgehend automatisch vom System verwaltet. Soll etwa ein `int` mit `cin >> i`; eingelesen werden, ist aber das nächste Zeichen in der Eingabe ein nicht zu einem `int`–Wert passender Buchstabe, so wird bekanntlich nichts gelesen und der Variablen `i` auch nichts zugewiesen. Darüberhinaus wird für den Strom `cin` die ‘‘Flagge‘’ `ios_base::failbit` gesetzt, welche anzeigt, dass das Lesen nicht geklappt hat!

Den Fehlerzustand eines Streams kann man mit folgenden Elementfunktionen abfragen, welche jeweils den Wahrheitswert (Typ `bool`) als Ergebnis liefern, ob der entsprechende Zustand zutrifft oder nicht:

Funktion	Bedeutung
<code>bool ios::good();</code>	Stream ist in Ordnung.
<code>bool ios::eof();</code>	<code>ios_base::eofbit</code> ist gesetzt.
<code>bool ios::fail();</code>	<code>ios_base::failbit</code> ist gesetzt.
<code>bool ios::bad();</code>	<code>ios_base::badbit</code> ist gesetzt.

Somit sind einfache Schleifen möglich, etwa: solange wie möglich ganzzahlige Werte einlesen:

```
int i;
...
while ( cin.good() )
{ cin >> i;
  ...
}
...
```

Alternativ könnte man hierfür schreiben:

```
int i;
...
while ( !cin.fail() ) // solange ios_base::failbit nicht gesetzt
{ cin >> i;
  ...
}
...
```

Wird ein Stream selbst als Bedingung verwendet, so wird für diesen implizit die Memberfunktion `fail` aufgerufen und das Funktionsergebnis negiert:

```
while ( cin ) { ... }
```

entspricht:

```
while ( !cin.fail() ) { ... } // Solange ios_base::failbit nicht gesetzt!
```

und:

```
if ( ! cin ) { ... }
```

entspricht:

```
if ( cin.fail() ) { ... } // Falls ios_base::failbit gesetzt
```

Da standardmäßig Ein- und Ausgabeoperator `>>` bzw. `<<` den Stream (nach der Ein-/Ausgabeoperation), für welchen sie aufgerufen wurden, als Funktionsergebnis wieder zurückgeben, sind folgende Konstrukte möglich:

```
int i;
...
while ( cin >> i )
{ ... }
...
```

In dieser Bedingung wird versucht, einen ganzzahligen Wert einzulesen (`cin >> i`). Das Ergebnis dieses Einlesens ist der Stream `cin` nach dem Einleseversuch und in diesem Stream ist sein Zustand nach dem Einlesen festgehalten. Dieser Zustand wird auf `ios_base::failbit` überprüft und die Bedingung ist genau dann wahr, wenn `ios_base::failbit` nicht gesetzt ist. Bei jeder Überprüfung der Bedingung wird erneut eingelesen und erneut der Zustand von `cin` zur Steuerung der `while`-Schleife ermittelt.

Zur Abfrage des Fehlerzustandes eines Streams kann folgende Element-Funktionen verwendet werden:

```
ios_base::iostate ios::rdstate();
```

Der Aufruf von `rdstate` liefert als Ergebnis den derzeitigen Fehlerzustand des Streams, für welchen die Funktion aufgerufen wurde, als Wert vom Typ `ios_base::iostate`, etwa:

```
ios_base::iostate status = cin.rdstate();
```

Dieser in der Variablen `status` abgespeicherte augenblickliche Zustand des Streams (hier `cin`) `cin` kann dann etwa wie folgt verwendet werden:

```
// falls ios_base::badbit gesetzt ist
if ( status & ios_base::badbit ) { ... }
...
```

Zum Setzen des Fehler-Zustandes eines Streams können folgende Funktionen verwendet werden:

1. `void ios::clear()`

(Ohne Argument) Zurücksetzen den Stream auf "*in Ordnung*":

```
cin.clear();
```

2. `void ios::clear(ios_base::iostate status);`

Setzt den Zustand des Stroms auf den als Argument angegebenen, etwa:

```
cin.clear ( cin.rdstate() & ~ios_base::failbit );
```

(Beim bisherigen, von `cin.rdstate()` gelieferten Zustand `ios_base::failbit` zurücksetzen — geschieht durch logische *UND*-Verknüpfung `&` mit dem Bit-Komplement `~ios_base::failbit` dieser Flagge.)

3. `void ios::setstate(ios_base::iostate flagge);`

Fügt dem augenblicklichen Fehlerzustand die als Argument angegebene Flagge hinzu, etwa:

```
cin.setstate(ios_base::failbit);
```

(Setzen von `ios_base::failbit`.)

Definiert man die Ein-/Ausgabeoperatoren `>>` bzw. `<<` für eigene Datentypen, so sollte man auch, wenn eine Lese-/Schreib-Operation nicht ordnungsgemäß durchgeführt werden konnte, durch entsprechende `clear`- oder `setstate`-Aufrufe den Fehlerzustand für den beteiligten Stream setzen.

8.6.1 Beispiel zum Einlesen eines selbstdefinierten Datentypes

Wie wollen für die in früheren Kapiteln bereits behandelte Klasse `Bruch` eine vernünftige Eingabe realisieren.

Es soll der Eingabeoperator `operator>>` verfügbar sein, wie bereits bei der Ausgabe behandelt, ruft diese eine Member-Funktion `scanFrom` der Klasse `Bruch` auf:

```
class Bruch {
protected:
    int zaehler;
    int nenner;
public:
    // Konstruktor:
    Bruch(int =0, int =1);

    // multiplikative Zuweisung:
    const Bruch & operator*=(const Bruch &);

    // sonstiges:
    ...

    // Ausgabe auf Stream:
    void printOn(ostream & =cout) const;

    // Lesen von Stream:
    void scanFrom(istream & =cin);
};
```

```
// Ausgabeoperator << global ueberladen
ostream& operator<<(ostream &strm, const Bruch &b)
{ // ueberkreuzt Ausgabefunktion aufrufen:
    b.printOn(strm);

    // Stream wegen Verkettung zurueckgeben:
    return strm;
}

// Eingabeoperator >> global ueberladen
istream& operator>>(istream &strm, Bruch &b)
{ // ueberkreuzt Eingabefunktion aufrufen:
    b.scanFrom(strm);

    // Stream wegen Verkettung zurueckgeben:
    return strm;
}
```

Die eigentliche Ausgabefunktion ist sehr einfach: es werden Zähler und Nenner durch ein '/' getrennt ausgegeben:

```
void Bruch::printOn( ostream & strm) const
{
    strm << zaehler << '/' << nenner;
}
```

Die Eingabe soll ein wenig flexibler sein:

- es soll zunächst ein ganzzahliger Wert für den Zähler gelesen werden,
- anschließend soll optionaler Zwischenraum überlesen werden,
- anschließend soll der optionale Bruchstrich '/' gelesen werden,
- anschließend soll (möglicherweise nach erneuten Zwischenraumzeichen) der Nenner gelesen werden, wobei der Nenner natürlich nicht gleich 0 sein darf!
- Darüberhinaus soll im abgespeicherten Bruch der Nenner positiv sein, ggf. sind Zähler und Nenner mit -1 zu multiplizieren.

Beim Lesen eines Bruches kann also einiges schiefgehen:

- falsche Zeichen, etwa ein Buchstabe an einer Stelle, wo Zähler oder Nenner oder '/' erwartet wurde,
- Nenner gleich 0.

In beiden Fällen war das Einlesen eines Bruches nicht erfolgreich und dies sollte im Zustand des Streams vermerkt werden!

Hier nun die (gegenüber der Ausgabe etwas aufwändigere) Implementierung der Eingabefunktion `scanFrom`:

```
void Bruch::scanFrom(istream &strm)
{ // Zwischenspeicher fuer gelesene int-Werte
  int z, n;

  // Zaehler einlesen:
  // sollte hierbei etwas schiefgehen,
  // sorgt der Standard fuer das Setzen des failbit's
  strm >> z;

  // Zwischenraumzeichen ueberlesen:
  strm >> ws;

  // optionales '/' einlesen
  if ( strm.peek() == '/' ) // peek: naechstes Zeichen mal anschauen
    strm.get();             // falls es '/' ist, dann lesen!

  // Nenner einlesen:
  // sollte hierbei etwas schiefgehen,
  // sorgt der Standard fuer das Setzen des failbit's
  strm >> n;

  // falls bislang etwas nicht geklappt hat: Funktion beenden:
  // ggf. gesetztes failbit ist in strm gesetzt
  // und kommt somit beim Aufrufer an!
  if ( !strm )
    return;

  // Nenner == 0
  if ( n == 0 )
  { // falls n= 0: failbit setzen und Funktion beenden:
    strm.setstate (ios_base::failbit);

    return;
  }

  if ( n < 0 )
  { zaehler = -z;
    nenner  = -n;
  }
  else
  { zaehler = z;
    nenner  = n;
  }

  return;
}
```


8.7 Manipulatoren

Wir haben bereits einige, vom Standard vorgesehene Manipulatoren kennengelernt, etwa:

```
cout << endl;           // Zeilenvorschub
cout << scientific;      // wissenschaftliche Ausgabe von Gleitkommawerten
...
cout << setw(10);       // Feldbreite setzen
...
cin >> ws;              // Zwischenraum ueberlesen
...
```

Diese beeinflussen den Aus- bzw. Eingabestrom, ohne dass explizit etwas ausgegeben oder eingelesen werden muss.

Man muss zwischen Ausgabe- und Eingabemanipulatoren unterscheiden, d.h. gewisse Manipulatoren sind für Ausgabeströme vorgesehen, andere für Eingabeströme (manche für beide!).

Einige dieser Manipulatoren haben keine Argumente, einige haben ein Argument.

Man kann eigene Manipulatoren definieren, d.h. selbst eine Funktionalität implementieren, welche wie ein Manipulator anzuwenden ist.

Die hierzu notwendige Technik und das, was der Standard hier an Vorarbeit leistet, soll kurz beschrieben werden:

8.7.1 Manipulatoren ohne Argumente

Ein argumentloser Ausgabemanipulator `omanip` ist eine Funktion folgenden Types:

```
ostream & omanip( ostream &);
```

also eine Funktion mit einer `ostream`-Referenz als Parameter und Funktionsergebnis. Der Ausgabemanipulator `endl` könnte also wie folgt definiert sein:

```
ostream & endl( ostream & strm)
{
    strm << '\n';        // Zeilenvorschub ausgeben:
    strm.flush();        // Ausgabepuffer leeren

    return strm;         // Stream zwecks Verkettung zurueckgeben
}
```

Zur Klasse `ostream` ist in der Standardbibliothek der Ausgabeoperator `<<` bereits für solche Funktionsadressen passend, etwa wie folgt als Member-Funktion zur Klasse `ostream` überladen (könnte aber auch als globale Operatorfunktion realisiert sein!):

```
ostream & ostream::operator<<( ostream & (*manip) (ostream &) )
{ return (*manip) ( *this);
}
```

d.h. als Parameter hat diese Operatorfunktion einen entsprechenden Funktionszeiger und der Aufruf dieser Operator-Funktion mit einer Funktionsadresse als Argument wird durch “Überkreuzung” umgesetzt in den Aufruf der Manipulatorfunktion (über diese Adresse) mit dem aktuellen Stream als Argument.

```
cout << omanip
```

```
    omanip(cout)
```

Dadurch, dass die Manipulatorfunktion einen `ostream` als Ergebnis zurückliefert, kann der Manipulator auch verkettet aufgerufen werden:

```
cout << ... << omanip << ... << opmanip << ... ;
```

Analog ist ein (argumentloser) Eingabemanipulator eine Funktion folgenden Types:

```
istream & imanip( istream &);
```

also eine Funktion mit einem `istream`-Referenz-Parameter und Funktionsergebnis. Für die Adresse einer solchen Funktion ist der Eingabeoperator (möglicherweise als Member-)Funktion der Klasse `istream` analog wie folgt überladen (Könnte wiederum auch als globale Operatorfunktion definiert sein!):

```
istream & istream::operator>>( istream & (*manip) (istream &) )
{ return (*manip) ( *this);
}
```

Als Beispiel für einen selbstdefinierten Eingabemanipulator wollen wir einen Manipulator `ignoreLine` vorstellen, der wie folgt aufgerufen:

```
cin >> ignoreLine;
```

dafür sorgt, dass alles bis (einschließlich) des nächsten Zeilenvorschubes überlesen wird:

```
istream & ignoreLine( stream & strm)
{
    char c;

    // in Schleife solange zeichenweise lesen,
    // solange Lesen moeglich und gelesenes Zeichen
    // kein Zeilenvorschub ist:
    while ( strm.get(c) && c != '\n' ) ;    // leerer Anweisungsteil!

    return strm; // Stream zwecks Verkettung zurueckgeben!
}
```

8.7.2 Manipulatoren mit einem Argument

Beispiel für einen selbstdefinierten Manipulator mit einem `int` Argument:

Wie wollen einen Manipulator `space(int n)` entwickeln, der in der Form

```
cout << space(10);
```

aufgerufen werden kann und dafür sorgt, dass soviele Blanks, wie im Argument angegeben, ausgegeben werden. (Hier soll also ein Ausgabemanipulator mit einem `int`-Argument geschrieben werden, bei einem Eingabemanipulator ist entsprechend zu verfahren!)

Die Situation bei diesem Aufruf ist nicht ganz so einfach, weil dieses `space(10)` aufgrund der C++-Syntax als Funktionsaufruf aufgefasst wird und nicht ohne weiteres (durch “Überkreuzen”) in den Aufruf (über die Adresse) einer Funktion mit `cout` und dem zusätzlichen Argument 10 umgesetzt werden kann.

Man muss hier den Manipulator `space(10)` und die über den Manipulator auszulösende Funktion trennen:

Man kann (muss) eine Funktion folgenden Types schreiben (diese Funktion heißt im Folgenden *Manipulator-Funktion*):

```
// Funktion mit ostream-Referenz und einem int als Parameter und
// ostream-Referenz als Ergebnis:
ostream & spaces( ostream & strm, int n)
{
    for ( int i = 0; i < n; ++i)
        strm << ' ';    // n Blanks ausgeben
}
```

und dafür sorgen, dass der Aufruf:

```
cout << space(10)
```

durch “Überkreuzen” in den Aufruf dieser Manipulator-Funktion:

```
spaces(cout, 10)
```

umgesetzt wird.

Die Vorgehensweise ist hier:

1. Der Aufruf `space(10)` gibt ein Objekt zurück, in dem irgendwie die aufzurufende Manipulator-Funktion (`spaces`) über deren Adresse und der (ganzzahlige) Wert 10 (als Argument für die Manipulator-Funktion) abgespeichert sind.
2. Der `ostream`-Ausgabeoperator `<<` ist für ein derartiges Objekt als zweiten Operand so zu überladen, dass die entsprechende, im Objekt abgespeicherte Funktion mit dem im Objekt abgespeicherten Wert für den entsprechende `ostream` aufgerufen wird.
3. Da beim Aufruf `space(10)` der Name `spaces` der tatsächlich aufzurufenden Manipulator-Funktion nicht angegeben ist, muss dieses `space` selbst bereits die Manipulator-Funktion `spaces` kennen! (D.h. `space` muss ein Objekt einer Klasse sein, in der die Manipulator-Funktion `spaces` als Funktionszeiger abgespeichert ist, und für welche der Funktionsaufruf mit einem ganzzahligen Argument möglich ist!)

Die Umsetzung kann wie folgt geschehen:

1. Generelle Vorbereitung:

```

// Klasse, welche von einem Manipulator-Aufruf zurueckgegeben wird:
class manipulator_objekt {
private:
    // Zeiger auf Manipulator-Funktion
    ostream& (*mo_fkt_ptr)(ostream &, int);
    // Argument fuer Manipulator-Funktion
    int arg;

public:
    // Konstruktor:
    // 1.) als 1. Argument angegebene Funktion in
    //     Komponente mo_fkt_ptr speichern    und
    // 2.) als 2. Argument angegebene Zahl in
    //     Komponente arg speichern:
    manipulator_objekt(ostream& (*fkt_ptr)(ostream &, int),
                       int wert)
        : mo_fkt_ptr(fkt_ptr), arg(wert) {}

    // Anwenden: Aufruf der gespeicherten Funktion mit
    //             dem gespeicherten Argument
    ostream & anwenden( ostream & strm)
    { // gespeicherte Funktion auf strm mit
      // gespeichertem Argument anwenden:
      return (*mo_fkt_ptr)( strm, arg);
    }
};

// Klasse, in der die Manipulator-Funktion abgespeichert ist
// und welche beim Aufruf der Operatorfunktion operator()(int)
// ein passendes Objekt der Klasse manipulator_objekt zurueckgibt:
class manipulator {
private:
    // Zeiger auf manipulator_funktion
    ostream & (*fkt_ptr)( ostream &, int);
public:
    // Konstruktor: Parameter ist passender Funktionszeiger:
    manipulator( ostream& (*f)(ostream &, int) ) : fkt_ptr(f) { }
    // Ueberladung des ()-Operators:
    manipulator_objekt operator()(int n)
    { return manipulator_objekt( fkt_ptr, n); }
};

// globale Ueberladung des Ausgabeoperators <<
// fuer ein manipulator_objekt:
ostream& operator<<( ostream & strm, manipulator_objekt mo)
{ mo.anwenden(strm); // "ueberkreuzt" anwenden:
}

```

2. Konkrete Manipulator-Funktion definieren:

```
// Manipulator-Funktion
ostream& spaces(ostream & strm, int n)
{
    for ( int i = 0; i < n; ++i)
        strm << ' ';
    return strm;
}
```

3. Manipulator erzeugen, diesen mit der definierten Manipulator-Funktion `spaces` verknuepfen:

```
manipulator space(spaces);
```

(Im Objekt `space` der Klasse `manipulator` ist jetzt in der Funktions-Zeiger-komponente `fkt_ptr` die Adresse der Manipulator-Funktion `spaces` abgelegt!)

4. Manipulator anwenden:

```
cout << space(10)
      ①
cout << manipulator_objekt(spaces,10)
      ②
      spaces(cout,10)
```

① `space(10)`

Bei diesem Aufruf der Operator-Funktion `operator()` für das Objekt `space` wird ein `manipulator_objekt` zurückgegeben, in dem die Manipulator-Funktion `spaces` (als Funktionszeiger) und das entsprechende ganzzahlige Argument 10 abgespeichert ist.

② `cout << manipulator_objekt(spaces,10)`

Durch die Überladung von `operator<<` für solche `manipulator_objekte` wird die gespeicherte Funktion mit dem gespeicherten Argument für den `ostream cout` aufgerufen!

Verallgemeinerung: Manipulator mit einem Argument beliebigen Types:

Mittels der allgemeinen Vorbereitung in obigem Beispiel können wir beliebige Manipulatoren mit einem `int`-Argument definieren und anwenden:

```
// Manipulator--Funktionen definieren:
ostream & manip1_f( ostream & strm, int i) { ... }
ostream & manip2_f( ostream & strm, int j) { ... }
ostream & manip3_f( ostream & strm, int j) { ... }
```

```
// Manipulatoren vereinbaren, dabei jeweilige
// Manipulator-Funktion eintragen:
manipulator manip1( manip1_f);
manipulator manip2( manip2_f);
manipulator manip3( manip3_f);
...
// Manipulatoren verwenden:
cout << ... << manip1(7) << ... << manip2(5) << ... << manip3(-3);
...
```

Allen wie oben definierten Manipulatoren ist gemeinsam, dass ihr zusätzliches Argument ein `int` ist.

Mittels Templates kann man das auf beliebige Typen verallgemeinern:

Manipulator mit einem Argument eines beliebigen Types `T`:

```
template <class T>
class manipulator_objekt {
private:
    // Zeiger auf Manipulator-Funktion:
    ostream& (*mo_fkt_ptr)(ostream &, T);
    // Argument fuer Manipulator-Funktion:
    T arg;
public:
    // Konstruktor:
    manipulator_objekt( ostream& (*fkt_ptr)(ostream &, T), T wert)
        : mo_fkt_ptr(fkt_ptr), arg(wert) {}
    ostream & anwenden( ostream & strm)
    { return (*mo_fkt_ptr)( strm, arg);
    }
};
```

```
template <class T>
class manipulator {
private:
    // Zeiger auf manipulator_funktion
    ostream & (*fkt_ptr)( ostream &, T);
public:
    // Konstruktor: Parameter ist passender Funktionszeiger:
    manipulator( ostream& (*f)(ostream &, T) ) : fkt_ptr(f) { }
    // Ueberladung des ()-Operators:
    manipulator_objekt<T> operator()(T n)
    { return manipulator_objekt<T>( fkt_ptr, n); }
};
```

```
// globale Ueberladung des Ausgabeoperators <<
// fuer ein manipulator_objekt:
template <class T>
ostream& operator<<( ostream & strm, manipulator_objekt<T> mo)
{ mo.anwenden(strm);
}
```

Konkrete Manipulator-Funktion (für konkreten Typ T, hier `int`) definieren:

```
ostream & spaces(ostream &strm, int n)
{
    for ( int i = 0; i < n; ++i)
        strm << ' ';
    return strm;
}
```

Entsprechenden Manipulator (instanziiert für `int`) vereinbaren:

```
manipulator<int> space(spaces);
```

Manipulator anwenden:

```
cout << ... << space(5) << ... << space(10) << ... ;
```

Headerdatei `<iomanip>`

Genau derartige Templates und deren Instantiierungen für die vom Standard vorgesehenen Manipulatoren mit einem Argument (etwa `setw(int)`) sind in der Headerdatei `<iomanip>` definiert, so dass für die Verwendung eines parameterbehafteten Standard-Manipulators diese Headerdatei immer zu inkluden ist!

Meistens kann man die in dieser Headerdatei definierten Templates dazu verwenden, auch eigene Manipulatoren mit Argument schnell und einfach zu erzeugen, die allgemeine Vorbereitung durch Templates ist dann nicht mehr erforderlich. (Man braucht dann nur noch die eigentliche Manipulator-Funktion zu definieren und ein Objekt des entsprechend instantiierten Manipulator-Templates zu erzeugen.)

Auf dem GCC-Compiler reicht etwa folgendes:

```
#include <iostream>
#include <iomanip>
...
// Manipulator-Funktion definieren:
ostream& spaces( ostream & strm, int n) { ... }
// Manipulator definieren:
oapp<int> space(spaces);
...
// Manipulator verwenden:
cout << ... << space(5) << ... << space(10) << ... ;
```

(Der Name des Templates für Ausgabemanipulatoren ist beim GCC also `oapp`, der für Eingabemanipulatoren ist entsprechend `iapp`!)
 Leider sind die Namen und genaue Funktionalität der Templates in `<iomanip>` nicht standardisiert, so dass das ganze auf anderen Compilern zwar ähnlich, aber nicht identisch funktioniert!

Verallgemeinerung auf Manipulatoren mit mehreren Argumenten:

Die oben beschriebene Technik kann leicht für Manipulatoren mit mehreren Argumenten unterschiedlicher Typen verallgemeinert werden, hier exemplarisch für einen Ausgabemanipulator mit zwei Argumenten:

```
template <class T1, class T2>
class omanip2 {
private:
    // Zeiger auf Manipulator-Funktion
    ostream& (*mo_fkt_ptr)(ostream &, T1, T2);
    // erstes Argument fuer Manipulator-Funktion
    T1 arg1;
    // zweites Argument fuer Manipulator-Funktion
    T2 arg2;
public:
    // Konstruktor:
    omanip2(ostream& (*fkt_ptr)(ostream &, T1, T2),
            T1 wert1, T2 wert2)
        : mo_fkt_ptr(fkt_ptr), arg1(wert1), arg2(wert2) {}
    ostream & anwenden( ostream & strm)
    { return (*mo_fkt_ptr)( strm, arg1, arg2);
    }
};

template <class T1, class T2>
class oapp2 {
private:
    // Zeiger auf manipulator_funktion
    ostream & (*fkt_ptr)( ostream &, T1, T2);
public:
    // Konstruktor: Parameter ist passender Funktionszeiger:
    oapp2( ostream& (*)(ostream &, T1, T2) ) : fkt_ptr(f) { }
    // Ueberladung des ()-Operators:
    omanip2<T1, T2> operator()(T1 n, T2 m)
    { return omanip2<T1,T2>( fkt_ptr, n, m); }
};

// globale Ueberladung des Ausgabeoperators <<
// fuer ein manipulator_objekt:
```



```
template <class T1, class T2>
ostream& operator<< ( ostream & strm,omanip2<T1, T2> mo)
{ mo.anwenden(strm);
}
```

Ein konkreter Manipulator, hier etwa `fuellen(n,c)` zur n -maligen Ausgabe des Zeichens c , kann dann wie folgt erzeugt und verwendet werden:

```
// Manipulator-Funktion definieren:
ostream & fuellen(ostream &strm, int n, char c)
{ // n mal das Zeichen c ausgeben:
  for ( int i = 0; i < n; ++i)
    strm << c;
  return strm;
}

// Manipulator erzeugen:
oapp2<int, char> fuelle(fuellen);

// Manipulator anwenden:
cout << ... << fuelle( 5,' ') << ... ; // 5 Blanks ausgeben
cout << ... << fuelle(10,'_') << ... ; // 10 Underscores ausgeben
...
```

Da der Standard keine Manipulatoren mit zwei oder mehreren Argumenten enthält, sind die entsprechenden Templates hierfür in `<iomanip>` nicht vorhanden!

8.8 Dateibehandlung

Natürlich kann in C++ auch von Dateien gelesen und auf Dateien geschrieben werden. Hierzu gibt es die Klassen:

- `ifstream` (*input file stream*) für Dateien, von denen gelesen werden soll (von der Klasse `istream` abgeleitet),
- `ofstream` (*output file stream*) für Dateien, auf die geschrieben werden soll (von `ostream` abgeleitet), und
- `fstream` (*file stream*) für Dateien zum Lesen und Schreiben (von `iostream` abgeleitet, `iostream` selbst ist von `istream` und `ostream` abgeleitet).

8.8.1 Öffnen und Schließen von Dateien mittels Konstruktoren und Destruktoren

Objekte der Klassen `istream`, `ostream` bzw. `fstream` werden (i. Allg.) bei ihrer Erzeugung (durch Konstruktoren) mit einer Datei des Systems verknüpft — bei der Erzeugung muss dann ein Dateiname angegeben werden — und bei der “Zerstörung“ der entsprechenden Objekte durch den Destruktor automatisch geschlossen:

```

void f(void)
{ ...
    ofstream ausgabe("Ausgabe.txt");
    /* ausgabe ist ein Objekt vom Typ ofstream und ist mit
       der Datei Ausgabe.txt des Systems verknuepft.
       Diese Datei wird zum Schreiben geoeffnet          */
    ...
    ifstream eingabe("Eingabe.txt");
    /* eingabe ist ein Objekt vom Typ ifstream und ist mit
       der Datei Eingabe.txt des Systems verknuepft.
       Diese Datei wird zum Lesen geoeffnet              */
    ...
    // Hier, beim Ende des Blocks werden die Dateien
    // automatisch mittels des Destruktors geschlossen!
}

```

Die Art des Öffnens einer Datei kann beim Konstruktoraufruf als zusätzliches Argument vom implementierungsspezifischen (ganzzahligen) Typ `ios_base::openmode` angegeben werden. Hierzu stehen *ODER*-Verknüpfungen folgender, auch in `ios_base` definierter Konstanten zur Verfügung:

Konstante	Bedeutung
<code>ios_base::in</code>	Zum Lesen öffnen.
<code>ios_base::out</code>	Zum Schreiben öffnen.
<code>ios_base::app</code>	<u>Jede</u> Schreiboperation erfolgt am Dateiende.
<code>ios_base::ate</code>	Datei wird beim Öffnen (einmalig) auf's Dateiende positioniert.
<code>ios_base::binary</code>	Binär-Modus statt Text-Modus.
<code>ios_base::trunc</code>	Datei wird beim Öffnen auf Länge 0 gekürzt.

Eine mit einem Objekt vom Typ `ofstream` verknüpfte Datei wird standardmäßig mit dem Modus `ios_base::out | ios_base::trunc` geöffnet, d.h. als Textdatei zum Schreiben, wobei sie, falls sie bereits existiert, auf Länge 0 gekürzt wird:

```
ofstream ausgabe("datei.txt");
```

entspricht:

```
ofstream ausgabe("datei.txt", ios_base::out|ios_base::trunc);
```

Existiert die Datei noch nicht, wird sie erzeugt. Die Dateiposition ist in jedem Fall der Dateianfang.

Beim standardmäßigen Öffnen einer Eingabedatei:

```
ifstream eingabe("datei.txt");
```

ist `ios_base::in` der Öffnungsmodus, entspricht also folgendem Öffnen:

```
ifstream eingabe("datei.txt", ios_base::in);
```

und die Datei muss natürlich vorhanden sein.

Natürlich kann das Öffnen einer Datei schiefgehen (etwa, wenn eine Eingabedatei noch nicht existiert oder eine Ausgabedatei nicht erzeugt werden kann). Dieser Fehlerzustand wird ebenfalls wieder im Objekt abgespeichert (Fehlerflaggen `ios_base::failbit`

und `ios::badbit` werden gesetzt!), so dass ein solcher Fehler wie folgt abgefangen werden kann:

```
...
ifstream eingabe ("datei.txt");
if ( !eingabe ) // Oeffnen hat nicht geklappt!
{ ... }
...
```

Beim Öffnen und Verknüpfen einer Datei mit einem Strom vom Typ `fstream`, wird diese standardmäßig im Modus `ios_base::in | ios_base::out` geöffnet (bei unserem Compiler muss dennoch dieser Modus beim Öffnen explizit angegeben werden!). Im Übrigen kann man jeden Dateistrom (also ein Objekt vom Typ `ifstream`, `ofstream` oder `fstream`) zum Lesen (`ios_base::in`) und zum Schreiben (`ios_base::out`) öffnen — der Typ des Objektes legt nur fest, welche Operationen mit dem Stream möglich sind, also etwa Eingabeoperationen für `ifstream`'s, Ausgabeoperationen für `ofstream`'s und beides für `fstream`'s.

D.h. man kann also auch einen `ofstream` zum Lesen öffnen, man hat dann aber für den `ofstream` keine Leseoperationen zur Verfügung:

```
...
int i;
// Ausgabestrom zum Lesen oeffnen!!!
ofstream strom("datei.txt",ios_base::in); // Seltsam, aber ok!
...
strom >> i; // Fehler: kein >> Operator fuer ofstream definiert!
...
```

8.8.2 Öffnen und Schließen von Dateien mittels Elementfunktionen

Mittels der Member-Funktion `close` kann eine mit einem Stream-Objekt verknüpfte Datei auch explizit geschlossen werden, ohne, dass das Stream-Objekt als ganzes zerstört wird:

```
...
/* ofstream-Objekt ausgabe erzeugen und mit der
   Datei Ausgabe.txt verknuepfen, diese dabei oeffnen: */
ofstream ausgabe("Ausgabe.txt");
...
ausgabe.close();
/* die mit ausgabe verknuepfte Datei wird geschlossen, das
   ofstream-Objekt ausgabe ist jedoch noch vorhanden, aber
   nicht mehr mit einer Datei verknuepft */
...
```

Ein so durch `close` nicht mehr mit einer Datei verknüpftes — oder auch ein ohne Angabe eines Dateinamens erzeugtes Stream-Objekt:

`ofstream` `ausgabe`; // noch nicht mit einer Datei verknuepft!

kann man explizit durch Aufruf der Member-Funktion `open` mit einer Datei verknüpfen. Hierbei wird dann wiederum die Datei geöffnet:

```
...
ofstream ausgabe("Ausgabe1.txt"); // mit Datei verknuepfter Stream
ifstream eingabe;                // nicht mit Datei verknuepfter Stream
...
eingabe.open("Eingabe.txt");      // eingabe mit einer Datei verknuepfen
...
ausgabe.close(); // Ausgabe1.txt schliessen,
                // ausgabe ist nicht mehr mit einer Datei verknuepft
...
ausgabe.open("Ausgabe2.txt");    // ausgabe mit neuer Datei verknuepfen
...
```

Beim Aufruf von `open` kann wiederum als zweites Argument der gewünschte Modus des Öffnens der Datei angegeben werden, etwa:

```
ausgabe.open("Ausgabe2.txt", ios::out|ios::app);
```

falls man etwa ans Ende der Datei schreiben möchte!

Mit der Elementfunktion `is_open()`; erhält man als Ergebnis einen boolschen Wert, der darüber Auskunft gibt, ob der Stream mit einer Datei verknüpft ist oder nicht, etwa:

```
...
ofstream ausgabe;
...
if ( ! ausgabe.is_open() ) // nicht mit einer Datei verknuepft?
{ ... }
...
```

8.8.3 Verwenden geöffneter Dateien

Durch die Ableitung von den Klassen `ostream` und `istream` sind für diese neuen Klassen `ifstream`, `ofstream` und `fstream` alle Funktionen, Operatoren und Techniken (insbes. Manipulatoren) mit gleicher Bedeutung definiert und verfügbar, wie sie in den entsprechenden Basisklassen definiert und in den letzten Abschnitten beschrieben worden sind.

Die Ausgabe auf eine geöffnete und mit `ofstream` `ausgabe` verknüpfte Datei `Ausgabe.txt` kann also wie folgt geschehen:

```
ostream ausgabe("Ausgabe.txt");
...
ausgabe << "Ergebnis: " << i << endl;
...
```

und das Lesen von einer geöffneten und mit `ifstream` `eingabe` verknüpften Datei `Eingabe.txt` kann wie folgt geschehen:

```

int i;
double x;
istream eingabe("Eingabe.txt");
...
eingabe >> i;           // Einlesen eines int
if ( !eingabe ) { ... } // Lesen hat nicht geklappt
...
eingabe >> x;           // Einlesen eines double
if ( !eingabe ) { ... } // Lesen hat nicht geklappt
...
if ( eingabe.eof() ) { ... } // Falls Dateiende
...

```

8.8.4 Dateipositionierung

Zu jeder geöffneten Datei wird die augenblickliche Dateiposition verwaltet. Nach dem Öffnen einer Datei ist diese i. Allg. der Dateianfang (es sei denn, die Datei wird mit Modus `ios::ate` geöffnet!).

Zum wahlfreien Zugriff auf einen mit einer Datei verknüpften Stream sind im Standard folgende Datentypen:

Typ	Bedeutung						
<code>pos_type</code>	(Ganzzahliger, in <code>char_traits</code> definierter) Datentyp zur Beschreibung der Position in einem Stream. (Bei unserem Compiler heißt dieser Typ noch <code>streampos</code> .)						
<code>ios_base::seekdir</code>	Datentyp zur Beschreibung eines Bezugspunktes innerhalb einer Datei. Standardmäßig gibt es die drei Bezugspunkte (von diesem Typ): <table border="1" data-bbox="581 1241 1253 1362"> <tr> <td><code>ios_base::beg</code></td><td>Dateianfang</td></tr> <tr> <td><code>ios_base::cur</code></td><td>augenblickliche Dateiposition</td></tr> <tr> <td><code>ios_base::end</code></td><td>Dateiende</td></tr> </table>	<code>ios_base::beg</code>	Dateianfang	<code>ios_base::cur</code>	augenblickliche Dateiposition	<code>ios_base::end</code>	Dateiende
<code>ios_base::beg</code>	Dateianfang						
<code>ios_base::cur</code>	augenblickliche Dateiposition						
<code>ios_base::end</code>	Dateiende						
<code>off_type</code>	(Ganzzahliger, in <code>char_traits</code> definierter) Datentyp zur Beschreibung des Offsets (Abstandes) von einem Bezugspunkt. (Bei unserem Compiler heißt dieser Typ noch <code>streamoff</code> .)						

und folgende Funktionen definiert:

1. Für `ofstream`'s (die zugehörigen Funktionsnamen enden mit dem Buchstaben `p` für *put* und es wird die sog. Schreibposition beeinflusst!):
 - `pos_type tellp();`
Gibt augenblickliche Schreibposition als Wert vom Typ `pos_type` zurück.
 - `ostream& seekp (pos_type position);`
Setzt die Schreibposition auf das angegebene Argument `position` vom Typ `pos_type`, etwa:

```

...
pos_type position;
ofstream ausgabe(...);
...
position = ausgabe.tellp(); // akt. Schreibposition merken
...
ausgabe << ...;           // weitere Ausgaben
...
ausgabe.seekp(position);   // auf gemerkte Schreibposition
                           zurueckstellen
...

```

Das Funktionsergebnis von `seekp` ist der Stream, für welchen die Funktion aufgerufen wurde!

- `ostream& seekp (off_type anzahl, ios::seekdir ursprung);`
Setzt die Schreibposition ausgehend vom angegebenen Bezugspunkt `ursprung` um `anzahl` Zeichen weiter (rückwärts, falls `anzahl < 0`, sonst vorwärts). Es kann nicht hinter das Dateiende positioniert werden (d.h. ist `ios::end` der Ursprung, so darf `anzahl` nicht positiv sein!). Es kann nicht vor den Dateianfang positioniert werden (d.h. ist `ios::beg` der Ursprung, so darf `anzahl` nicht negativ sein!).

2. Für `ifstream`'s (die zugehörigen Funktionsnamen enden mit dem Buchstaben `g` für `get` und es wird die sog. Leseposition beeinflusst!):

- `pos_type tellg();`
Gibt augenblickliche Leseposition als Wert vom Typ `pos_type` zurück (`g` wie `get`, Leseposition!).
- `istream& seekg (pos_type position);`
Setzt die Leseposition auf das angegebene Argument `position` vom Typ `pos_type`.
- `istream& seekg (off_type anzahl, ios::seekdir ursprung);`
Setzt die Leseposition ausgehend vom angegebenen Bezugspunkt `ursprung` um `anzahl` Zeichen weiter (rückwärts, falls `anzahl < 0`, sonst vorwärts). Es kann wiederum nicht hinter das Dateiende bzw. vor den Dateianfang positioniert werden.
Zu beachten ist, dass man, wenn man nach Erreichen des Dateiendes mittels einer Leseoperation (hierbei wird ja die Flagge `ios::eofbit` gesetzt!) die Datei etwa auf den Anfang "zurückspult":
`eingabestrom.seekg(0,ios::beg); // auf Dateianfang stellen`
anschließend die EOF-Flagge noch zurücksetzen muss, um wieder ordnungsgemäß Lesen zu können:
`eingabestrom.clear();`
(vgl. Abschnitt 8.6!).

3. Für `fstream`'s sind alle oben beschriebenen Funktionen (`tellp`, `seekp`, `teelg` und `seekg`) aufrufbar, i. Allg. ist hierbei jedoch die Lese- und Schreibposition identisch!

8.9 String-Streams

Zum Schreiben auf Strings und Lesen von Strings stehen im Standard die Klassen `istream` (Lesen), `ostream` (Schreiben) und `stringstream` (Lesen und Schreiben) zur Verfügung. Zu deren Verwendung muss die Headerdatei `<string>` inkludiert werden.

Auf die Besonderheiten von String-Streams wird in Abschnitt 10.13 des Kapitels 10 über die C++-Standard-Strings etwas genauer eingegangen.

8.10 Sonstiges

8.10.1 Verbinden eines Eingabestroms mit einem Ausgabestrom

Ein Objekt der Klasse `istream` (also etwa `cin`) kann durch die Element-Funktion:

```
ostream* istream::tie(ostream *);
```

mit (maximal) einem Objekt der Klasse `ostream` "verknüpft" werden, etwa:

```
cin.tie( &cout);
```

Als Argument ist die Adresse des `ostreams` anzugeben, mit dem der `istream` zu verknüpfen ist! Dieses "Verknüpfen" bewirkt, dass vor jeder Leseoperation für den `istream` (hier `cin`) der `ostream` (hier `cout`) "geflusht" (d.h. Ausgabepuffer geleert) wird.

Funktionsergebnis dieser Funktion ist die Adresse des `ostreams`, mit dem bislang der `istream` verknüpft war bzw. 0, falls es keinen derartigen gab!

Durch ein Argument 0 bei einem Aufruf von `tie` wird die Verknüpfung des `istreams` mit einem `ostream` aufgehoben.

Der Aufruf von `tie` ohne Argument dient zur Abfrage, mit welchem `ostream` der `istream` zur Zeit verknüpft ist (dessen Adresse wird als Funktionsergebnis geliefert bzw. 0, falls keine Verknüpfung bestand).

Standardmäßig ist `cin` mit `cout` verknüpft, so dass eventuelle Eingabeaufforderungen (auf `cout` ausgegeben) wirklich auf dem Bildschirm erscheinen, bevor dann von `cin` eingelesen wird.

8.10.2 Ströme und Ausnahmen

Der Fehlerzustand eines Stroms kann sich bei jeder Ein-/Ausgabeoperation ändern und zumindest nach jeder Eingabeoperation sollte überprüft werden, ob das Einlesen geklappt hat.

Eine derartige Überprüfung ist oft lästig (insbesondere bei Ausgabeoperationen — aber auch diese sollen bisweilen schiefgehen).

Der Standard sieht Folgendes vor (funktioniert nicht auf unserem Compiler):

Man kann zu einem Stream und jeder der “Fehlerflaggen” `ios_base::failbit`, `ios_base::eofbit` und `ios_base::badbit` einstellen, dass, wenn die entsprechende Flagge gesetzt wird, eine Ausnahme vom Typ `ios_base::failure` ausgelöst wird. Dieses Einstellen erfolgt durch den Aufruf der Funktion

```
void ios::exceptions(ios_base::iostate);
```

wobei man als Argument die (*ODER*–) Verknüpfung der Flaggen angeben muss, bei denen die Ausnahme ausgelöst werden soll, etwa:

```
cin.exceptions(ios_base::badbit | ios_base::failbit | ios_base::eofbit );
```

Hiermit wird zum Stream `cin` eingestellt, dass, wann immer der Zustand des Streams auf `eof`, `fail` oder `bad` wechselt, eine Ausnahme (Typ: `ios_base::failure`) ausgelöst wird.

Beim Aufruf von `exceptions()`; ohne Argument wird eine überladene Form der Funktion

```
ios::iostate ios::exceptions();
```

aufgerufen, welche als Funktionsergebnis einen Wert vom Typ `ios_base::iostate` liefert, in dem diejenigen Flaggen gesetzt sind, für welche zur Zeit eine Ausnahme ausgeworfen wird!

Zurücksetzen auf “*keine Ausnahmen auswerfen*“ geschieht durch den Aufruf von `exceptions` mit Argument 0.

Kapitel 9

Ausnahmen in der Standardbibliothek

Die Behandlung von Ausnahmen haben wir bereits in Abschnitt 2.11 kennengelernt. Die Standardbibliothek bedient sich dieser Technik, um auf unvorhergesehene Situationen zu reagieren.

9.1 Die Headerdatei `exception`

In der Standardbibliothek ist hierzu (in der Headerdatei `<exception>`) eine Klasse `exception` definiert und alle in der Standard-Bibliotheksfunktionen ausgeworfenen Ausnahmetypen sind von dieser Klasse abgeleitet.

Die Funktionalität dieser Klasse ist:

```
class exception {
public:
    // Konstruktoren:
    exception() throw(); // parameterlos
    exception(const exception&) throw(); // Copy-Konstruktor

    exception operator=(const exception&) throw(); // Zuweisung

    virtual ~exception() throw(); // Destruktor

    virtual const char * what() const throw(); // Fehlermeldung liefern
};
```

Bemerkenswert an diesen Definitionen ist, dass alle `exception`-Member-Funktionen keine weiteren Ausnahmen auswerfen.

Die (virtuelle) Funktion `what()` liefert einen Zeiger auf einen C-String (`'\0'`-terminiert), der eine Fehlermeldung enthält.

Der genaue Wortlaut der Fehlermeldung ist für diese ziemlich allgemeine Basis-Fehlerklasse implementierungsabhängig, kann aber für abgeleitete Klassen (durch Neudefinition der virtuellen Funktion) abgeändert und eigenen Bedürfnissen angepasst werden.

(Bei unserem GCC-Compiler lautet diese Meldung: `9exception` und beim SUN-Workshop-Compiler: `Unamed exception`.)

In dieser Headerdatei `<exception>` ist bereits die von der Klasse `exception` abgeleitete Klasse `bad_exception` definiert, welche im Zusammenhang mit Ausnahmespezifikationen von Funktionen (vgl. Abschnitt 2.11) eine Rolle spielt:

```
class bad_exception : public exception {
public:
    // Konstruktoren:
    bad_exception() throw();                // parameterlos
    bad_exception(const exception&) throw(); // Copy-Konstruktor

    bad_exception operator=(const bad_exception&) throw(); // Zuweisung

    virtual ~bad_exception() throw();        // Destruktor

    virtual const char * what() const throw(); // Fehlermeldung liefern
};
```

Die Member-Funktionen sind hier alle neudefiniert und die Funktion `what()` gibt eine modifizierte (ebenfalls implementierungsabhängige) Fehlermeldung zurück.

(Bei unserem GCC-Compiler lautet diese Meldung: `13bad_exception` und beim SUN-Workshop-Compiler: `Bad exception`.)

In dieser Headerdatei sind ebenfalls folgende, zum Teil bereits kennengelernten Typen und Funktionen deklariert:

- `typedef void (*unexpected_handler)();`

Zeigertyp auf eine Funktion zur Reaktion auf den Auswurf einer in der Ausnahmespezifikation einer Funktion nicht vorgesehenen Ausnahme.

- `unexpected_handler set_unexpected(unexpected_handler f) throw();`

Die angegebene Funktion `f` wird als Reaktion auf den Auswurf einer in der Ausnahmespezifikation einer Funktion nicht vorgesehenen Ausnahme installiert, Ergebnis ist die vordem installierte Funktion.

- `void unexpected();`

Funktion, welche vom System aufgerufen wird, wenn in einer Funktion eine Ausnahme ausgeworfen wird, welche nicht in der Ausnahmespezifikation vorgesehen ist.

Diese Funktion `unexpected` ruft die standardmäßig oder mittels `set_unexpected` installierte `unexpected_handler`-Funktion auf.

Die standardmäßig installierte `unexpected_handler`-Funktion sorgt für einen Programmabbruch.

- `typedef void (*terminate_handler)();`

Zeigertyp auf eine Funktion zur Reaktion auf eine nicht abgefangene Ausnahme.

– `terminate_handler set_terminate(terminate_handler f) throw();`

Die angegebene Funktion `f` wird als Reaktion auf eine nicht abgefangene Ausnahme installiert, Ergebnis ist die vormem installierte Funktion.

– `void terminate()`

Funktion, welche vom System aufgerufen wird, wenn eine Ausnahme nicht abgefangen wird.

Diese Funktion `terminate` ruft die standardmäßig oder mittels `set_terminate` installierte `terminate_handler`-Funktion auf.

Die standardmäßig installierte `terminate_handler`-Funktion sorgt für einen Programmabbruch.

– `bool uncaught_exception();`

liefert `true`, wenn eine noch nicht abgefangene Ausnahme vorliegt.

9.2 Die Headerdatei `stdexcept`

In der Headerdatei `<stdexcept>` ist eine Hierarchie weiterer, (auch indirekt) von der Klasse `exception` abgeleitete Fehlerklassen definiert, welche in der Standardbibliothek verwendet werden.

Alle hier vorgesehenen Fehlerklassen *fehlerklasse* haben neben der von der Klasse `exception` geerbten und standardmäßig vorhandenen Funktionalität zusätzlich einen expliziten Konstruktor:

```
explicit fehlerklasse::fehlerklasse(const string& what_arg);
```

mit einer Referenz auf einen konstanten String, mit dem die Fehlermeldung, welche durch die von `exception` geerbte Funktion `what()` zurückgibt, "gesetzt" werden kann.

9.2.1 Die Klasse `logic_error` und hiervon abgeleitete Fehlerklassen

Die Klasse

```
class logic_error: public exception {
public:
    explicit logic_error( const string& what_arg);
};
```

ist die Basisklasse für alle in der Logik eines Programms liegenden (also bei sorgfältiger Programmierung vermeidbarer) Fehler.

Alle von dieser Klasse abgeleiteten Fehlerklassen haben genau den gleichen Aufbau (innerhalb der geschweiften Klammern) wie diese Klasse, mit der Ausnahme, dass der explizite Konstruktor einen anderen Namen, nämlich den Klassennamen hat.

Von dieser Basisklasse sind im Standard folgende abgeleitete Klassen vorgesehen:

- `class domain_error: public logic_error { ... };`
Bereichsfehler.
- `class invalid_argument: public logic_error { ... };`
Unzulässiges Argument.
- `class length_error: public logic_error { ... };`
Konstruktion eines *zu großen* Objektes.
- `class out_of_range: public logic_error { ... };`
Zugriff mit unerlaubtem Index.

9.2.2 Die Klasse `runtime_error` und hiervon abgeleitete Fehlerklassen

Die Klasse

```
class runtime_error: public exception {  
    public:  
        explicit runtime_error( const string& what_arg);  
};
```

ist die Basisklasse für alle Laufzeitfehler eines Programms.

Alle von dieser Klasse abgeleiteten Fehlerklassen haben genau den gleichen Aufbau (innerhalb der geschweiften Klammern) wie diese Klasse, mit der Ausnahme, dass der explizite Konstruktor einen anderen Namen, nämlich den Klassennamen hat.

Von dieser Basisklasse sind im Standard folgende abgeleitete Klassen vorgesehen:

- `class range_error: public runtime_error { ... };`
interne Feldzugriffsfehler.
- `class overflow_error: public runtime_error { ... };`
arithmetischer Überlauf.
- `class underflow_error: public runtime_error { ... };`
arithmetischer Unterlauf.

9.3 Sonstige Standard-Fehlerklassen

9.3.1 Die Fehlerklasse `bad_alloc`

In der Headerdatei `<new>` ist die von `exception` abgeleitete Fehlerklasse

```

class bad_alloc: public exception {
public:
    bad_alloc() throw();           // parameterloser Konstruktor
    bad_alloc(const bad_alloc&) throw(); // Copy-Konstruktor

    bad_alloc& operator=(const bad_alloc&) throw(); // Zuweisung

    virtual ~bad_alloc() throw(); // Destruktor

    virtual const char * what() const throw(); // Fehlermeldung liefern
};

```

definiert. Diese wird im Zusammenhang mit unerfüllbaren Speicheranforderungen verwendet (vgl. Abschnitt 2.12!).

Die durch die Funktion `what()` gelieferte Fehlermeldung ist wiederum implementierungsabhängig.

(Bei unserem GCC-Compiler lautet diese Meldung: `bad_alloc` und beim SUN-Workshop-Compiler: `Out of Memory`.)

9.3.2 Die Fehlerklassen `bad_typeid` und `bad_cast`

In der Headerdatei `<typeinfo>` sind die von `exception` abgeleitete Fehlerklassen

```

class bad_typeid: public exception {
public:
    bad_typeid() throw();           // parameterloser Konstruktor
    bad_typeid(const bad_typeid&) throw(); // Copy-Konstruktor

    bad_typeid& operator=(const bad_typeid&) throw(); // Zuweisung

    virtual ~bad_typeid() throw(); // Destruktor

    virtual const char * what() const throw(); // Fehlermeldung liefern
};

```

und

```

class bad_cast: public exception {
public:
    bad_cast() throw();           // parameterloser Konstruktor
    bad_cast(const bad_cast&) throw(); // Copy-Konstruktor

    bad_cast& operator=(const bad_cast&) throw(); // Zuweisung

    virtual ~bad_cast() throw(); // Destruktor

    virtual const char * what() const throw(); // Fehlermeldung liefern
};

```

definiert. Ausnahmen des Types `bad_typeid` werden im Zusammenhang mit dem `typeid`-Operator und Ausnahmen vom Typ `bad_cast` im Zusammenhang mit dem `dynamic_cast`-Operator verwendet (vgl. Abschnitte 7.5.2 und 7.5.3).

Die durch die Funktion `what()` gelieferten Fehlermeldungen sind wiederum implementierungsabhängig.

(Bei unserem GCC-Compiler:

`bad_typeid::what()` liefert 10`bad_typeid`

`bat_cast::what()` liefert 8`bad_cast`

Beim SUN-Workshop-Compiler:

`bad_typeid::what()` liefert `Bad typeid`

`bat_cast::what()` liefert `Bad dynamic cast.`)

9.3.3 Die Fehlerklasse `ios_base::failure`

In der Headerdatei `<iostream>` ist in der allen Stream-Klassen zugrundeliegenden Klasse `ios_base` die von `exception` abgeleitete Fehlerklasse

```
class ios_base::failure: public exception {
public:
    explicit failure(const string& msg);           // Konstruktor mit
                                                    // Fehlermeldung

    virtual ~failure() throw();                   // Destruktor

    virtual const char * what() const throw(); // Fehlermeldung liefern
};
```

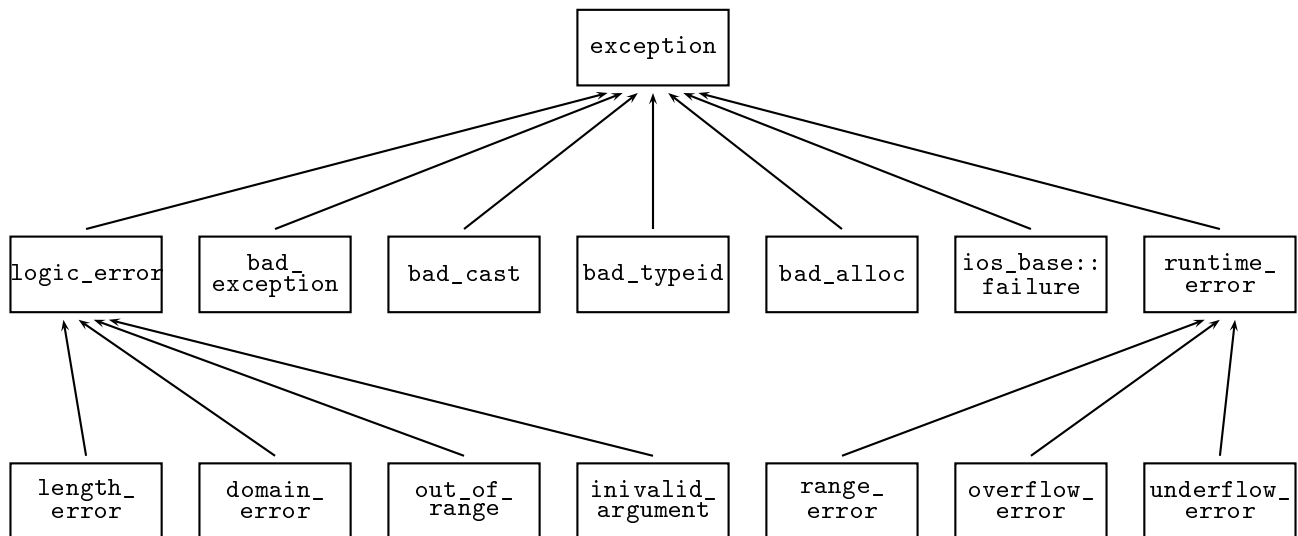
definiert, welche im Zusammenhang mit Fehlern bei Streams verwendet wird.

Bei der Konstruktion einer solchen Ausnahme muss eine konkrete Fehlermeldung angegeben werden, die dann durch die entsprechende `what()`-Funktion zurückgegeben wird:

```
...
// Nenner == 0?
if ( n == 0)
    throw ios_base::failure("Nenner gleich 0");
...
```

9.4 Übersicht über die Fehlerklassen der Standardbibliothek

Folgendes Schaubild gibt eine Übersicht über die Fehlerklassen der Standardbibliothek:



9.5 Fehlerklassen verwenden

Wie bereits gesehen kann man auch selber Standard-Ausnahmen auslösen und bei solchen, welche einen entsprechenden Konstruktor mit einem String-Argument haben, der Ausnahm eine Fehlermeldung mitgeben:

```

#include <stdexcept>
void fkt_08_15(...)
{
    ...
    if ( ... )
        throw out_of_range("Saudummer Zugriffsfehler in Funktion fkt_08_15");
    ...
}
  
```

Nach folgendem Muster kann man von den Standardfehlerklassen `logic_error`, `runtime_error` oder `ios_base::failure` selbst weitere Fehlerklassen ableiten, um sie in eigenen Programmen oder Bibliotheken zu verwenden:

```

...
class my_length_error: public length_error {
public:
    my_length_error(const string& str) : length_error(str) { };
};
  
```

Der Konstruktor mit String-Argument für die neue Klasse muss den entsprechenden Konstruktor der Basisklasse aufrufen!

Kapitel 10

C++-Strings

Der C++-Standard bietet zur einfachen Verarbeitung von Zeichenketten, *Strings* genannt (Folge/Feld von einzelnen Zeichen), eine reichhaltige Funktionalität — insbesondere sind die C++-Möglichkeiten umfangreicher als die Verarbeitung von Zeichenketten in C (mit `\0` terminierte `char`-Felder).

Wie bei den Strömen abstrahiert der Standard hierbei vom konkreten Zeichentyp, ist `CH` ein Typ, für welchen Zeicheneigenschaften vereinbart sind — für den es also eine Spezialisierung `char_traits<CH>` gibt —, so kann man Strings mit diesem Zeichentyp `CH` definieren.

Ein wesentliches Problem bei C-Zeichenfeldern war die Vermeidung von Feldüberläufen (etwa in ein `char`-Feld der Länge 50 eine Zeichenkette der Länge 100 hineinschreiben!).

Dieses Problem wird dadurch vermieden, dass die in C++ vorgesehenen String-Klassen-Objekte bei Bedarf automatisch dynamisch vergrößert werden.

Wegen dieses dynamischen Aspektes hat die Template-Klasse `basic_string<>`, welche allen C++-Stringklassen zugrundeliegt, drei Template-Parameter:

```
template <class CH, class traits, class allocator>
class basic_string { ... }
```

der erste gibt die Zeichenart an, der zweite legt die Zeicheneigenschaften des Types `CH` fest (hier werden als Default die in `char_traits<CH>` zur Zeichenart `CH` festgelegten Eigenschaften zugrundegelegt) und als drittes Argument kann die Art der dynamischen Speicherverwaltung festgelegt werden, mit der bei Bedarf Strings vergrößert bzw. verkleinert werden (hier wird als Default die Speicherverwaltung mittels `new`, `new[]`, `delete` und `delete[]` gewählt!).

Von dieser Template-Klasse `basic_string<>` gibt es im Standard bereits für die Standardzeichenarten `char` und `wchar_t` Instanziierungen, für welche im Standard die zusätzlichen Namen

```
typedef basic_string<char>  string;
typedef basic_string<char> wstring;
```

vergeben sind.

Zur Verwendung dieser Klassen muss die Headerdatei `<string>` includet werden!

Im Folgenden werde ich mich auf die Klasse `string` beschränken, also die Instantiierung der Template-Klasse `basic_string` für den Typen `char`.

Natürlich können gleichzeitig neben diesem neuen `string`-Typ wie in C auch einfache, durch das `'\0'`-Zeichen terminierte `char`-Felder und, nach Einbinden von `<cstring>`, die bekannten C-Funktionen (etwa `strcmp`, `strcpy`, ...) verwendet werden.

Im Folgenden bezeichne ich ein Objekt der (neuen) Klasse `string` mit *String* oder *C++-String* und durch `'\0'` terminierte `char`-Felder als *C-String*.

Der Standard sieht eine Reihe von Beziehungen zwischen dem "neuen" und dem "alten" Typ vor.

Beispiele für die einfache Verwendung von Strings haben wir bereits in Abschnitt 3.2 gesehen.

10.1 String-Iteratoren

Ein wesentlicher Unterschied zwischen C++-Strings und C-Strings ist der, dass C++-Strings nicht so einfach mit Zeigern bearbeitet werden können!

Folgende einfache Bearbeitung eines C-Strings

```
char w[] = "Superkalifragilistischexpialigorisch";
char *p;

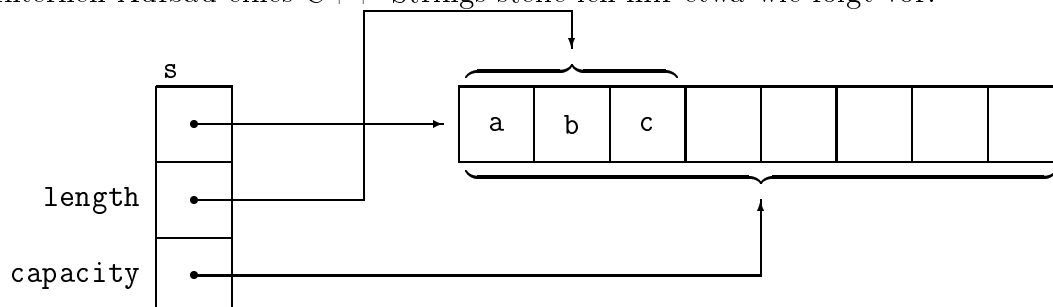
// jeden Buchstaben in Grossbuchstaben umwandeln:
for ( p = w; *p != '\0'; ++p)
    *p = toupper(*p);
...
```

ist für einen C++-String nicht mehr möglich!

Dies liegt daran, dass ein C++-String *s* intern völlig anders abgespeichert wird:

- C++-Strings sind eine Klasse mit dynamischen Komponenten,
- in ihnen wird ein dynamisch angelegtes Feld verwaltet, in dem die eigentliche Zeichenkette steht,
- zusätzlich wird im C++-String die Länge (`size()`) der gespeicherten Zeichenkette und die Größe des dynamischen Feldes (`capacity()`) verwaltet (Länge und Kapazität müssen nicht unbedingt gleich sein!).

Den internen Aufbau eines C++-Strings stelle ich mir etwa wie folgt vor:



Die eigentlichen Zeichen sind nicht direkt über den Namen des Strings zugreifbar, sondern über den Operator []:

```
string s("hallo");
char cs[]="hallo";

*cs = 'H'; // Zugriff auf erstes Zeichen des C-Strings
*s ... ; // FEHLER: *s nicht definiert

cs[0] = 'H'; // OK!
s[0] = 'H'; // OK!

if ( cs == &cs[0] ) // immer richtig!
{ ... }

if ( s == &s[0] ) // FEHLER: s hat Typ string,
{ ... }          // &s[0] hat den Typ char*
```

In der Standardbibliothek ist es vorgesehen, zeigerähnlich über die einzelnen Zeichen eines C++-Strings, etwa in einer Schleife, “laufen“ zu können.

Hierzu ist in der Klasse `string` ein Typ `string::iterator`, *Iterator* genannt, definiert und ein Iterator hat ähnliche Eigenschaften wie ein Zeiger:

- mittels des Operators `*` kann man über einen Iterator auf Zeichen eines C++-Strings zugreifen,
- mittels des Operators `++` kann man den Iterator erhöhen, so dass er aufs nächste Zeichen des C++-Strings “zeigt“,
- String-Iteratoren kann man mittels `==`, `!=`, `<`, ... vergleichen,
- ...

Wie ein Zeiger muss auch ein Iterator “initialisiert“ werden, damit er nicht “irgendwohin“ in die Gegend “zeigt“, sondern zunächst mal auf das erste Element eines Strings. Hierzu gibt es die String-Member-Funktionen

- `string::iterator string::begin();`
liefert “Iteratorposition“ des ersten Zeichens des Strings,
- `string::iterator string::end();`
liefert “Iteratorposition“ *eins hinter dem letzten* Zeichen des Strings.

Mittels dieses Iterator-Types und dieser Funktionen können dann auch sehr einfach Schleifen über alle Zeichen des C++-Strings formuliert werden (man beachte: das Element an der Position `begin()` gehört i. Allg. zum String, das mit der Position `end()` nicht mehr — insbesondere ist der `string s` leer, wenn die durch `s.begin()` und `s.end()` gelieferten Iteratorpositionen gleich sind!):

```

string s;
...
for ( string::iterator iter = s.begin(); iter < s.end(); ++iter )
{
    // *iter ist das aktuelle Zeichen des Strings
    ... // mach was mit *iter!
}
...

```

Um einen String auch rückwärts durchlaufen zu können, gibt es den entsprechenden *Rückwärtsiteratortyp*:

```
string::reverse_iterator;
```

und zugehörige Member-Funktionen:

- `string::reverse_iterator string::rbegin();`
liefert Position des letzten Zeichens des Strings,
- `string::reverse_iterator string::rend();`
liefert Position vor dem ersten Zeichens des Strings.

Ein `reverse_iterator` wird durch Erhöhen mittels `++` um eine Position “weitergestellt“, so dass auf das vorherige Zeichen des Strings “gezeigt“ wird!

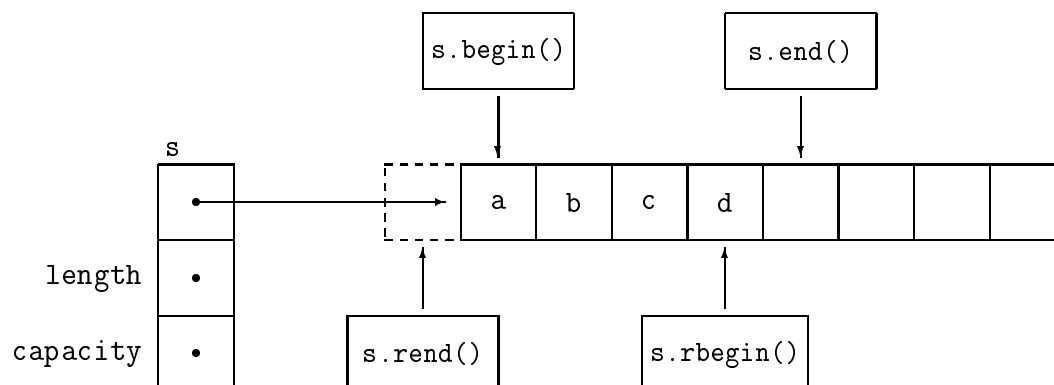
Mittels `reverse_iteratoren` sind Schleifendurchläufe von hinten nach vorne sehr einfach:

```

...
string s;
string::reverse_iterator r_iter;
...
for ( r_iter = s.rbegin(); r_iter != s.rend(); ++r_iter )
{
    // *riter ist aktuelle Zeichen des Strings
    ... // mach was mit *riter!
}
...

```

Schaubild: Iteratorpositionen bei einem C++-String `string s("abcd")`:



Genau wie man bei Zeigern zwischen *Zeigern* und *Zeigern auf const* unterscheiden muss, muss man auch bei Iteratoren zwischen *Iteratoren* und *Iteratoren auf const* unterscheiden, bei ersteren können die Elemente, auf die mittels des Iterators zugegriffen wird, verändert werden, bei zweiteren nicht!

Formal gibt es zur Klasse `string` folgende Iteratortypen und Funktionen, welche Iteratorpositionen liefern:

// Iterator-Typen:

```
string::iterator;           // Vorwaertsiterator fuer string
string::const_iterator;    // Vorwaertsiterator fuer const string
string::reverse_iterator;  // Rueckwaertsiterator fuer string
string::const_reverse_iterator; // Rueckwaertsiterator fuer const string
```

// Funktionen, welche Iteratorpositionen liefern:

```
// Anfangsposition fuer Vorwaertsiterator
string::iterator string::begin();
```

```
// Anfangsposition fuer Vorwaertsiterator fuer konstante Strings
string::const_iterator string::begin() const;
```

```
// Endeposition fuer Vorwaertsiterator
string::iterator string::end();
```

```
// Endeposition fuer Vorwaertsiterator fuer konstante Strings
string::const_iterator string::end() const;
```

```
// Anfangsposition fuer Rueckwaertsiterator
string::reverse_iterator string::rbegin();
```

```
// Anfangsposition fuer Rueckwaertsiterator fuer konstante Strings
string::const_reverse_iterator string::rbegin() const;
```

```
// Endeposition fuer Rueckwaertsiterator
string::reverse_iterator string::rend();
```

```
// Endeposition fuer Rueckwaertsiterator fuer konstante Strings
string::const_reverse_iterator string::rend() const;
```

10.2 Größe und Kapazität eines Strings

Strings stellen eine Klasse mit dynamischen Komponenten dar — ein String wird bei Bedarf vergrößert (ggf. auch verkleinert).

Ein String hat eine gewisse Länge (Anzahl der abgespeicherten Zeichen) — für solche Längenangaben (oder auch Positionsangaben eines Zeichens im String) gibt es einen

zur Klasse `String` passenden vorzeichenlosen, ganzzahligen Typen `size_type`, der im Umfeld der `String`-Klasse definiert ist.

Ebenfalls definiert ist zur Klasse `string` eine Konstante `npos` dieses Types `size_type`, welche den größtmöglichen Wert dieses Types `size_type` repräsentiert! (Dieser größtmögliche Wert selbst ist als Positions- oder Längenangabe nicht mehr erlaubt — durch ihn werden Fehlerfälle gekennzeichnet!)

Die `string`-Member-Funktionen

```
size_type string::size() const;
size_type string::length() const;
```

liefern zu einem `String` die Anzahl der aktuell im `String` abgelegten Zeichen und die Member-Funktion

```
size_type string::capacity() const;
```

liefert die maximale Anzahl von Zeichen, die der `String` ohne zusätzliche Speicheranforderung aufnehmen kann.

Die Member-Funktion

```
bool string::empty() const;
```

liefert als Ergebnis die Antwort auf die Frage, ob der `String` augenblicklich leer ist oder nicht.

Die Funktion

```
size_type string::max_size() const;
```

liefert die maximal auf dem System mögliche Größe des `Strings`, i. Allg. also `npos-1`. (Auf Rechnern mit kleinem Speicher kann auch ein kleinerer Wert herauskommen!)

Die Funktionen

```
void string::resize( size_type n);
void string::resize( size_type n, char c);
```

ändern die Länge des `Strings` auf `n`.

Ist hierbei `n` kleiner als die bisherige Länge `size()`, so werden die restlichen Zeichen aus dem `String` entfernt.

Ist `n` größer als `size`, werden am `String`ende entsprechend viele Zeichen angehängt, so dass die neue Länge gleich `n` ist.

Bei der ersten Funktion `void string::resize(size_type n);` werden die zusätzlichen Zeichen mit “dem `char`-Standardkonstruktor“ erzeugt, bei der zweiten Funktion erhalten die zusätzlichen Zeichen den Wert des zweiten Argumentes `c`.

Ist `n` größer als die maximale Größe `max_size()` des `Strings`, wird eine Ausnahme des im Standard in `<stdexcept>` definierten Types `length_error` ausgelöst.

Zu beachten ist, dass durch den Aufruf dieser Funktion `resize` Iteratorpositionen ungültig werden, so dass etwa Schleifen folgender Art vermieden werden sollten:

```
string s(...);
string::iterator iter;
...
```

```

for ( iter = s.begin(); iter != s.end(); ++iter)
{
    ...           // Problem: String wird geaendert und dadurch
    s.resize(...); // werden Iteratorpositionen ungueltig!
    ...           // Fazit: Schleife laeuft unkontrolliert ab!
}
...

```

Durch Aufruf der Funktion

```
void string::reserve(size_type n);
```

teilt man dem System mit, dass man beabsichtigt, (irgendwann später, auch wenn der String zur Zeit kürzer oder leer ist) *n* Zeichen in dem String abzulegen.

Das System vergrößert den internen Speicher für den String, ohne den eigentlichen Inhalt zu ändern. (Hierdurch werden später “teure“ dynamische Vergrößerungen unnötig! Achtung: Auch hierbei werden Iteratorpositionen ggf. ungültig!)

Im Allgemeinen sollte das Argument *n* größer als die augenblickliche Stringlänge `size()` sein!

Ist das Argument *n* größer als die maximale Größe `max_size()` des Strings, wird wiederum eine Ausnahme vom Typ `length_error` ausgelöst.

10.3 Erzeugen/Zerstören von C++-Strings

Es sind eine Reihe von Konstruktoren der Klasse `string` vorgesehen.

Die Wichtigsten:

- `string::string();`

parameterloser Konstruktor, erzeugt einen leeren String, Anwendung:

```
string s;
```

- `string::string(const char *cs);`

erzeugt neuen String und initialisiert diesen mit einem C-String, etwa:

```

char w[100] = "hallo";    // char-Feld
char *p = w;              // char-Zeiger

string s1("hallo"); // Initialisierung mit Zeichenkettenliteral
string s2(w);       // Initialisierung mit char-Feld ('\0' am Ende)
string s2(p);       // Initialisierung mit char-Zeiger, Zeiger darf
                    // nicht 0 sein und muss auf '\0'-terminiertes
                    // char-Feld zeigen!

```

Man beachte, dass hiermit eine Typumwandlung von C-Strings nach C++-Strings definiert ist — dass also überall in Funktionsaufrufen, wo ein C++-String erwartet wird, auch ein C-String stehen darf!

- `string::string(const char *cs, size_type n);`

erzeugt neuen String und initialisiert ihn mit den ersten `n` Zeichen des C-Strings `cs`, etwa:

```
string s("hallo", 3); // initialisiert mit "hal"
```

Hier ist der Programmierer selbst dafür verantwortlich, dass das zweite Argument `n` nicht größer als die Stringlänge `strlen(cs)` des ersten Argumentes ist. Ist `n` gleich der Konstanten `npos`, wird eine Ausnahme des Types `length_error` ausgelöst!

- `string::string(const string &s);`

Copy-Konstruktor, erzeugt einen C++-String als Kopie eines anderen, Anwendung:

```
string s1("hallo"); // Konstruktor aus char *
string s2(s1);      // Copy-Konstruktor
```

- `string::string(const string &s, size_type pos);`

Copy-Konstruktor mit “Anfangsindex“, erzeugt neuen C++-String, wobei der Inhalt des neuen Strings der des alten Strings `s` ab der angegebenen Position `pos` ist.

Ist hierbei `pos` größer als `s.size()`, wird eine Ausnahme des im Standard in `<stdexcept>` definierten Types `out_of_range` ausgeworfen, ist `pos` gleich `s.size()`, wird ein leerer String erzeugt:

```
string s1("hallo");
string s2(s1, 3); // initialisiert mit "lo"
string s3(s1,100); // -> Ausnahme: out_of_range
```

- `string::string(const string &s, size_type pos, size_type n);`

Copy-Konstruktor mit “Anfangsindex“ `pos` und “Länge“ `n`, erzeugt einen neuen String, wobei der Inhalt des neuen Strings aus der Teilzeichenkette von `s` ab Zeichen mit Position `pos` und Länge `n` besteht. Einschränkungen:

- Ist `pos` größer als `s.size()` wird wiederum eine `out_of_range`-Ausnahme ausgelöst.
- Ist `n` größer als `s.size()-pos`, wird `s.size()-pos` anstelle von `n` genommen (d.h. der neue String wird mit dem an Position `pos` beginnenden Rest von `s` initialisiert!).

Mittels der durch `string::string(const char *cs);` definierten “Typumwandlung“ von C-Strings nach C++-Strings kann dieser Konstruktor auch mit einem C-String als Argument aufgerufen werden!


```

string s1("hallo");
string s2(s1, 2, 2);    // initialisiert mit "ll"
string s3(s1, 2,10);    // initialisiert mit "llo"
string s4(s1, 2, 1);    // initialisiert mit "l"
string s5(s1,10,20);    // -> Ausnahme: out_of_range
string s6("hallo", 2,1); // aus C-String "hallo" wird ein
                        // C++-String und mit dessen Teilstring
                        // von Position 2 mit Laenge 1 wird der
                        // neu erzeugte String initialisiert!

```

– `string::string(size_type n, char c);`

Initialisiert neuen String mit `n` mal dem Zeichen `c`:

```
string s(10, ' '); // mit 10 Leerzeichen initialisiert
```

Auch diese Funktion wirft eine `length_error`-Ausnahme, falls das erste Argument `n` gleich der Konstante `npos` ist!

– `template <class iter> string::string(iter Anfang, iter Ende);`

Initialisiert neuen String mittels der durch die beiden Iteratorpositionen `Anfang` und `Ende` gegebenen *Sequenz* von Zeichen.

Der Typ `iter` muss hierbei irgendein Iterortyp sein (nicht unbedingt ein String-Iterortyp), bei dem der Zugriffsoperator `*` ein Zeichen (oder ein Objekt, welches nach `char` umgewandelt werden kann) liefert.

Der Destruktor

```
string::~~string();
```

sorgt für das ordnungsgemäße Zerstören eines Strings (inklusive dynamischer Komponenten).

10.4 Zugriff auf einzelne Zeichen eines Strings

Für die Klasse `string` ist der Feldzugriffsoperator `[]` definiert, mit dem man auf ein Zeichen des Strings zugreifen kann:

```

char  string::operator[](size_type n) const;
char& string::operator[](size_type n);

```

Die erste Funktion ist für konstante Strings gedacht, zurückgegeben wird der Wert des `n`-ten Zeichens des Strings, die zweite ist für variable Strings vorgesehen, zurückgegeben wird die Referenz auf das `n`-te Zeichen im String — das zurückgegebene Zeichen könnte also geändert werden:

```

const string cs("hallo");    // konstanter String!
string s("hallo");           // variabler String!
...

```

```
cs[0] = 'H';      // FEHLER: cs[0] kann nicht ge"ander werden!
s[0] = 'H';      // OK, s st jetzt "Hallo"!
...
```

Aus Performancegründen wird der Zugriff über diese Operatorfunktionen nicht daraufhin überprüft, ob der Index `n` im erlaubten Bereich von 0 bis `size()-1` ist — der Programmierer ist für den korrekten Indexzugriff verantwortlich. Ggf. sind (wie in C) merkwürdige Laufzeitfehler die Folge.

Alternativ ist ein geprüfter Elementzugriff über die Funktionen

```
const char & string::at(size_t n) const;
char & string::at(size_t n);
```

möglich.

Liegt hier bei einem Aufruf `s.at(n)` der "Index" `n` nicht im Bereich von 0 bis `s.size()-1`, wird eine Ausnahme vom Typ `out_of_range` ausgelöst.

Der Zugriff auf die einzelnen Zeichen eines C++-Strings ist, wie in Abschnitt 10.1 bereits erläutert, auch mittels Iteratoren möglich!

10.5 Zuweisungen an einen String

Aufgrund der dynamischen Komponenten ist für einen String der Zuweisungsoperator vernünftig definiert — es gibt sogar mehrere, sich in der Signatur unterscheidende Zuweisungsoperatoren:

- Zuweisung eines C++-Strings an einen C++-String:

```
string& string::operator=( const string &);
```

- Zuweisung eines C-Strings an einen C++-String:

```
string& string::operator=(const char *);
```

- Zuweisung eines Zeichens an einen C-String:

```
string& string::operator=(char c);
```

Der String, dem etwas zugewiesen wurde, bekommt Zeichenkette der Länge 1 mit diesem Zeichen `c` zum Inhalt!

Beispiele:

```
string s1(...), s2;    // C++-Strings
char w[] = ...;       // muss C-String enthalten!
char *p = ...;        // muss auf C-String zeigen!
char c;

s2 = s1 ;             // Zuweisung eines C++-Strings an C++-String
```

```

s2 = "hallo"; // Zuweisung eines C-Strings an einen C++-String
s2 = w;      // Zuweisung eines C-Strings an einen C++-String
s2 = p;      // Zuweisung eines C-Strings an einen C++-String

s2 = 'A';    // Zuweisung eines Zeichens an einen C++-String
s2 = c;      // Zuweisung eines Zeichens an einen C++-String

```

Es gibt zusätzlich folgende Zuweisungsfunktionen, welche den obigen Zuweisungsoperatoren bzw. Konstruktoren mit gleicher Signatur entsprechen:

- `string& string::assign(const string &);`
entspricht: `string& string::operator=(const string &)`
- `string& string::assign(const char *);`
entspricht: `string& string::operator=(const char *)`
- `string& string::assign(const string& s, size_type pos, size_type n);`
dem String wird der Teilstring des Strings `s` ab Position `pos` der Länge `n` zugewiesen.
Das Verhalten ist analog zum Konstruktor
`string::string(const string& s, size_type pos, size_type n):`

- es wird eine `out_of_range`-Ausnahme ausgeworfen, falls `pos` größer als `s.size()` ist,
- falls `n` größer als `s.size()-pos` ist, wird mit `s.size()-pos` anstelle von `n` gearbeitet, d.h. es wird der an Position `pos` beginnende Rest von `s` zugewiesen.

Mittels der durch den Konstruktor `string::string(const char*)` gegebenen "Typumwandlung" von C-Strings nach C++-Strings kann diese Funktion auch mit einem C-String als Argument aufgerufen werden!

- `string& string::assign(const char *cs, size_t n);`
die ersten `n` Zeichen des C-Strings `cs` werden zugewiesen.
Ungeprüfter Laufzeitfehler, falls `n` größer als `strlen(cs)` ist, eine Ausnahme vom Typ `length_error` wird ausgeworfen, falls `n` gleich der Konstanten `npos` ist!
- `string& string::assign(size_type n, char c);`
zugewiesen wird eine Folge der Länge `n` aus lauter Zeichen `c`. Falls wiederum `n` gleich `npos` gilt, wird eine `length_error`-Ausnahme ausgeworfen.
- `template <class iter> string& string::assign(iter Anfang, iter Ende);`
weist einem String die durch die beiden Iteratorpositionen `Anfang` und `Ende` gegebene Sequenz von Zeichen zu.

Beispiele für die Verwendung von `assign`:

```
string s1("hallo");
string s2;
char w[] = ...;           // muss C-String enthalten!
char *p = ...;            // muss auf C-String zeigen!
char c;
...
s2.assign(s1);             // C++-string s1 wird C++-String s2 zugewiesen
...
s2.assign("hallo");        // C-String "hallo" wird C++-String s2 zugewiesen
s2.assign(w);              // C-String w wird C++-String s2 zugewiesen
s2.assign(p);              // C-String p wird C++-String s2 zugewiesen
...
s2.assign(s1, 2, 3);       // zugewiesen wird "ll"
s2.assign(s1, 2,10);       // zugewiesen wird "llo"
s2.assign(s1, 2, 1);       // zugewiesen wird "l"
s2.assign(s1,10,20);       // -> Ausnahme: out_of_range
...
s2.assign("hallo",3);      // zugewiesen wird "hal"
...
s2.assign(10, ' ');        // 10 Leerzeichen zuweisen
...
s2.assign(s1.begin(), s1.end()); // Zuweisung ueber Iteratoren,
...                          // entspricht: s2.assign(s1)
```

10.6 C++-Strings wie C-Strings verwenden

Ein C++-String ist kein C-String, es gibt aber Möglichkeiten, einen C++-String in einen C-String umzuwandeln.

Die Funktion:

```
const char * string::c_str() const;
```

liefert die Adresse eines `'\0'`-terminierten C-Strings, dessen Inhalt mit dem des C++-Strings übereinstimmt. Der resultierende C-String wird vom String selbst verwaltet und kann vom Anwender nicht verändert werden, da das Funktionsergebnis von `c_str` ein Zeiger auf `const` ist.

Die Funktion:

```
const char * string::data() const;
```

liefert, falls der String nicht die Länge 0 hat (`size() != 0`), die Adresse eines `char`-Feldes, dessen ersten `size()` Elemente mit den ersten Zeichen des C++-Strings übereinstimmen (das `char`-Feld ist jedoch nicht notwendigerweise durch `'\0'` terminiert!). Hat der C++-String die Länge 0, so liefert diese Funktion `data()` den `const char *`-Wert 0 (ungültige Adresse).

Auch bei dieser Funktion kann der Anwender das Zeichenfeld nicht ändern!

Die Funktion:

```
size_type string::copy( char *s, size_type n, size_type pos);
```

kopiert maximal `n` Zeichen aus dem Inhalt des Strings ab Position `pos` in das `char`-Feld, auf dessen Anfang der Zeiger `s` zeigt.

Ist `pos` größer als die Länge `size()` des C++-Strings, so wird eine `out_of_range`-Ausnahme ausgeworfen.

Das Kopieren endet spätestens mit dem Ende des C++-Strings.

An das `char`-Feld mit Adresse `s` wird kein `'\0'` angehängt und das Feld muss groß genug sein, um die Zeichen aufzunehmen.

Mit dieser Funktion hat man die Möglichkeit, eine C-String-Kopie eines C++-Strings anzulegen und diese Kopie zu bearbeiten. Natürlich bleibt durch die Bearbeitung der C-Kopie der originale C++-String unverändert!.

10.7 Vergleiche von Strings

Zum Vergleich stehen für C++-Strings Member-Funktionen mit dem Namen `compare` (und unterschiedlicher Signatur) und eine Reihe von globalen Operator-Funktionen (welche auf entsprechende Member-Funktionen zurückgeführt werden) zur Verfügung. (Bei allen Vergleichen von Zeichenketten wird der in den `char_traits` festgelegte Vergleich einzelner Zeichen zugrundegelegt, d.h. zwei nicht identische Zeichen können ggf. bezüglich des dort definierten Vergleiches durchaus “gleich” sein! Für den Zeichentyp `char` wird hier der übliche Vergleich anhand des Maschinenzeichensatzes zugrundegelegt!)

10.7.1 String-Member-Funktionen `compare`

Beim Vergleich zweier Zeichenketten mit `compare` wird das Ergebnis 0 geliefert, wenn die beiden Zeichenketten gleich sind, das Ergebnis ist negativ, falls die erste Zeichenkette lexikographisch kleiner als die zweite ist, ansonsten ist das Ergebnis positiv!

Verglichen werden die durch die unterschiedlichen Stringtypen dargestellten eigentlichen Zeichenketten.

Die unterschiedlichen `compare`-Funktionen unterscheiden sich in ihrer Signatur:

```
- int string::compare( string &str);
```

Vergleich zweier `string`-Objekte, etwa:

```
string s1, s2;
...
if ( s1.compare(s2) < 0 )
{ ... }
...
```

```
- int string::compare( size_type pos1, size_type n1, string &str);
```

vergleicht den an `pos1` beginnenden Teilstring der Länge `n1` des aktuellen Strings mit dem im Argument angegebenen String `str`.

```
- int string::compare( size_type pos1, size_type n1, string &str,
                      size_type pos2, size_type n2);
```

vergleicht den an Position `pos1` beginnenden Teilstring der Länge `n1` des aktuellen Strings mit dem an Position `pos2` beginnenden Teilstring der Länge `n2` des im Argument angegebenen Strings `str`.

Mittels der durch den Konstruktor `string::string(const char*)` gegebenen "Typumwandlung" von C-Strings nach C++-Strings kann diese Funktion auch mit einem C-String als Argument aufgerufen werden!

```
- int string::compare(const char *cs);
```

vergleicht den aktuellen C++-String mit dem als Argument angegebenen C-String.

```
- int string::compare(size_type pos1, size_type n1,
                      const char *cs, size_type n);
```

vergleicht den an Position `pos1` beginnenden Teilstring der Länge `n1` des aktuellen Strings mit dem aus den ersten `n` Zeichen bestehenden Teilstring des als Argument angegebenen C-Strings. (Ungeprüfter Laufzeitfehler, falls `n` größer als `strlen(cs)` ist!)

Zu beachten ist, dass das Objekt, für welches eine dieser `compare`-Funktionen aufgerufen wird, ein C++-String sein muss und dass, wann immer über Positionsangabe `pos` und Längenangabe `n` auf einen Teilstring eines Strings `s` zugegriffen wird, eine `out_of_range`-Ausnahme ausgeworfen wird, falls `pos` größer als `s.size()` ist und mit `s.size()-pos` anstelle von `n` gearbeitet wird, falls `n` größer als `s.size()-pos` ist!

10.7.2 Globale Operator-Funktionen zum Vergleich

Folgende (globale) Vergleichs-Operator-Funktionen für Strings liefern einen Wahrheitswert und werden auf die entsprechenden `compare`-Member-Funktionen zurückgeführt:

```
- bool operator== ( string &a, string &b);
```

gibt das Ergebnis des Vergleichs `a.compare(b) == 0` als Funktionsergebnis zurück.

Man beachte, dass mittels des Konstruktors `string::string(const char *)` ein C-String in einen C++-String umgewandelt werden kann und dass somit mittels dieser Operator-Funktion folgende Vergleiche möglich sind:

```
string s1, s2;
char w[] = "hallo";    // '\0' terminiertes char-Feld
char *p = w;           // zeigt auf '\0'-terminiertes char-Feld
...
if ( s1 == s2 ){...}   // Vergleich zweier C++-Strings
```

```

if ( s1 == w ) {...} // Vergleich: C++-Strings mit einem C-String
if ( p == s1 ) {...} // Vergleich: C-Strings mit einem C++-String
if ( s2 == "hallo" ) // Vergleich: C-Strings mit einem C++-String
{ ... }

```

Wichtig ist: mindestens einer der beteiligten Operanden muss ein C++-String sein (da als globale Funktion realisiert, wird ggf. auch im ersten Argument eine Typumwandlung von C-String nach C++-String durchgeführt)!

- `bool operator!= (string &a, string &b);`
gibt als Funktionsergebnis `!(a == b)` zurück.
- `bool operator< (string &a, string &b);`
gibt das Ergebnis des Vergleichs `a.compare(b) < 0` als Funktionsergebnis zurück.
- `bool operator> (string &a, string &b);`
gibt das Ergebnis des Vergleichs `a.compare(b) > 0` als Funktionsergebnis zurück.
- `bool operator<= (string &a, string &b);`
gibt das Ergebnis des Vergleichs `a.compare(b) <= 0` als Funktionsergebnis zurück.
- `bool operator>= (string &a, string &b);`
gibt das Ergebnis des Vergleichs `a.compare(b) >= 0` als Funktionsergebnis zurück.

Wie beim `==`-Operator braucht bei all diesen anderen Operatoren nur einer der beteiligten Operanden ein C++-String zu sein, der andere (u.U. auch der erste Operand) kann ein C-String sein!

10.8 Einfügen, Anhängen, Verketteten

Zum Einfügen, Anhängen und Hintereinanderhängen gibt es ebenfalls eine Reihe von Funktionen und Operatoren, die hier kurz vorgestellt werden sollen:

10.8.1 Einfügen in einen String

Zum Einfügen von Zeichenketten in einen String steht die Member-Funktion `insert` (mit verschiedenen Signaturen) zur Verfügung. All diese Funktionen geben den (geänderten) String selbst als Funktionsergebnis zurück!

Teilweise wird in folgenden Funktionen über zwei Argumente `size_type pos` und `size_type n` auf den an Position `pos` beginnenden Teilstring der Länge `n` zugegriffen. Bei derartigen Zugriffen wird wie immer verfahren:

- Ist `pos` größer als `s.size()`, wird eine `out_of_range`-Ausnahme ausgelöst.
- Ist `n` größer als `s.size()-pos`, wird `s.size()-pos` anstelle von `n` genommen.

Es gibt folgende `insert`-Funktionen:

- `string& string::insert(size_type pos1, const string & str);`

fügt in den aktuellen String vor der durch `pos1` angegebenen Position eine Kopie des als Argument angegebenen Strings `str` ein.

Ist `pos1` gleich 0, so kommt die eingefügte Kopie also vor den bisherigen Inhalt des aktuellen Strings.

Ist `pos1` gleich `size()`, so wird die Kopie hinten an den bisherigen Inhalt des aktuellen Strings angehängt.

Ist `pos1` größer als `size()`, wird eine Ausnahme `out_of_range` ausgeworfen:

```
string s1("abcdef");
string s2("123456");
string s3;
...
s3 = s1;
cout << s3.insert(0,s2);           // -> "123456abcdef"
...
s3 = s1;
cout << s3.insert(3,s2);           // -> "abc123456def"
...
s3 = s1;
cout << s3.insert(s3.size(),s2);    // -> "abcdef123456"
...
s3 = s1;
cout << s3.insert(100, s2);         // -> Ausnahme: out_of_range
...
```

- `string& string::insert(size_type pos1, const string & str, size_type pos2, size_type n2);`

in den aktuellen String wird vor der durch `pos1` ($\leq \text{size}()$, ansonsten Ausnahme `out_of_range`) spezifizierten Stelle der an Position `pos2` beginnende Teilstring der Länge `n2` des Strings `str` eingefügt.

Mittels der durch den Konstruktor `string::string(const char*)` gegebenen "Typumwandlung" von C-Strings nach C++-Strings kann diese Funktion auch mit einem C-String als Argument aufgerufen werden!

- `string& string::insert(size_type pos1, const char *cs);`

fügt vor der Position `pos1` des aktuellen Strings eine Kopie des C-Strings `cs` ein.

- `string& string::insert(size_type pos1, const char *cs, size_type n);`
 fügt vor die Position `pos1` ($\leq \text{size}()$, ansonsten `out_of_range`-Ausnahme) des aktuellen Strings (maximal) `n` Zeichen des C-Strings `cs` ein. (Ungeprüfter Laufzeitfehler, falls `n` größer als `strlen(cs)` ist!)
- `string& string::insert(size_type pos1, size_type n, char c);`
 fügt vor der Position `pos1` ($\leq \text{size}()$, ansonsten `out_of_range`-Ausnahme) des aktuellen Strings eine Zeichenkette ein, welche aus `n` aufeinanderfolgenden Zeichen `c` besteht.
- `string& string::insert(string::iterator p, size_type n, char c);`
 fügt in den aktuellen String vor das durch die Iteratorposition `p` gegebene Zeichen eine Zeichenkette ein, welche aus `n` aufeinanderfolgenden Zeichen `c` besteht. Die Iteratorposition muss eine für den String gültige Position zwischen `begin()` oder `end()` (beide eingeschlossen) sein!
- `template <class iter>`
`string& string::insert(string::iterator p, iter Anfang, iter Ende);`
 (`p` wie oben) fügt vor der durch `p` gegebenen Iteratorposition die durch `Anfang` und `Ende` gegebene Sequenz von Zeichen ein.

Bei all diesen Funktionen wird eine `length_error`-Ausnahme ausgelöst, falls der resultierende String zu groß für das System wird!

Durch das Einfügen von neuen Zeichen in den String werden vorherige Iteratorpositionen ggf. ungültig.

10.8.2 Anhängen an einen String

Zum Einfügen am Stringende gibt es spezielle (möglicherweise effizienter als `insert` implementierte) Member-Funktionen mit dem Namen `append`:

- `string& string::append(const string &str);`
 Der Aufruf: `s1.append(s2)` entspricht: `s1.insert(s1.size(), s2)`.
- `string& string::append(const string &str, size_type pos, size_type n);`
`s1.append(s2, pos, n)` entspricht `s1.insert(s1.size(), s2, pos, n)`.
 Mittels der durch den Konstruktor `string::string(const char*)` gegebenen "Typumwandlung" von C-Strings nach C++-Strings kann diese Funktion auch mit einem C-String als Argument aufgerufen werden!
- `string& string::append(const char* cs);`
`s1.append("hallo")` entspricht: `s1.insert(s1.size(), "hallo")`.

```

- string& string::append(const char* cs, size_type n);
  s1.append("hallo", n) entspricht: s1.insert(s1.size(), "hallo", n).

- string& string::append(size_type n, char c);
  s1.append(n, c) entspricht: s1.insert(s1.size(), n, c).

- template <class iter>
  string& string::append(iter Anfang, iter Ende);
  s1.append(Anfang, Ende) entspricht s1.insert(s1.end(), Anfang, Ende);.

```

Bei `append` können gleichartige Fehler wie bei den entsprechenden `insert`-Funktionen auftreten.

Iteratorpositionen werden gegebenenfalls ebenfalls ungütig.

Zum Anhängen eines einzelnen Zeichens kann die Member-Funktion:

```
void string::push_back(char c);
```

verwendet werden.

Auf die entsprechende `append`-Funktionen zurückgeführt werden folgende Member-Operator-Funktionen zum Anhängen an einen C++-String:

```

- string& string::operator+=(const string &str);
  hängt an den aktuellen String eine Kopie des als Argumentes angegebenen
  C++-Strings str an.

- string& string::operator+=(const char* cs);
  hängt an den aktuellen String eine Kopie des als Argumentes angegebenen C-
  Strings cs an.

- string& string::operator+=(char c);
  hängt an den aktuellen String das angegebene Zeichen c an.

```

10.8.3 Strings verketten

Der Operator `+` hängt zwei C++-Strings zu einem (temporären) C++-String zusammen (*Verkettung*):

```
string operator+(const string &a, const string &b);
```

Da als globale Funktion realisiert, kann dieser Operator auch dazu verwendet werden, einen C++-String und einen C-String (oder umgekehrt) aneinanderzuhängen:

```

string s1(...), s2(...), s3;
char w[] = ...;      // '\0'-terminiert
...
s3 = s1 + s2;        // Verkettung zweier C++-Strings
...
s3 = s1 + w;         // Verkettung eines C++-Strings mit einem C-String
s3 = w + s1;         // Verkettung eines C-Strings mit einem C++-String

```

```
...
s3 = s1 + "hallo"; // Verkettung eines C++-Strings mit einem C-String
...
```

Damit man auch einen String und ein einzelnes Zeichen mit dem Operator `+` verketteten kann, gibt es noch die beiden (globalen) Funktionen:

```
string operator+(const string &a, char c)
```

(String plus Zeichen) und

```
string operator+(char c, const string &a)
```

(Zeichen plus String).

10.9 Teilstrings

10.9.1 Auf Teilstrings zugreifen

Auf einen Teilstring (lesend) zugreifen kann man mit der Member-Funktion:

```
string string::substr(size_type pos = 0, size_type n = npos);
```

Wie üblich ist `pos` die Anfangsposition des Teilstrings und `n` die Länge des Teilstrings. (Und wie immer wird, falls `pos` größer als `size()` ist, eine `out_of_range`-Ausnahme ausgeworfen, und falls `n` größer als `size()-pos` ist, wird `size()-pos` anstelle von `n` genommen.)

Falls man keine Argumente angibt, erhält man (wegen der Default-Argumente) den ganzen String.

10.9.2 Teilstrings löschen

Mittels der Funktion

```
string string::erase(size_type pos = 0, size_type n = npos);
```

kann man den an Position `pos` beginnenden Teilstring der Länge `n` löschen. Funktionsergebnis ist der aktuelle String nach dem Löschen.

(Wie immer wird, falls `pos` größer als `size()` ist, eine `out_of_range`-Ausnahme ausgeworfen, und falls `n` größer als `size()-pos` ist, wird `size()-pos` anstelle von `n` genommen. Es macht allerdings wenig Sinn, einen leeren Teilstring zu löschen!)

Ohne Argumente aufgerufen wird (wegen der Default-Argumente) der ganze String gelöscht, der String ist also anschließend leer.

Alternativ kann man hierzu auch die Funktion:

```
void string::clear();
```

verwenden.

Die Funktion

```
string::iterator string::erase(string::iterator p);
```

löscht das Zeichen, auf das der Iterator `p` zeigt. Beim Aufruf:

```
string s(...);
string::iterator p;
```

```
...
p = erase(p);
...
```

muss allerdings der Iterator `p` eine gültige Iteratorposition innerhalb des aktuellen String `s` innehaben, d.h. muss zwischen `s.begin()` (einschließlich) und `s.end()` (ausschließlich) liegen. Funktionsergebnis ist die Iteratorposition des Zeichens im String, welches ursprünglich hinter dem gelöschten stand (bzw. `s.end()`, falls das letzte Zeichen gelöscht wurde!).

Eine ganze Teilsequenz eines Strings kann man mit der Funktion

```
string::iterator erase(string::iterator first, string::iterator last);
```

(`first` und `last` müssen gültige Iteratorpositionen im aktuellen String sein!)

Gelöscht werden alle Elemente zwischen `first` (einschließlich) und `last` (ausschließlich). Funktionsergebnis ist die Iteratorposition des ersten nicht mehr gelöschten Elementes. (Da bei Löschen sich die Iteratorpositionen ändern, ist das Funktionsergebnis i. Allg. nicht gleich `last`!)

10.9.3 Einen Teilstring ersetzen

Zum Ersetzen eines Teilstrings kann die Funktion `replace` verwendet werden.

Bei den folgenden fünf Funktionen wird der zu ersetzende Teilstring durch Anfangsposition `size_type pos1` und Länge `size_type n1` beschrieben (ersten beiden Argumente).

Wie üblich wird eine `out_of_range`-Ausnahme ausgelöst, wenn `pos1` größer als `size()` ist und falls `n1` größer `size()-pos1` ist, wird mit `size()-pos1` anstelle von `n1` gearbeitet.

```
- string& string::replace(size_type pos1, size_type n1,
                        const string &s)
```

ersetzt den durch Anfangsposition `pos1` und Länge `n1` gegebenen Teilstring des aktuellen Strings durch den als drittes Argument angegebenen C++-String `s`.

```
- string& string::replace(size_type pos1, size_type n1,
                        const string &s, size_type pos2, size_type n2);
```

ersetzt den durch Anfangsposition `pos1` und Länge `n1` beschriebenen Teilstring des aktuellen Strings durch den durch Anfangsposition `pos2` und Länge `n2` gegebenen Teilstring des als drittes Argument angegebenen C++-String `s`. (Auch für `pos2` und `n2` gelten die üblichen Einschränkungen!)

Mittels der durch den Konstruktor `string::string(const char*)` gegebenen "Typumwandlung" von C-Strings nach C++-Strings kann diese Funktion auch mit einem C-String als Argument aufgerufen werden!

```
- string& string::replace(size_type pos1, size_type n1,
                        const char* cs);
```

ersetzt den durch Anfangsposition `pos1` und Länge `n1` gegebenen Teilstring des aktuellen Strings durch den als drittes Argument angegebenen C-String `cs`.

```
- string& string::replace(size_type pos1, size_type n1,
                        const char* cs, size_type n);
```

ersetzt den durch Anfangsposition `pos1` und Länge `n1` gegebenen Teilstring des aktuellen Strings durch den aus den ersten `n` Zeichen des als drittes Argument angegebenen C-String `cs`. (Ungeprüfter Laufzeitfehler, falls `n` größer als `strlen(cs)` ist.)

```
- string& string::replace(size_type pos1, size_type n1,
                        size_type n, char c);
```

ersetzt den durch Anfangsposition `pos1` und Länge `n1` gegebenen Teilstring des aktuellen Strings durch eine nur aus `c`-Zeichen bestehende Zeichenkette der Länge `n`.

Ersetzen eines Teilstrings der Länge 0 wird als Einfügen umgesetzt!

In folgenden fünf `replace`-Funktionen wird der zu ersetzende Teilstring durch zwei Iteratorpositionen `Anfang` und `Ende` beschrieben, d.h. `Anfang` ist eine gültige Iteratorposition und `Ende` eine nicht kleinere, gültige Iteratorposition im aktuellen String. Die durch `Anfang` (einschließlich) und `Ende` (ausschließlich) beschriebene Teilsequenz wird ersetzt (falls `Anfang` und `Ende` gleich sind, wird an der entsprechenden Position eingefügt!).

```
- string& string::replace(iterator Anfang, iterator Ende,
                        const string &s);
```

der durch `Anfang` und `Ende` gegebene Teilstring des aktuellen Strings wird durch den als Argument angegebenen C++-String `s` ersetzt.

```
- string& string::replace(iterator Anfang, iterator Ende,
                        const char *cs);
```

der durch `Anfang` und `Ende` gegebene Teilstring des aktuellen Strings wird durch den als Argument angegebenen C-String `cs` ersetzt.

```
- string& string::replace(iterator Anfang, iterator Ende,
                        const char *cs, size_type n);
```

der durch `Anfang` und `Ende` gegebene Teilstring des aktuellen Strings wird durch den aus den ersten `n` Zeichen bestehenden Teilstring des als Argument angegebenen C-Strings `cs` ersetzt. (Ungeprüfter Laufzeitfehler, falls `n` größer als `strlen(cs)` ist!)

```
- string& string::replace(iterator Anfang, iterator Ende,
                        size_type n, char c);
```

der durch `Anfang` und `Ende` gegebene Teilstring des aktuellen Strings wird durch eine aus `n` gleichen Zeichen `c` bestehende Zeichenkette ersetzt.

```
– template <class iter>
  string& string::replace(iterator Anfang, iterator Ende,
                        iter Anfang2, iter Ende2);
```

der durch **Anfang** und **Ende** gegebene Teilstring des aktuellen Strings wird durch die durch die beiden Iteratoren **Anfang2** und **Ende2** gegebene Sequenz von Zeichen ersetzt.

Funktionsergebnis aller **replace**-Funktionen ist jeweils der ersetzte String.

Durch Einfügen werden i. Allg. bisherige Iteratorpositionen ungültig!

Sollte bei einer der **replace**-Funktionen der resultierende String zu groß für das System werden, wird eine **length_error**-Ausnahme ausgelöst.

10.10 Suchen in Strings

In einem C++-String kann nach einem mit einem Muster übereinstimmenden Teilstring oder nach einem einzelnen, im Muster vorkommenden Zeichen (oder auch nicht vorkommenden Zeichen) gesucht werden.

Ergebnis ist jeweils die Position (vom Typ **size_type**) des Treffers oder des Beginns der Übereinstimmung bzw. der Wert **npos**, falls kein Treffer gefunden wurde.

10.10.1 Suchen eines Teistrings in einem String

Nach dem ersten Auftreten eines mit einem Muster übereinstimmenden Teilstrings innerhalb eines C++-Strings kann mit folgenden Funktionen gesucht werden:

```
– size_type string::find(const string &s, size_type pos = 0) const;
```

sucht nach dem ersten Auftreten des im ersten Argument angegebenen C++-String **s** im aktuellen String hinter der angegebenen Position **pos**. Defaultwert für die Position ist 0, so dass defaultmäßig vom Anfang des aktuellen Strings an gesucht wird.

```
– size_type string::find(const char *cs, size_type pos,
                        size_type n) const;
```

sucht im aktuellen String ab Position **pos** nach der ersten mit den ersten **n** Zeichen des C-Strings **cs** übereinstimmenden Teilzeichenkette.

```
– size_type string::find(const char *cs, size_type pos = 0) const;
```

sucht im aktuellen String ab Position **pos** (Defaultwert: 0) nach der ersten mit dem als erstes Argument angegebenen C-String **cs** übereinstimmenden Teilzeichenkette.

```
– size_type string::find(char c, size_type pos = 0) const;
```

sucht im aktuellen String ab Position **pos** (Defaultwert: 0) nach dem ersten Auftreten des Zeichens **c**.

Folgende Funktionen `rfind` sind analog zu den entsprechenden `find`-Funktionen, gesucht wird jeweils der letzte Treffer:

– `size_type string::rfind(const string &s, size_type pos = 0) const;`

sucht nach dem letzten Auftreten des im ersten Argument angegebenen C++-String `s` im aktuellen String vor der angegebenen Position `pos`. Defaultwert für die Position ist `npos`, so dass defaultmäßig bis zum Ende des aktuellen Strings gesucht wird.

– `size_type string::rfind(const char *cs, size_type pos, size_type n) const;`

sucht im aktuellen String bis zur Position `pos` nach der letzten mit den ersten `n` Zeichen des C-Strings `cs` übereinstimmenden Teilzeichenkette.

– `size_type string::rfind(const char *cs, size_type pos = npos) const;`

sucht im aktuellen String bis zur Position `pos` (Defaultwert: `npos`, d.h. bis zum Ende des aktuellen Strings) nach der letzten mit dem als erstes Argument angegebenen C-String `cs` übereinstimmenden Teilzeichenkette.

– `size_type string::rfind(char c, size_type pos = npos) const;`

sucht im aktuellen String bis zur Position `pos` (Defaultwert: `npos`, d.h. bis zum Ende des aktuellen Strings) nach dem letzten Auftreten des Zeichens `c`.

10.10.2 Suchen nach einzelnen Zeichen

Bei den Funktionen `find_first_of` wird im aktuellen String (ab einer gewissen Position `pos`) nach dem ersten Auftreten eines auch im Suchmuster vorkommenden Zeichen gesucht:

– `size_type string::find_first_of(const string &s, size_t pos = 0) const;`

Zeichen muss auch im C++-String `s` vorkommen.

– `size_type string::find_first_of(const char *cs, size_t pos, size_t n) const;`

Zeichen muss auch innerhalb der ersten `n` Zeichen des C-Strings `cs` vorkommen.

– `size_type string::find_first_of(const char *cs, size_t pos = 0) const;`

Zeichen muss auch im C-String `cs` vorkommen.

– `size_type string::find_first_of(char c, size_t pos = 0) const;`

gesucht wird das Zeichen `c`.

Bei den Funktionen `find_last_of` wird im aktuellen String (bis zu einer gewissen Position `pos`) nach dem letzten Auftreten eines auch im Suchmuster vorkommenden Zeichen gesucht:

```
- size_type string::find_last_of(const string &s,
                                size_t pos = npos) const;
```

Zeichen muss auch im C++-String `s` vorkommen.

```
- size_type string::find_last_of(const char *cs, size_t pos,
                                size_t n) const;
```

Zeichen muss auch innerhalb der ersten `n` Zeichen des C-Strings `cs` vorkommen.

```
- size_type string::find_last_of(const char *cs,
                                size_t pos = npos) const;
```

Zeichen muss auch im C-String `cs` vorkommen.

```
- size_type string::find_last_of(char c, size_t pos = npos) const;
```

gesucht wird das Zeichen `c`.

Bei den Funktionen `find_first_not_of` wird im aktuellen String (ab einer gewissen Position `pos`) nach dem ersten Auftreten eines Zeichens gesucht, welches nicht im Suchmuster vorkommt.

```
- size_type string::find_first_not_of(const string &s,
                                      size_t pos = 0) const;
```

Zeichen darf nicht im C++-String `s` vorkommen.

```
- size_type string::find_first_not_of(const char *cs, size_t pos,
                                      size_t n) const;
```

Zeichen darf nicht innerhalb der ersten `n` Zeichen des C-Strings `cs` vorkommen.

```
- size_type string::find_first_not_of(const char *cs,
                                      size_t pos = 0) const;
```

Zeichen darf nicht im C-String `cs` vorkommen.

```
- size_type string::find_first_not_of(char c, size_t pos = 0) const;
```

gesucht wird ein von `c` verschiedenes Zeichen.

Bei den Funktionen `find_last_not_of` wird im aktuellen String (bis zu einer gewissen Position `pos`) nach dem letzten Auftreten eines Zeichens gesucht, welches nicht im Suchmuster vorkommt.

```
- size_type string::find_last_not_of(const string &s,
                                      size_t pos = npos) const;
```

Zeichen darf nicht im C++-String `s` vorkommen.


```
- size_type string::find_last_not_of(const char *cs, size_t pos,
                                     size_t n) const;
```

Zeichen darf nicht innerhalb der ersten `n` Zeichen des C-Strings `cs` vorkommen.

```
- size_type string::find_last_not_of(const char *cs,
                                     size_t pos = npos) const;
```

Zeichen darf nicht im C-String `cs` vorkommen.

```
- size_type string::find_last_not_of(char c,
                                     size_t pos = npos) const;
```

gesucht wird ein von `c` verschiedenes Zeichen.

10.11 Ein- und Ausgabe von C++-Strings

Zur Ausgabe eines C++-Strings sieht der Standard folgende globale Funktion vor:

```
ostream& operator<<( ostream &strm, const string &s);
```

welche wie üblich aufzurufen ist:

```
string s(...);
...
cout << s << endl; // String, anschliessend Zeilenvorschub ausgeben
...
```

Zur Eingabe ist die entsprechende Operatorfunktion verfügbar:

```
istream& operator>>( istream & strm, string &s);
```

Aufruf etwa:

```
string s;
...
cin >> s; // String einlesen
...
```

Beim Einlesen mit `>>` wird (wie üblich) zunächst Zwischenraum (Leerzeichen, Tabulatoren, ...) überlesen, alle folgenden Nichtzwischenraumzeichen werden der Reihe nach im String gespeichert, das Lesen endet vor dem nächsten Zwischenraumzeichen. Zum Einlesen eines Strings gibt es zusätzlich die beiden folgenden, globalen Funktionen:

```
istream& getline(istream & strm, string &s, char eol);
istream& getline(istream & strm, string &s);
```

Die erste der beiden Funktionen liest bis einschließlich des nächsten Auftretens des *End of Line-Zeichens* `eol` und speichert das Gelesene bis auf das abschließende `eol` im String `s` ab (auch anfänglichen Zwischenraum!).

Bei der zweiten Funktion übernimmt das Zeilenvorschubzeichen `'\n'` die Rolle des `eol`.

10.12 Vertauschen von Strings

Da C++-Strings dynamische Komponenten haben, kann das Vertauschen des Inhalts zweier C++-Strings sehr effizient implementiert werden (es brauchen nur Verweise vertauscht zu werden, das Umkopieren ganzer Zeichenketten entfällt!).

Hierzu gibt es die Member-Funktion:

```
void string::swap(string &);
```

sowie die globale Funktion:

```
void swap(string &a, string &b);
```

Aufrufe:

```
string s1(...);
string s2(...);
...
s1.swap(s2);    // Vertausche Inhalt von s1 und s2, Member-Funktion
...
swap(s1,s2);    // Vertausche Inhalt von s1 und s2, globale Funktion
...
```

10.13 String-Streams

Im Zusammenhang mit Stream-Ein-Ausgabe haben wir bereits die String-Stream-Klassen

`istringstream` (Lesen von Strings, von `istream` abgeleitet), `ostringstream` (Schreiben auf Strings, von `ostream` abgeleitet) und `stringstream` (Lesen von und Schreiben auf Strings, von `iostream` abgeleitet) kennengelernt.

Einem String-Stream liegt ein C++-String zugrunde, zusätzlich sind die entsprechenden Lese- bzw. Schreiboperationen bzw. beide verfügbar.

Zur Verwendung dieser String-Stream-Klassen muss die Headerdatei `<sstream>` includet werden. (Auf dem GCC-Compiler ist der Name dieser Headerdatei noch `<strstream>` und die Klassen heißen `istrstream`, `ostrstream` und `strstream`, die Funktionalität entspricht aber weitgehend dem des im Standard vorgesehenen.)

Jeder String-Stream kann bei seiner Erzeugung mit einem C++-String initialisiert werden, wobei ein `istringstream` standardmäßig im Modus `ios_base::in` (Lesen) geöffnet wird, ein `ostringstream` standardmäßig im Modus `ios::base::out` (Schreiben) und ein `stringstream` standardmäßig im Modus `ios_base::in | ios_base::out` (Lesen und Schreiben). Konstruktoren für String-Streams:

```
istringstream::istringstream(ios_base::openmode modus = ios_base::in);
istringstream::istringstream(const string &s,
                             ios_base::openmode modus = ios_base::in);

ostringstream::ostringstream(ios_base::openmode modus = ios_base::out);
ostringstream::ostringstream(const string &s,
                             ios_base::openmode modus = ios_base::out);
```

```
stringstream::stringstream(ios_base::openmode
                           modus = ios_base::in | ios_base::out);
stringstream::stringstream(const string &s, ios_base::openmode
                           modus = ios_base::in | ios_base::out);
```

(Diese Konstruktoren sind im Klassenrumpf der String-Stream-Klassen als **explicit** vereinbart, so dass diese nicht "versehentlich" zu Typumwandlung herangezogen werden!)

Beim Schreiben wird immer an das bisherige Stringende angefügt (hierbei wird der String ggf. dynamisch vergrößert), beim Lesen wird zunächst vom Stringanfang gelesen und der nächste Lesevorgang beginnt dort, wo der vorherige geendet hat. Beim Stringende endet jeder Lesevorgang (entspricht dem EOF bei Datei-Strömen).

```
// Modus: out; C-String "Ergebnis: " wird hierbei in einen
// C++-String umgewandelt, mit dem o_stst initialisiert wird!
ostream o_stst("Ergebnis: ");
```

```
// Modus: in ; Umwandlung wie oben
istream i_stst(" 1 2 3 4");
```

```
int i,j;
```

```
i_stst >> i;    // lese erstes int von i_stst
i_stst >> j;    // lese zweites int von i_stst
```

```
o_stst << i*j;  // schreibe Produkt auf o_stst
...
```

Neben den Ein-Ausgabe-Funktionen zu String-Streams sind die folgenden Funktionen die wichtigsten:

```
- string istream::str() const;
  string ostream::str() const;
  string stringstream::str() const;
```

liefert eine Kopie des derzeitigen Inhalts des String-Streams als C++-String.

```
- void istream::str(const string &s);
  void ostream::str(const string &s);
  void stringstream::str(const string &s);
```

überschreibt den bisherigen Inhalt des String-Streams mit dem als Argument angegebenen C++-String.

Aufgrund der mittels des Konstruktors `string::string(const char *)`; definierten Typumwandlung von C-Strings nach C++-Strings kann in dieser Funktion auch ein C-String als Argument angegeben werden!

Mit diesen Funktionen kann man zwischenzeitlich bei Bedarf einen String-Stream in einen gewöhnliche C++-String umwandeln und umgekehrt. Somit hat man etwa die Möglichkeit, Ausgabe formatiert in eine `ostream` zu schreiben und diese (über einen String) in einen `istream` zu kopieren und von diesem anschließend formatiert zu lesen:

```
ostream o_stst;
istream i_stst;

// formatierte Ausgabe auf ostream o_stst:
o_stst << setw(...) << setprecision(...) << ... ;
o_stst << ... << ... << ... ;
...
// i_stst mit dem aus o_stst erhaltenen String initialisieren!
i_stst.str( o_stst.str() );
...
// vom istream i_stst lesen:
i_stst >> ...;
...
```

Kapitel 11

Die Standard–Template–Library (STL)

In der STL sind durch Templates eine Reihe von Datentypen und Funktionen vereinbart, welche für unterschiedliche Anwendungen konkretisiert werden können.

11.1 Universelle Hilfsmittel der Standardbibliothek

11.1.1 Vergleichsoperatoren

Sind zu einem Datentyp `T` die Vergleichsoperatoren `<` und `==` definiert, so sind im Standard (genauer: in der Headerdatei `<utility>`) durch folgende Templates:

```
template <class T>
bool operator!=(const T& x, const T& y)
{ return ! ( x == y); }
```

```
template <class T>
bool operator>(const T& x, const T& y)
{ return  ( y < x); }
```

```
template <class T>
bool operator<=(const T& x, const T& y)
{ return ! ( y < x); }
```

```
template <class T>
bool operator>=(const T& x, const T& y)
{ return ! ( x < y); }
```

auch die übrigen Vergleichsoperatoren für diesen Typen `T` definiert!

Natürlich kann man für diese, für einen Typen `T` dann standardmäßig vorhandenen Vergleichsoperatoren eine eigene alternative Implementierung anbieten.

11.1.2 Die Template-Klasse `pair<>`

Bei einigen Anwendungen, z.B. auch in der Standardbibliothek bei den Containerklassen `map` und `multimap` (siehe etwa Abschnitt 11.3.7), werden Wertepaare verwendet. In der Headerdatei `<utility>` wird ein universeller Datentyp für Paare von Werten, deren erste Komponente von einem gewissen Typ `T1` und deren zweite Komponente von einem möglicherweise anderen Typen `T2` sind, bereitgestellt:

```
template <class T1, class T2>
struct pair {

    // Typnamen der beteiligten Typen
    typedef T1 first_type;
    typedef T2 second_type;

    // Komponenten
    T1 first;
    T2 second;

    // Standard-Konstruktor
    pair() {}

    // Konstruktor mit Werten zur Initialisierung
    pair(const T1 &a, const T2 &b): first(a), second(b) {}

    // Copy-Konstruktor
    template <class U, class V>
    pair( const pair<U,V> &p): first(p.first), second(p.second) {}
};
```

Ein Objekt `a` vom Typ `pair<T1, T2>` hat somit zwei (öffentlich verfügbare) Komponenten, nämlich `a.first` vom Typ `T1` und `a.second` vom Typ `T2`.

Der Standard-Konstruktor erzeugt ein Paar, dessen Komponenten mit deren Standard-Konstruktor erzeugt werden.

Der Copy-Konstruktor ist selbst als Template realisiert, so dass eine automatische Typumwandlung bei der Initialisierung der einzelnen Komponenten möglich ist.

Neben diesen Element-Funktionen gibt es folgende Vergleichsfunktionen:

```
// Test auf Wertgleichheit
template <class T1, class T2>
bool operator== (const pair<T1, T2> &x, const pair<T1, T2> &y)
{ return x.first == y.first && x.second == y.second; }

// Test auf "kleiner als"
template <class T1, class T2>
bool operator< (const pair<T1, T2> &x, const pair<T1, T2> &y)
{ return (x.first < y.first) ||
        ( !(y.first < x.first) && x.second < y.second); }
```

sowie folgende Funktion zur Erzeugung eines Paares, ohne Typen angeben zu müssen:

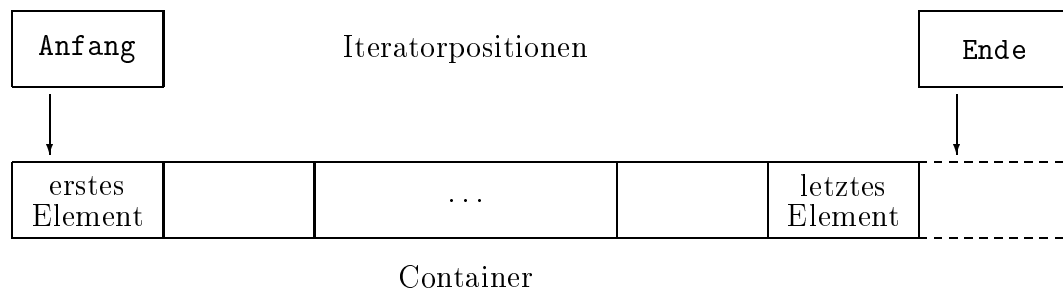
```
// Erzeugen eines Paares, ohne explizite Typangabe
template <class T1, class T2>
pair<T1, T2> make_pair( const T1 &x, const T2 &y)
{ return pair<T1,T2> (x,y); }
```

11.2 Iteratoren

Iteratoren kommen immer ins Spiel, wenn etwas der Reihe nach mit irgendwelchen Elementen (aus irgendeinem endlichen “Reservoir“ von Elementen, etwa aus einen Standardcontainer) durchgeführt werden soll.

Sie sind zur Formulierung von Algorithmen hilfreich, bei denen es nur darauf ankommt, der Reihe nach auf die Elemente zugreifen zu können und die interne Element-Abspeicherung nicht interessiert. (Die vom Standard vorgesehenen Algorithmen werden in einem späteren Kapitel behandelt!)

Üblicherweise ist die über einen Iterator mit Positionen **Anfang** bis **Ende** verfügbare Menge von Daten eine *Sequenz*, d.h. eine “halboffene“ Menge von Elementen, zu der das Element mit der Position **Anfang** hinzugehört, es aber kein Element mit der Position **Ende** gibt:



Eine solche Sequenz ist genau dann leer, wenn **Anfang** gleich **Ende** ist. (In gewisser Hinsicht ist das Iterator-Konzept die Verallgemeinerung des Konzeptes: *Zeiger auf Felder*!)

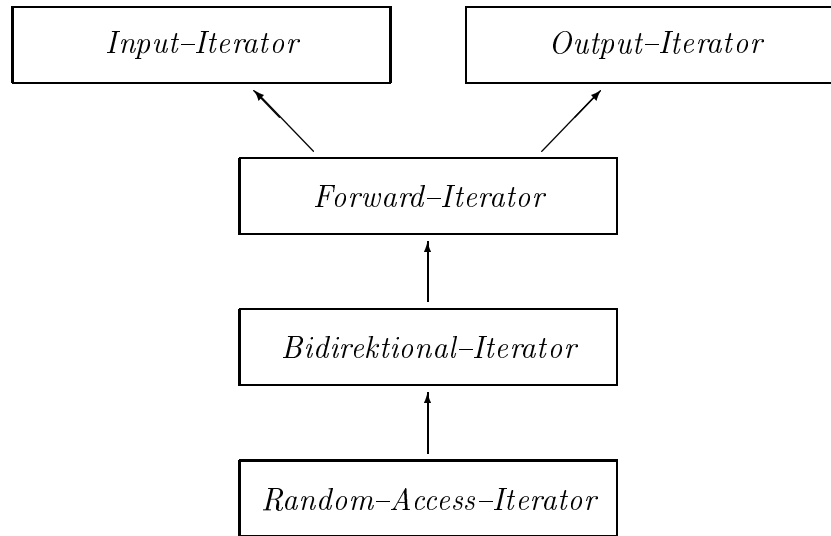
Neben Iteratoren bei Containern wird das Iterator-Konzept in C++ auch für Datenströme so verallgemeinert, dass sich auch hier überraschende Anwendungsmöglichkeiten eröffnen! (Auch ein *istream* kann als endliches “Reservoir“ von Daten aufgefasst werden, auf welche man der Reihe nach — aber nur lesend — zugreifen kann. Ein *ostream* kann ebenfalls als ein “Reservoir“ von Daten aufgefasst werden, in welches man der Reihe nach Daten “hineinschreiben“ kann.)

Das universelle Iterator-Konzept wird in C++ in der Headerdatei `<iterator>` zur Verfügung gestellt — diese Headerdatei wird jedoch in der Regel von den Headerdateien der Containerklassen automatisch implizit eingebunden.

Das Iterator-Konzept in C++ stellt zunächst keine konkreten oder abstrakten Klassen zur Verfügung, sondern nur eine Funktionalität als “Abstraktion“, d.h. alles, was die Funktionalität eines Iterators hat, ist ein Iterator. Z.B. hat ein Zeiger die Funktionalität eines *Random-Access-Iterators* — somit ist ein Zeiger ein *Random-Access-Iterator*!

11.2.1 Iterator-Kategorien

Entsprechend ihrer Funktionalität gibt es unterschiedliche “Kategorien“ von Iteratoren, welche in folgendem Bild veranschaulicht werden sollen:



Ein Pfeil in diesem Bild bedeutet: die Iterator-Kategorie, von dem der Pfeil ausgeht, umfasst in seiner Funktionalität diejenige der Iteratorkategorie, an der der Pfeil endet.

Es handelt sich hierbei um die Funktionalität — dieses Schaubild stellt **keine** Klassenhierarchie dar!

Input-Iteratoren Input-Iteratoren stellen das universelle Konzept zum lesenden Zugriff auf die Elemente eines “Reservoirs“ dar, also Werte aus dem Reservoir herausholen — keine Werte in “Reservoir“ hineinschreiben. Hierbei ist **nicht** gefordert, dass bei mehrmaligem Durchlauf desselben “Reservoirs“ jedesmal die gleichen Elemente geliefert werden (das Reservoir könnte beispielsweise der Datenstrom `cin` sein!). Neben Copy-Konstruktor muss ein Input-Iterator über folgende Operationen verfügen:

<code>iter1 == iter2</code>	Test auf Gleichheit (gleiches “Reservoir“ und gleiche Position).
<code>iter1 != iter2</code>	Test auf Ungleichheit.
<code>*iter</code>	Lesezugriff auf das aktuelle Element.
<code>iter->komp</code>	Zugriff auf eine Komponente des aktuellen Elementes.
<code>++iter</code>	Weitersetzen des Iterators, Ergebnis ist neue Position.
<code>iter++</code>	Weitersetzen des Iterators, Ergebnis ist alte Position.

Im Allgemeinen ist die Präfix-Inkrementierung `++iter` effizienter als die Postfix-Inkrementierung `iter++` eines Iterators, da das Ergebnis die aktuelle Position des Iterators ist.

Output-Iteratoren Mit einem Output-Iterator kann man Elemente eines “Reservoir“ ändern bzw. neue Elemente in ein “Reservoir“ hineinschreiben. Elemente aus dem “Reservoir“ herausholen kann man mit einem Output-Iterator nicht!

Neben Copy-Konstruktor muss ein Output-Iterator über folgende Funktionalität verfügen:

<code>*iter = wert</code>	Schreibzugriff auf das aktuelle Element.
<code>++iter</code>	Weitersetzen des Iterators, Ergebnis ist neue Position.
<code>iter++</code>	Weitersetzen des Iterators, Ergebnis ist alte Position.

Forward-Iteratoren Forward-Iteratoren haben die Funktionalität eines Input-Iterators **und** eines Output-Iterators. Zusätzlich muss zu einem Forward-Iterator neben dem Copy-Konstruktor ein Standard-Konstruktor und ein Zuweisungsoperator existieren. Neben den Konstruktoren ergeben sich für Forward-Iteratoren zusammengefasst folgende Operationen:

<code>iter1 == iter2</code>	Test auf Gleichheit (gleiches "Reservoir" und gleiche Position).
<code>iter1 != iter2</code>	Test auf Ungleichheit.
<code>*iter</code>	Lese-/Schreibzugriff auf das aktuelle Element.
<code>iter->komp</code>	Zugriff auf eine Komponente des aktuellen Elementes.
<code>++iter</code>	Weitersetzen des Iterators, Ergebnis ist neue Position.
<code>iter++</code>	Weitersetzen des Iterators, Ergebnis ist alte Position.
<code>iter1 = iter2</code>	Zuweisungsoperator.

Bidirektional-Iteratoren Ein bidirektionaler Iterator ist ein Forward-Iterator, der auch "rückwärts" über die Elemente der Sequenz laufen kann. Hierzu ist zusätzlich der Dekrement-Operator `--` (in Präfix- und in Postfix-Form, Präfix ist effektiver) definiert, der die Position des Iterators um eins zurücksetzt:

<code>--iter</code>	Zurücksetzen des Iterators, Ergebnis ist neue Position.
<code>iter--</code>	Zurücksetzen des Iterators, Ergebnis ist alte Position.

Die Iteratoren aller Standardcontainer sind zumindest bidirektional.

Random-Access-Iteratoren Random-Access-Iteratoren sind bidirektionale Iteratoren, welche zusätzlich "wahlfreien" Zugriff auf die Elemente einer Sequenz bieten. Positionen eines Random-Access-Iterators können mit beliebigen Vergleichsoperatoren verglichen werden; Positionen kann man durch Addition bzw. Subtraktion von ganzzahligen Werten (nicht nur um eine Position) erhöhen bzw. erniedrigen und man kann zwei Iteratorpositionen einer Sequenz voneinander abziehen und erhält die Anzahl der Elemente der zwischen den Positionen liegenden Sequenz.

Die über die Operatoren für Forward- und Bidirektionale-Iteratoren hinausgehenden Operatoren für Random-Access-Iteratoren sind in folgender Tabelle aufgelistet:

<code>iter[n]</code>	Zugriff auf das <code>n</code> -te Element hinter bzw. vor der augenblicklichen Iteratorposition. (Hinter, falls <code>n</code> positiv — vor, falls <code>n</code> negativ ist!)
<code>iter += n</code>	Iterator um <code>n</code> Positionen weitersetzen (bzw. zurück, falls <code>n</code> negativ ist).
<code>iter -= n</code>	Iterator um <code>n</code> Positionen zurücksetzen (bzw. vor, falls <code>n</code> positiv ist).

<code>iter + n</code> <code>n + iter</code>	Iterator für das <code>n</code> -te folgende (bzw. vorherige, falls <code>n</code> negativ ist) Element liefern.
<code>iter - n</code>	Iterator für das <code>n</code> -te vorhergehende (bzw. folgende, falls <code>n</code> negativ ist) Element liefern.
<code>iter1 - iter2</code>	Abstand der beiden Iteratoren liefern.
<code>iter1 < iter2</code> <code>iter1 > iter2</code> <code>iter1 <= iter2</code> <code>iter1 >= iter2</code>	Vergleiche.

Von den Standardcontainern haben `vector<T>` und `deque<T>` Random-Access-Iteratoren, die übrigen verfügen über Bidirektional-Iteratoren.

11.2.2 Iterator-Traits

Zu jedem Iterator kann sein Typ (Kategorie) sowie weitere seiner Eigenschaften in Erfahrung gebracht werden. Hierzu gibt es zu jedem Iterator `iter` den Datentyp `iterator_traits<iter>`, in dem nur zum Iterator `iter` "passende" Typen vereinbart werden:

```
struct iterator_traits<iter>
{
    typedef ... iterator_category;
    typedef ... value_type;
    typedef ... difference_type;
    typedef ... pointer;
    typedef ... reference;
};
```

(hier sind ... jeweils zum Iterator "passende" Typen!)

Der Typ `iterator_category` kann einer der folgenden, vom Standard vorgegebenen Typen sein, welche die vom Standard vorgesehenen Iterator-Kategorien widerspiegeln:

```
struct output_iterator_tag
{};          // leerer Typ
struct input_iterator_tag
{};          // leerer Typ
struct forward_iterator_tag: public input_iterator_tag
{};          // von input_iterator_tag abgeleitet
struct bidirectional_iterator_tag: public forward_iterator_tag
{};          // von forward_iterator_tag abgeleitet
struct random_access_iterator_tag: public bidirectional_iterator_tag
{};          // von bidirectional_iterator_tag abgeleitet
```

Es handelt sich hierbei um eine Hierarchie von formalen (leeren) Typen.

Der Typ `value_type` beinhaltet den Typ der Elemente, auf welchen der Iterator verweist.

Der Typ `difference_type` ist der Typ der Differenz von Iteratorpositionen.

Der Typ `pointer` ist der Ergebnistyp des `->`-Operators im Zusammenhang mit dem Iterator, also der Typ von `iter->`.

Der Typ `reference` ist der Ergebnistyp des `*`-Operators im Zusammenhang mit dem Iterator, also der Typ von `*iter`.

Die in `iterator_traits<iter>` definierten Hilfstypen spiegeln die Eigenschaften des Iterators `iter` wider.

Der Grund, warum diese Eigenschaften des Iterators `iter` in `iterator_traits<iter>` abgelegt ist und nicht etwa im Iteratortyp `iter` selber ist der, dass auch ein Standardtyp, etwa ein Zeiger, als Iterator verwendet werden kann und in diesem Typ (etwa Zeiger) die entsprechende Information nicht untergebracht werden kann.

Für beliebige Zeiger `T*` sind die zugehörigen Traits (im Standard bereits) mittels folgenden Templates definiert:

```
template <class T>
struct iterator_traits<T*> {
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef T& reference;
};
```

(hierbei ist `ptrdiff_t` der systemabhängige, bereits in C definierte Datentyp für die Differenz von Zeigern!)

Für einen anderen, selbstdefinierten Iteratortyp `Iterator` geht der Standard davon aus, dass dieser von der Template-Klasse

```
template <class Category, class T, class Distance = ptrdiff_t,
         class Pointer = T*, class Reference = T*>
struct iterator {
    typedef Category iterator_category;
    typedef T value_type;
    typedef Distance difference_type;
    typedef Pointer pointer;
    typedef Reference reference;
};
```

abgeleitet ist, in der im Iterator selbst die entsprechenden Eigenschaften (Typen) definiert sind.

Zu einem solchen Iteratortyp `Iterator` werden standardmäßig (wieder über ein Template) folgende Traits zur Verfügung gestellt:

```
template <class Iterator>
struct iterator_traits {
    typedef typename Iterator::iterator_category iterator_category;
    typedef typename Iterator::value_type value_type;
```

```

typedef typename Iterator::difference_type    difference_type;
typedef typename Iterator::pointer            pointer;
typedef typename Iterator::reference          reference;
};

```

(Das Schlüsselwort `typename` muss hier jeweils stehen, damit der Compiler erkennt, dass es sich beim folgenden Namen — etwa `Iterator_category` — nicht um eine Datenkomponente der Klasse `Iterator`, sondern um einen in dieser Klasse definierten Typ handelt.

Definiert man einen eigenen Iterator:

```

class MyIterator: public iterator<random_access_iterator_tag,T>
{ ... };

```

und ist nicht mit den hierzu vom Standard erzeugten Traits zufrieden, so kann man die zugehörigen Traits selbst definieren:

```

struct iterator_traits<MyIterator>
{
    typedef random_access_iterator_tag;
    typedef ... value_type;           // passende Typen definieren
    ...
};

```

11.2.3 Funktionen, welche von der Iterator-Kategorie abhängen

Mittels der Iterator-Traits und der Typprüfung durch den Compiler können Funktionen geschrieben werden, welche für unterschiedliche Iterator-Kategorien unterschiedlich ablaufen, etwa eine Funktion `plusvier`, um die Position eines Iterators (effizient) um 4 Positionen zu erhöhen.

Bei einem Random-Access-Iterator `iter` könnte das durch `iter += 4` erfolgen, als Funktion (mit zweitem Argument vom Typ `random_access_iterator_tag`):

```

// Version fuer Random-Access-Iteratoren
template <class RaIterator>
void _plusvier (RaIterator &iter, random_access_iterator_tag)
{
    iter += 4;
}

```

Bei einem Forward-Iterator `iter` müsste dieser viermal (mit `++`) inkrementiert werden, als Funktion (mit zweitem Argument vom Typ `forward_iterator_tag`):

```
// Version fuer Forward-Iteratoren
template <class FoIterator>
void _plusvier(FoIterator &iter, forward_iterator_tag)
{
    for ( int i = 0; i < 4; ++i)
        iter++;
}
```

Folgende Funktion kann nun für Forward-, Bidirektionale- und Random-Access-Operatoren aufgerufen werden:

```
template <class Iter>
void plusvier(Iter &iter)
{
    _plusvier( iter, iterator_traits<iter>::iterator_category() );
}
```

Anhand der zum Argument `iter` gehörenden `iterator_traits<iter>` wird die Kategorie des Iterators ermittelt und als (mittels Standard-Konstruktor erzeugtes, temporäres) zweites Argument in den Aufruf der Funktion `_plusvier` gestellt.

Anhand der Typprüfung des Compilers wird dann die zur Iterator-Kategorie gehörende Version der Funktion `_plusvier` aufgerufen, d.h. für Random-Access-Iteratoren die Version für Random-Access-Iteratoren und für Bidirektionale- und Forward-Iteratoren die für Forward-Iteratoren (der Typ `bidirectional_iterator_tag` ist von `forward_iterator_tag` abgeleitet!).

Für sonstige Iteratoren (Input- oder Output-Iteratoren) ist der Aufruf von `plusvier` fehlerhaft, weil es keine Funktionen

```
void _plusvier(Iteratortyp,input_iterator_tag);    bzw.
void _plusvier(Iteratortyp,output_iterator_tag);
gibt!
```

11.2.4 Hilfsfunktionen für Iteratoren

Von diesem Typ: *Iterator-Kategorie-abhängige Funktionen* sind in der STL folgende bereits definiert:

```
- template <class InputIterator, class Distance>
  // Distance ist Distanztyp des Input-Iterators
  void advance(InputIterator &iter, Dist n);
```

welche die Position des Iterators um `n` Positionen weitersetzt.

Bei Bidirektionalen- oder Random-Access-Iteratoren darf `n` auch negativ sein — in diesem Fall wird die Iteratorposition zurückgesetzt.

Bei Random-Access-Iteratoren wird hierbei der Wert von `n` auf den Iterator addiert, bei sonstigen Iteratoren wird `n` mal mit `++` inkrementiert bzw. `-n` mal mit `--` dekrementiert, falls `n` negativ ist (nur bei Bidirektionalen-Iterator möglich)!

```
– template <class InputIterator>
  typename iterator_traits<InputIterator>::difference_type
  distance(InputIterator first, InputIterator last);
```

welche die Differenz der Positionen `first` und `last` eines Iterators ermittelt. Der Typ des Ergebnisses ist hierbei selbst wieder von den Iterator-Traits abhängig, nämlich der zugehörige `difference_type`.

Bei Random-Access-Iteratoren wird hier einfach `last - first` berechnet. Bei sonstigen Iteratoren wird `first` (ist lokale Variable) solange mittels `++` erhöht, bis eine Übereinstimmung mit `last` vorliegt, und die Anzahl der Erhöhungen zurückgegeben.

Die Funktion

```
template <class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 iter1, ForwardIterator2 iter2);
```

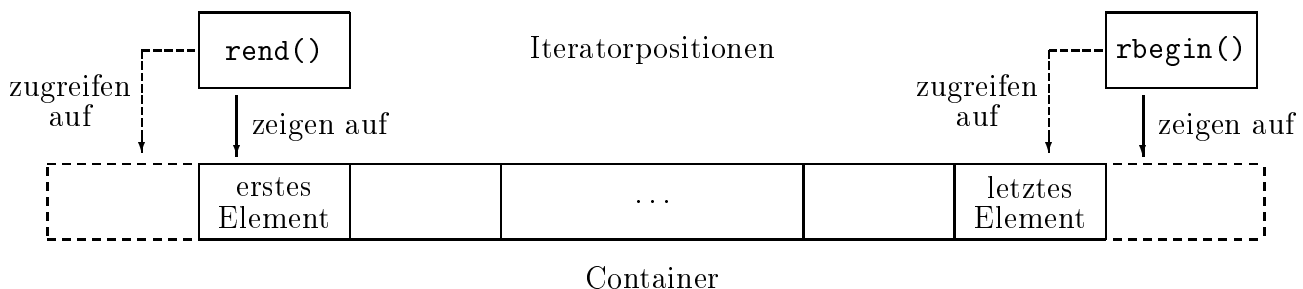
vertauscht den Wert, auf den der Iterator `iter1` zeigt, mit dem Wert, auf den `iter2` zeigt. Die beiden Iteratoren können verschieden sein, können sogar unterschiedlichen Typ haben — nur der `value_type` beider Iteratoren, also der Typ der Objekte, auf die die Iteratoren zeigen, müssen gleich sein und für die entsprechenden Elemente muss der Zuweisungsoperator definiert und aufrufbar sein!

11.2.5 Iterator-Adapter

Wie bei den Standardcontainern mittels Adapter neue Schnittstellen zu den Containern definiert wurden (siehe Abschnitt 11.4), so werden zu den vorgestellten Iteratoren mittels “Adapter” spezielle Schnittstellen zur Verfügung gestellt.

Reverse-Iteratoren Der zu jeder Containerklasse definierte `reverse_iterator`-Typ ist ein “Adapter” für den gewöhnlichen Iteratortyp `iterator` derselben Containerklasse, d.h. einem `reverse_iterator` liegt ein “normaler” `iterator` zugrunde und das Erhöhen des `reverse_iterator`’s (mit `++`) wird in ein Erniedrigen des zugrundeliegenden “normalen” `iterator`’s (mit `--`) umgesetzt und umgekehrt. (Man beachte: alle Containerklassen verfügen zumindest über bidirektionale Iteratoren!)

Definiert man einen `reverse_iterator`, so wird implizit ein gewöhnlicher Iterator erzeugt, der auf das erste Element bis eins hinter dem letzten Element eines zugrundeliegenden Containers zeigen kann. Greift man auf einen gewöhnlichen Iterator zu, so erhält man das Element, auf welches er zeigt. Greift man auf einen `reverse_iterator` zu, so erhält man jedoch das Element **vor** dem, auf welches der (dem `reverse_iterator` zugrundeliegende) gewöhnliche Iterator zeigt:



Ein `reverse_iterator` verfügt über alle Operationen, über die der zugrundeliegende “normale” Iterator verfügt.

Wendet man die Funktion:

```
iterator base(reverse_iterator);
```

auf irgendeinen `reverse_iterator` an, so erhält man (als Wert) die Position des dem `reverse_iterator` zugrundeliegenden “normalen” Iterators! Zu beachten ist: der Zugriff auf den `reverse_iterator` und der Zugriff auf diesen hieraus mittels `base` in einen “normalen” zurückverwandten Iterator liefern verschiedene, um eins verschobene Elemente der Sequenz!

Insert-Iteratoren Ist `iter` ein Iterator mit Elementtyp `T` und `elem` ein Wert mit Typ `T`, so wird bei der Zuweisung:

```
*iter = elem;
```

dem Element, auf welches der Iterator `iter` zeigt, ein neuer Wert zugewiesen. (Insbesondere ist hierbei der Fehler möglich, dass der Iterator nicht auf eine gültige Position einer Sequenz — etwa hinter das Ende der Sequenz — zeigt und diese Zuweisung somit nicht zu einem wohldefinierten Resultat führt!)

Ein Insert-Iterator `i_iter` (auch *Insertter* genannt) ist ein so gestalteter Iterator-Adapter (d.h. diesem liegt wiederum ein gewöhnlicher Iterator zugrunde), dass bei einer Zuweisung an `*i_iter` (oder an `i_iter` selbst):

```
*i_iter = elem;
i_iter = elem;
```

eine Kopie des Elementes `elem` an die Stelle **eingefügt** wird, auf die der dem Insert-Iterator zugrundeliegende gewöhnliche Iterator verweist.

Die Schnittstelle zu einem Insert-Iterator sieht (neben Konstruktoren und Destruktor) wie folgt aus:

<code>i_iter = elem</code>	fügt Kopie von <code>elem</code> an der Iteratorposition ein.
<code>*i_iter</code> <code>++i_iter</code> <code>i_iter++</code>	liefert den Iterator <code>i_iter</code> selbst, sonst geschieht nichts.

Bei der Erzeugung eines Insert-Iterators muss der Container und die (gewöhnliche) Iteratorposition angegeben werden, an welcher Stelle der Insert-Iterator in diesem Container Elemente einfügen kann.

Der Standard unterscheidet folgende drei Arten von Insert-Iteratoren:

– **Back-Insertter**

Fügen Elemente am Ende des Containers ein. Diese rufen implizit die Funktion `push_back(T&)` der zugrundeliegenden Containerklasse auf.

Erzeugt werden kann ein Back-Insertter durch

- den Konstruktor:

```
back_insert_iterator<containertyp> name(container);
```

wobei `container` ein Container des angegebenen Types `containertyp` ist.

- die (Template-)Funktion:

```
back_inserter(container);
```

hierbei wird ein unbenannter Back-Insertor erzeugt, dessen Typ sich nach dem Typ des angegebenen Argumentes richtet. Bei gewissen Funktionen kann man als Argument einen derartigen unbenannten Insertor angeben:

```
fkt(..., back_inserter(container) );.
```

– Front-Insertor

Fügen Elemente am Anfang des Containers ein. Diese rufen implizit die Funktion `push_front(T&)` der zugrundeliegenden Containerklasse auf. (Für die zugrundeliegende Containerklasse muss somit diese Funktion definiert sein!)

Erzeugt werden kann ein Front-Insertor durch

- den Konstruktor:

```
front_insert_iterator<containertyp> name(container);
```

wobei `container` ein Container des angegebenen Types `containertyp` ist.

- die (Template-)Funktion:

```
front_inserter(container);
```

hierbei wird ein unbenannter Front-Insertor erzeugt, dessen Typ sich nach dem Typ des angegebenen Argumentes richtet. Bei gewissen Funktionen kann man als Argument einen derartigen unbenannten Insertor angeben:

```
fkt(..., front_inserter(container) );.
```

– Insertor

Fügen Elemente an einer beliebigen, bei Erzeugung des Insertors angegebenen Position in den Container ein. Erzeugt werden kann ein Insertor durch

- den Konstruktor:

```
insert_iterator<containertyp> name(container, pos);
```

wobei `container` ein Container des angegebenen Types `containertyp` und `pos` eine gewöhnliche Iteratorposition in diesem Container ist.

- die (Template-)Funktion:

```
inserter(container, pos);
```

hierbei wird ein unbenannter Front-Insertor erzeugt, dessen Typ sich nach dem Typ des angegebenen Container-Argumentes richtet. Auch hier ist `pos` eine Iteratorposition in dem als erstes Argument angegebenen Container. (Bei gewissen Funktionen kann man als Argument einen derartigen unbenannten Insertor angeben: `fkt(..., inserter(container, pos));`.)

Es wird an der entsprechenden Position `pos` des Containers eingefügt. Hierzu wird implizit die Funktion `insert(pos, elem)` des Containers aufgerufen — derartige Insert-Iteratoren sind somit nur für solche Containerklassen möglich, welche diese Funktion `insert(pos, elem)` unterstützen!

11.2.6 Stream-Iteratoren

Stream-Iteratoren dienen dazu, Datenströme wie Container zu behandeln, in welche man der Reihe nach Elemente einfügen kann (Ostream-Iterator, also ein Insert-Iterator für einen Ostream) bzw. von denen man der Reihe nach Elemente “heraus-holen“ kann (Istream-Iterator, das Gegenstück zu einem Insert-Iterator in Zusammenhang mit einem Istream).

Stream-Iteratoren können ebenfalls als Adapter (spezielle Schnittstelle) für gewöhnliche Iteratoren aufgefasst werden.

Viele Algorithmen und Funktionen verarbeiten gewisse Elemente, welche sie aus einem Container erhalten und schreiben die bearbeiteten Elemente in einen anderen Container — das Prinzip ist also: Elemente aus einem Container holen und bearbeitet in einen anderen Container schreiben.

Zweck der Ostream- und Istream-Iteratoren ist es, derartige Algorithmen und Funktionen auch für Streams anwendbar zu machen, also ggf. die zu bearbeitenden Elemente nicht aus einem Container, sondern von einem Istream zu holen und die Ergebnisse ggf. nicht in einen anderen Container, sondern in einen Ostream zu schreiben.

Diese Algorithmen sind dann nicht mit Iteratoren für gewöhnliche Container, sondern mit Ostream- und/oder Istream-Iteratoren als Argument aufzurufen!

Ostream-Iteratoren Ostream-Iteratoren arbeiten wie Insert-Iteratoren, bei jeder Zuweisung an den Iterator (oder das, worauf der Iterator zeigt) wird das Zugewiesene auf dem Ostream ausgegeben.

Bei der Erzeugung des Ostream-Iterators muss der Ostream angegeben werden, auf dem die Elemente ausgegeben werden sollen:

- `ostream_iterator<T> o_iter(ostream);`
`o_iter` ist ein Ostream-Iterator zur Ausgabe von Elementen vom Typ `T` auf den als Argument angegebenen Ostream.
- `ostream_iterator<T> o_iter(ostream, string);`
`o_iter` ist ein Ostream-Iterator zur Ausgabe von Elementen vom Typ `T` auf den als Argument angegebenen Ostream, wobei nach jeder Ausgabe eines Elementes die angegebene Zeichenkette `string` als “Trenner“ ausgegeben wird.

Die Operatoren für einen Ostream-Iterator sind dieselben wie bei einem Insert-Iterator:

<code>o_iter = elem</code>	gibt Wert von <code>elem</code> auf dem Ostream aus (evtl. gefolgt von dem “Trenner“). (Hierzu wird implizit der <code><<</code> -Operator für <code>T</code> -Elemente aufgerufen!)
<code>*o_iter</code> <code>++o_iter</code> <code>o_iter++</code>	liefert den Iterator <code>o_iter</code> selbst, sonst geschieht nichts.

Beispiel:

```
#include <iostream>
#include <iterator>
...
// Ostream-Iterator zur Ausgabe von int's auf cout,
// Trennzeichen ist ein Doppelpunkt:
ostream_iterator<int> o_iter( cout, " : ");
...
for ( i = 0; i < 10; ++i)
    o_iter = i; // Zuweisung an Ostream-Iterator -> Ausgabe
                // des Wertes und des Trennzeichens!
...
```

Istream-Iteratoren Istream-Iteratoren sind das Gegenstück zu Ostream-Iteratoren: es wird nicht elementweise auf einen Ostream geschrieben, sondern elementweise von einem Istream gelesen.

Hierbei tritt zusätzlich folgendes Problem auf: es muss das Ende der Eingabe erkannt werden.

Hierzu gibt es neben dem Konstruktor:

```
istream_iterator<T> i_iter(istream);
```

welcher einen Istream-Iterator mit Namen `i_iter` erzeugt, welcher Elemente vom Typ `T` vom angegebenen `istream` (implizit mit dem `>>`-Operator) liest, den Standard-Konstruktor:

```
istream_iterator<T> i_eof();
```

der den sogenannten *End-Of-Stream*-Iterator (hier mit dem Namen `i_eof`) erzeugt, mit dem der Istream-Iterator `i_iter` verglichen werden kann (auch hier muss der Elementtyp `T` angegeben werden!)! Die "Iteratorposition" von `i_eof` stellt in gewisser Hinsicht das Ende der Eingabesequenz dar (so wie `container.end()` das Ende der Sequenz der Containerelemente darstellt!).

Eine typische Anwendung könnte dann so aussehen:

```
istream_iterator<int> i_iter(cin); // Iterator zum Lesen
                                // von int's von cin
istream_iterator<int> i_eof();    // int-EOF-Iterator
while ( i_iter != i_eof )
{ // solange das Lesen von int's noch nicht beendet ist!
    ...
}
```

Bei der Erzeugung eines Istream-Iterators (nicht beim EOF-Iterator) wird gleichzeitig versucht, ein Element des angegebenen Types vom angegebenen Istream zu lesen (hier im Beispiel also ein `int` von `cin`). Gelingt dies nicht, stimmt sofort dieser Iterator mit dem EOF-Iterator überein, d.h. das Lesen von Werten vom Typ `T` "ist sofort am Ende angekommen".

Die Operatoren für einen Istream-Iterator sind:

<code>*i_iter</code>	liefert den Wert des zuletzt eingelesenen Elementes (vom Typ <code>T</code>).
<code>i_iter -> komp</code>	liefert eine Komponente des zuletzt eingelesenen Elementes (vom Typ <code>T</code>).
<code>++i_iter</code>	liest das nächste Element (vom Typ <code>T</code>) und gibt “neue” Position zurück (d.h. <code>*++i_iter</code> ist das gerade gelesene Element).
<code>i_iter++</code>	liest das nächste Element (vom Typ <code>T</code>), gibt aber “alte” Position zurück (d.h. <code>*i_iter++</code> liest ein neues Element, ist aber das davor gelesene Element!).
<code>i_iter1 == i_iter2</code>	testet Istream-Iteratoren auf Gleichheit.
<code>i_iter1 != i_iter2</code>	testet Istream-Iteratoren auf Ungleichheit.

Gleich sind zwei Istream-Iteratoren, wenn beide vom gleichen Istream lesen und beide noch lesen können (d.h. noch nicht am Ende angekommen sind!).

11.3 Standardcontainer

Containerklassen sind Datentypen, welche Objekte eines beliebigen anderen Types (etwa Typ `T`) “aufnehmen” können und diese Objekte verwalten.

In den unterschiedlichen Containerklassen spiegelt sich die unterschiedliche Art der “Abspeicherung” der Objekte in den Containern und damit verbunden die unterschiedliche Verwaltung der Objekte, etwa unterschiedlicher Zugriff auf die einzelnen Elemente wider.

Die Containerklassen der Standardbibliothek sind als Templates (Schablonen) realisiert, so dass es nicht auf den eigentlichen Typ der verwalteten Objekte ankommt und man für einen beliebigen, auch selbstdefinierten Datentyp die entsprechende Containerklasse direkt zur Verfügung hat.

Im Standard sind als Containerklassen realisiert:

- Vektoren von beliebigem Typ (Abstraktion von Feldern, mit Indizierung aber dynamisch).
- Schlangen (*Fifo*-Speicher, *First in, first out*).
- Kellerspeicher (*Lifo*-Speicher, *Last in, first out*).
- Doppelt verkettete Listen.
- Klassen zur Verwaltung von Mengen mit beliebigen Elementen.
- Assoziative Felder (Feld von Wertepaaren, wobei mit dem jeweils ersten Element eines Paares indiziert werden kann).

Zu jedem dieser Standardcontainertypen existiert eine zugehörige Headerdatei, welche vor der Verwendung eingebunden werden muss.

Bei der Erzeugung eines Containers wird dann der konkrete, zu Grunde liegende Datentyp angegeben, d.h. anstelle des oben angegebenen `T` wird ein konkreter Typ angegeben:

```

#include <vector>
#include <list>

...
class A { ... };           // selbstdefinierter Datentyp
...
vector<int> iv(1000);      // int-Vektor der Laenge (zunaechst) 1000
vector<double> dv(100);   // double-Vektor der Laenge (zunaechst) 100
vector<A> av(10);         // A-Vektor der Lanegen (zunaechst) 10
...
list<int> il;              // (zunaechst leere) lineare Liste von int-Elem.
list<double> dl;          // (zunaechst leere) lineare Liste von double-Elem.
list<A> al;               // (zunaechst leere) lineare Liste von A-Elem.
...

```

Der Typ aller in obigem Beispiel angegebenen Container ist unterschiedlich, er hängt von der Containerklasse (`vector`, `list`, ...) und vom Elementtyp des Containers, also davon, ob `int`-, `double`-, `A`- oder andersartige Elemente im Container abgelegt sind.

11.3.1 Gemeinsamkeiten aller Containerklassen

Zu jeder Containerklasse sind einige Typen und Funktionen definiert, welche mit gleicher Syntax und Semantik (aber möglicherweise unterschiedlicher Effizienz) für jeden Container angewendet werden können:

In allen Containerklassen definierte Typnamen

In jeder Containerklasse sind u.a. folgende Typen definiert:

Typ	Erläuterung
<code>value_type</code>	Anderer Name für den bei der Template-Erzeugung angegebenen Typen <code>T</code> . Bei assoziativen Feldern (<code>map<key,T></code> und <code>multimap<key,T></code>) steht <code>value_type</code> für den Datentyp <code>pair<const key,T></code> , siehe Abschnitt 11.3.7.
<code>size_type</code>	(Ganzzahliger) Datentyp für die Anzahl der Elemente im Container.
<code>iterator</code> <code>const_iterator</code> <code>reverse_iterator</code> <code>const_reverse_iterator</code>	Datentypen, mit denen man "der Reihe nach" auf alle Elemente eines Containers zugreifen kann (s.u.)
<code>difference_type</code>	(Ganzzahliger) Datentyp für "Iteratorabstände".
<code>reference</code>	Referenztyp für Objekte vom Typ <code>value_type</code> (wird von einigen Elementfunktionen als Ergebnis geliefert).
<code>const_referenz</code>	Referenztyp — aber Referenz auf <code>const</code>
<code>pointer</code>	Zeigertyp für Objekte vom Typ <code>value_type</code> (wird von einigen Elementfunktionen als Ergebnis geliefert).
<code>const_pointer</code>	Zeigertyp — aber Zeiger auf <code>const</code>

Iteratoren

Jeder Container wird (u.a.) als eine Folge (Sequenz) von Objekten des entsprechenden Types `value_type` angesehen. Iteratoren bieten die Möglichkeit, “der Reihe nach“ auf alle in einem Container enthaltenen Objekte zuzugreifen. Die natürliche Reihenfolge hängt hierbei von der Art des Containers ab!

Zu jedem Containertyp existieren zugehörige Iteratortypen, wobei der eigentliche Typ eines Iterators vom Container (und vom bei der Containererzeugung angegebenen Typen `T`) abhängt. Iteratoren sind eigenständige Objekte, welche jedoch mit einem konkreten Container “in Verbindung“ stehen.

Die Semantik des Zugriffs mittels eines Iterators auf die Elemente eines Containers ist hierbei an die Verwendung von Zeigern bei Feldern angelehnt, d.h. zugegriffen wird über den Verweisoperator `*` (Ergebnis des Verweisoperators ist von dem in der Containerklasse definierten Typ `reference`) und durch Anwendung des Inkrement-Operators `++` (sinnvollerweise in der Praefix-Form anzuwenden) wird der Iterator auf das (in der Reihenfolge) nächste Element des Containers “gesetzt“.

Verglichen werden können zwei Iteratoren mit dem Vergleichsoperator `==` (bzw. `!=`), welcher den Wert *wahr* liefert, wenn beide Iteratoren (nicht) auf dasselbe Element desselben Containers “zeigen“.

Desweiteren ist der Zuweisungsoperator `=` für typgleiche Iteratoren aller Containerklassen definiert.

Mittels der `const`-Iteratoren können die Objekte nicht verändert werden, bei den `reverse`-Iteratoren wird die “natürliche“ Reihenfolge der Elemente des Containers umgekehrt, d.h. er wird von “hinten“ nach “vorne“ durchlaufen.

Die Definition von Iteratoren könnte wie folgt aussehen:

```
#include <list>
#include <vector>
...
class A { ... };    // selbstdefinierter Datentyp
...
list<int> intlist;
vector<A> Avector;
...
list<int>::iterator intlist_iterator; // Iterator fuer Typ list<int>
vector<A>::iterator Avector_iterator; // Iterator fuer Typ vector<A>
...
```

Zu jeder Containerklasse *ContTyp* gibt es einige Elementfunktionen, welche als Ergebnis des Wert eines Iteratortyps *IterTyp* liefern:

<i>IterTyp ContTyp::begin();</i>	Liefert einen (Vorwärts)-Iterator auf das erste Element des Containers.
<i>IterTyp ContTyp::end();</i>	Liefert einen (Vorwärts)-Iterator <u>hinter das letzte</u> Element des Containers.
<i>IterTyp ContTyp::rbegin();</i>	liefert einen (Rückwärts)-Iterator auf das letzte Element des Containers.
<i>IterTyp ContTyp::rend();</i>	liefert einen (Rückwärts)-Iterator <u>vor das erste</u> Element des Containers.

Bei einem konstanten Container sind die gelieferten Iteratoren `const`.

Wegen der Asymmetrie zwischen `begin()` (erstes Element) und `end()` (eins hinter dem letzten Element) bzw. zwischen `rbegin()` (letztes Element) und `rend()` (eins vor dem ersten Element) können in natürlicher Weise Schleifen über alle Elemente eines Containers formuliert werden:

```
...
list<int> intlist;           // Liste vom Typ int definieren
list<int>::iterator il_it;   // entsprechenden Iterator definieren
...
for ( il_it=intlist.begin(); il_it != intlist.end(); ++il_it)
{ // aktuelles Element der Liste bearbeiten, Zugriff auf dieses
  // Element erfolgt mit:  *il_it
  ...
}
...
```

Der Ausdruck

```
il_it=intlist.begin()
```

In der Initialisierung der `for`-Schleife weist dem Iterator `il_it` das Funktionsergebnis von `intlist.begin()` zu, also einen Iterator auf den Anfang des (konkreten) Containers `intlist` vom Typ `list<int>`. `il_it` wird somit auf den Anfang von `intlist` gesetzt!

In der Bedingung

```
il_it != intlist.end()
```

wird überprüft, ob sich der Iterator `il_it` von dem von `intlist.end()` gelieferten Iterator unterscheidet. Solange dies wahr ist, ist `il_it` noch nicht hinter's Ende des Containers `intlist` "angekommen". (Zu beachten ist, dass hier i. Allg. nicht der Vergleichsoperator, etwa `il_it < intlist.end()`, angewendet werden darf, da dieser nicht für alle Iteratoren von Standardcontainern definiert ist!)

Im Anweisungsteil der Schleife kann dann mittels `*il_it` auf das aktuelle Element des mit dem Iterator `il_it` verknüpften Containers `intlist` zugegriffen werden.

Im Inkrementierungsteil

```
++il_it
```

wird der Iterator um eins erhöht, zeigt also anschließend auf das nächste Objekt des Containers (bzw. hinter das Ende des Containers, wenn `il_it` zuvor auf das letzte Element gezeigt hat!).

Ein Rückwärtsdurchlauf ist etwa wie folgt möglich:

```
#include <list>
class A { ... }; // selbstdefinierter Datentyp
...
list<A> Alist;    // Liste von A-Objekten
list<A>::reverse_iterator ar_it; // Rueckwaerts-Iterator
...
for ( ar_it = Alist.rbegin(); ar_it != Alist.rend(); ++ar_it)
```

```
{
    /* aktuelles Element von Alist bearbeiten,
       Zugriff mittels: *ar_it
    */
    ...
}
...
```

Mittels

```
ar_it = Alist.rbegin()
```

wird der Iterator `ar_it` so initialisiert, dass er auf das letzte Element von `Alist` zeigt. Die Bedingung

```
ar_it != Alist.rend()
```

ist solange wahr, solange der Iterator noch nicht vor dem Anfang der Liste angekommen ist. (`Alist.rend()` bedeutet hierbei: hinter dem Ende des Rückwärtsdurchlaufs — also vor dem Anfang der Liste!)

Bei der Inkrementierung

```
++ar_it
```

mit dem Inkrement-Operator `++` wird der Iterator `ar_it` um eins erhöht — da es sich aber um einen Rückwärts-Iterator handelt, zeigt er anschließend auf das vorhergehende Element der Liste bzw. vor den Anfang der Liste, falls er vorher auf das erste Listenelement zeigte!

Zu einem konkreten Container können gleichzeitig mehrere Iteratoren verwendet werden, etwa, um einen Container gleichzeitig von “vorne” nach “hinten” und umgekehrt zu durchlaufen:

```
#include <list>
class A { ... };    // selbstdefinierter Datentyp
...
list<A> Alist;      // Liste von A-Objekten
list<A>::iterator av_it;    // Vorwaerts-Iterator
list<A>::reverse_iterator ar_it; // Rueckwaerts-Iterator
...
/* av_it laeuft von vorne nach hinten und
   ar_it laeuft von hinten nach vorne
*/
for ( av_it = Alist.begin(), ar_it = Alist.rbegin();
      ar_it != Alist.rend();
      ++av_it, ++ar_it)
{
    // Zugriff mittels *ar_it und *av_it
    ...
}
...
```

Konstruktoren, Destruktor

Zu jedem Containertyp *ContTyp* gibt es folgende Konstruktoren:

- den (expliziten) Default-Konstruktor (ohne Argument):

```
explicit ContTyp::ContTyp();
```

Der Aufruf

```
ContTyp a;
```

erzeugt einen leeren Container **a** vom Typ *ContTyp*.

Dieser Konstruktor ist in allen Containerklassen explizit, damit er vom System nicht für automatische Typumwandlungen “missbraucht” wird.

- den Copy-Konstruktor (mit einem Container vom gleichen Typ als Argument):

```
ContTyp::ContTyp(const ContTyp& );
```

Der Aufruf

```
ContTyp a(b);
```

erzeugt einen neuen Container **a** als Kopie des vorhandenen Containers **b** (muss den gleichen Typ *ContTyp* haben!).

- den Konstruktor mit Initialisierung durch eine Sequenz:

```
ContTyp::ContTyp( IterTyp anf , IterTyp ende );
```

Der Aufruf

```
ContTyp a(anfang, ende);
```

erzeugt einen Container **a** und initialisiert ihn mit Kopien von Elementen, welche aus dem durch die Iteratoren **anfang** (einschließlich) bis **ende** (ausschließlich) gegebenen Bereich stammen (meistens eines anderen Containers mit gleichem **value_type**).

(*IterTyp* ist ein zum Elementtyp des Containers passender Iteratortyp, es muss nicht unbedingt der zum Container selbst gehörende Iteratortyp sein!)

Zu jedem Containertyp existiert der entsprechende Destruktor

```
ContTyp::~ContTyp();
```

der am Ende der Lebenszeit eines Containers automatisch aufgerufen wird und für die ordnungsgemäße Zerstörung des Containers sorgt (etwa dynamischen Speicherbereich freigeben).

Anzahl der Elemente eines Containers

Mittels der Elementfunktion

```
size_type ContTyp::size() const;
```

erhält man die Anzahl der augenblicklich im Container abgespeicherten Elemente.

Die Elementfunktion


```
bool ContTyp::empty() const;
```

liefert, ob der Container leer ist (Ergebnis *wahr*) oder nicht (Ergebnis *falsch*)!

Die Elementfunktion

```
size_type ContTyp::max_size() const;
```

liefert als Ergebnis die (systemabhängige) maximale Anzahl von Elementen, die dieser Container aufnehmen kann.

Universelle Vergleichsoperatoren

Die Vergleichsoperatoren `==` und `<` sind für Standardcontainer definiert.

Sind **A** und **B** Container vom gleichen Typ (also gleiche Containerart — etwa **vector** — und gleicher Elementtyp **value_type** — etwa **int**), so liefert der Vergleich

```
A == B
```

genau dann *wahr*, wenn beide Container gleich viele Elemente beinhalten und elementweise übereinstimmen, d.h. das erste Element von **A** gleich dem ersten Element von **B** ist, das zweite Element von **A** mit dem zweiten Element von **B** übereinstimmt usw.. (Elementgleichheit wird hierbei mit dem Operator `==` für den gemeinsamen Elementtypen festgestellt!)

Der Vergleich

```
A < B
```

vergleicht die beiden Container “lexikalisch“, d.h.

- es werden der Reihe nach (spätestens, bis einer der Container am Ende angekommen ist) die (korrespondierenden) Elemente der beiden Container miteinander verglichen (d.h. erstes Element von **A** mit erstem Element von **B**, zweites Element von **A** mit zweitem Element von **B** usw.).

Der Vergleich eines Elementes (etwa des **k**-ten) von **A** mit dem korrespondierenden (also dem **k**-ten) von **B** geschieht (standardmäßig) mit dem Vergleichsoperator `<` des (gemeinsamen) Elementtypes **value_type** beider Container wie folgt:

Es wird zunächst der Vergleich

```
(k-tes Element von A) < (k-tes Element von B)
```

durchgeführt, liefert dieser den Wert *wahr*, so wird der Vergleich beendet und das **k**-te Element von **A** ist kleiner als das **k**-te Element von **B**.

Ist dieser Vergleich jedoch *falsch*, so wird anschließend der (umgekehrte) Vergleich:

```
(k-tes Element von B) < (k-tes Element von A)
```

durchgeführt.

Liefert auch dieser *falsch*, so werden die beiden Elemente als *gleich* angesehen (Gleichheit wird also ohne den Operator `==` festgestellt!).

Ansonsten ist das **k**-te Element von **A** größer als das **k**-te Element von **B**.

- Beim der ersten auftretenden Ungleichheit zweier Elemente von **A** und **B** der beiden Container (diese trete beim Vergleich der **k**-ten Elemente auf), wird der

Vergleich der beiden Container beendet und $A < B$ ist genau dann wahr, wenn das k -te Element von A kleiner ist als das k -te Element von B .

- Tritt bis zum “Ende“ eines der beiden Container keine Ungleichheit von Elementen auf, so wird die Größe der beiden Container beachtet:
Ist $A.size() < B.size()$, (d.h. beim Vergleich der Elemente ist Container A ans Ende angekommen, B jedoch noch nicht), so ist das Ergebnis von $A < B$ *wahr*.
Ansonsten ist das Ergebnis von $A < B$ *falsch*!

Da mittels Template-Funktionen in der Standardbibliothek für jeden Datentyp, für den die Vergleichsoperatoren $=$ und $<$ definiert sind, die anderen Vergleichsoperatoren $!=$, $>$, $<=$ und $>=$ auf diese Operatoren $=$ und $<$ zurückgeführt werden (siehe Abschnitt 11.1.1), sind diese “anderen“ Vergleichsoperatoren auch für Standardcontainer verfügbar!

Sonstige Funktionen

Durch folgende Element-Funktionen kann man auf das erste bzw. letzte Element eines (von den Containerklassen `set<T>`, `multiset<T>`, `map<Key,T>` und `multimap<Key,T>` abgesehen) jeden Containers zugreifen (der Containertyp sei *ContTyp* und der Elementtyp des Containers sei T):

reference <i>ContTyp</i> ::front()	Zugriff auf's erste Element
reference <i>ContTyp</i> ::back()	Zugriff auf's letzte Element

Funktionsergebnis ist jeweils vom (Container-eigenen) **reference**-Typ. Das Funktionsergebnis ist nur dann definiert, wenn der Container nicht leer ist!

Sind A und B Container vom gleichen Typ (gleiche Containerart und gleicher Elementtyp), so ist die Zuweisung

$A = B;$

definiert. Hierdurch erhält A genau die gleichen Elemente wie B .

Mittels der Funktion `swap` kann man die Elemente zweier Container (vom gleichen Typ *ContTyp*) vertauschen.

Diese Funktion gibt es

- als Memberfunktion

```
void ContTyp::swap( ContTyp& );
```

Aufruf also `a.swap(b)`, falls a und b entsprechende Container sind,

- und auch als globale Funktion

```
void swap( ContTyp& , ContTyp& );
```

Aufruf also `swap(a,b)` mit a und b wie oben.

Die Element-Funktion

```
void ContTyp::clear();
```

löscht den Inhalt des Containers, für den sie aufgerufen wird — anschließend ist der Container also leer. (Hierbei wird bei selbstdefiniertem Elementtyp für jedes “zu löschende” Element der Destruktor aufgerufen.)

In Zusammenhang mit Iteratoren auf einen Container gibt es für jeden Containertyp *ContTyp* noch die Element-Funktionen:

– `iterator ContTyp::insert(iterator pos, value_type elem);`

fügt vor die Stelle im Container, welche durch den Iterator `pos` angezeigt wird, eine Kopie des als zweites Argument angegebenen Wertes `elem` vom Typ `value_type` ein. Funktionsergebnis ist die Iteratorposition des neu eingefügten Elementes.

– `iterator ContTyp::erase(iterator pos);`

löscht im Container das an der durch den Iterator `pos` angezeigten Stelle stehende Element. Funktionsergebnis ist die Iteratorposition des ursprünglichen Nachfolgers des entfernten Elementes bzw. die Iteratorposition `end()`, falls das letzte Element gelöscht wurde.

(Diese Funktion hat bei den Containerklassen `set<T>`, `multiset<T>`, `map<Key,T>` und `multimap<Key,T>` kein Funktionsergebnis!)

– `iterator ContTyp::erase(iterator anf, iterator end);`

löscht im Container alle Elemente, angefangen von dem durch den Iterator `anf` gegebenen (einschließlich) bis zu dem durch den Iterator `end` gegebenen (ausschließlich). Funktionsergebnis ist die Iteratorposition des ursprünglichen Nachfolgers des letzten gelöschten Elementes bzw. `end()`, falls es hinter dem letzten gelöschten Element kein Element mehr gibt!

(Diese Funktion hat bei den Containerklassen `set<T>`, `multiset<T>`, `map<Key,T>` und `multimap<Key,T>` kein Funktionsergebnis!)

Diese Einfüge- und Lösch-Funktionen sind für einige Standard-Containertypen, etwa Vektoren, nicht sehr effizient!

Allokatoren für Standardcontainer

Alle im Standard definierten Containerklassen können dynamisch vergrößert werden — zur Verwendung von Standardcontainern ist also eine dynamische Speicherverwaltung notwendig.

Der Standard sieht vor, dass bei der Erzeugung eines Containers die Art der dynamischen Speicherverwaltung (als Template-Argument) angegeben werden kann — die Funktionalität einer solchen Speicherverwaltung ist als Abstraktion *Allokator* im Standard definiert — man kann also bei der Erzeugung eines Containers einen Allokator angeben.

Gibt man keinen Allokator an, so wird die normale Speicherverwaltung mittels `new` und `delete` verwendet.

Im Folgenden wird nur diese “normale” Speicherverwaltung zu Grunde gelegt und es wird vom Allokator-Template-Argument bei den Containerklassen abgesehen!

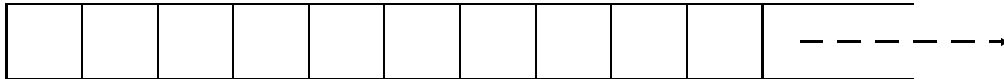
11.3.2 Die Containerklasse `vector<T>`

Die Containerklasse `vector<T>` repräsentiert ein dynamisches Feld von Elementen vom Typ `T`.

Mit einem Vektor `vector<T>` kann also wie mit einem (eingebauten) Feld vom Typ `T` mittels Indizierung gearbeitet werden — zusätzlich zu eingebauten Feldern (*Array's*) können Vektoren jedoch dynamisch vergrößert (und eingeschränkt auch verkleinert) werden.

Zur Verwendung von `vector<T>`'en muss die Headerdatei `<vector>` eingebunden werden!

Der Aufbau eines Vektors kann wie folgt veranschaulicht werden:



wobei jedes Quadrat ein Element vom Typ `T` repräsentiert.

Typen der Vektorklasse

In der Template-Klasse `vector<T>` sind (wie bei allen Containern üblich) folgende Typen definiert (vgl. Abschnitt 11.3.1):

```
template <class T>
class vector {
    ...
public:
    ...
    typedef ... reference;
    typedef ... const_reference;
    typedef ... iterator;
    typedef ... const_iterator;
    typedef ... reverse_iterator;
    typedef ... const_reverse_iterator;
    typedef ... size_type;
    typedef ... difference_type;
    typedef T    value_type;
    typedef ... pointer;
    typedef ... const_pointer;
    ...
};
```

Die Bedeutung dieser Typen ist in Abschnitt 11.3.1 erläutert.

Erzeugen, Zuweisen, Zerstören eines Vektors

Folgender Ausschnitt aus der Klassendefinition der Template-Klasse `vector<T>` zeigt die Möglichkeiten, ein `vector<T>`-Objekt zu erzeugen, zerstören und ihm etwas zuzuweisen:

```

template <class T>
class vector {
    ...
public:
    ...
    explicit vector();           // Standardkonstruktor
    vector( const vector<T>&);    // Copy--Konstruktor

    template <class InputIterator> // mit Sequenz initialisieren
    vector( InputIterator anf, InputIterator ende);

    explicit vector ( size_type n, const T& value = T() );

    ~vector();                   // Destruktor

    vector<T> & operator=( const vector<T> &); // Zuweisungsoperator

    void assign( size_type n, const T& value); // Zuweisungsfunktion

    template <class InputIterator> // Zuweisungsfunktion
    void assign( InputIterator anf, InputIterator ende);
    ...
};

```

Neben der in Abschnitt 11.3.1 erläuterten, für alle Container vorhandenen Funktionalität (Standardkonstruktor, Copy-Konstruktor, Initialisierung mit Sequenz, Destruktor und Zuweisungsoperator) gibt es somit zusätzlich

- den Konstruktor:

```
explicit vector ( size_type n, const T& value = T() );
```

der einen neuen Vektor der Länge *n* erzeugt, wobei jedes der *n* Elemente mit dem angegebenen Wert *value* initialisiert wird. Ist kein Initialisierungswert angegeben, wird der Wert mit dem Standardkonstruktor des Types *T* erzeugt,

- die Zuweisungsfunktion

```
void assign( size_type n, const T& value);
```

welche einen vorhandenen Vektor mit einem Vektor aus *n* Elementen mit Wert *value* überschreibt,

- der Zuweisungsfunktion

```

template <class InputIterator>
void assign( InputIterator anf, InputIterator ende);

```

welche einen vorhandenen Vektor mit einem Vektor, dessen Elemente sich aus der durch Iteratoren gegebenen Sequenz `[anf, ende)` ergeben, überschreibt. (Der Elementtyp des Iteratortypes `InputIterator` muss zum Typ `T` des Vektors passen!)

Man muss unterscheiden zwischen

– `vector<T> a(10);`

erzeugt einen Vektor der Länge 10 mit 10 Standard-Elementen vom Typ `T`.

– `vector<T> a[10];`

Erzeugt ein Feld (Array) der Länge 10, wobei jedes Feldelement ein (leerer) Vektor vom Typ `T` ist!

Größe und Kapazität eines Vektors

Bei Vektoren muss man zwischen der Größe (= Anzahl der im Vektor abgespeicherten Elemente) und Kapazität (= Anzahl der im für den Vektor reservierten Speicherplatz passenden Elemente vom Typ `T`) unterscheiden. Die Kapazität kann größer als die augenblickliche Anzahl der Elemente sein.

```
template <class T>
class vector {
    ...
public:
    ...
    size_type size() const;      // aktuelle Groesse
    size_type max_size() const; // maximale Groesse
    bool empty() const;         // Vektor leer?

    size_type capacity() const; // Kapazitaet

    // Vektor vergroessern/verkleinern:
    void resize(size_type n, T value = T());

    // Kapazitaet vergroessern:
    void reserve(size_type n);
    ...
};
```

Zur Ermittlung der Kapazität eines Vektors steht die Element-Funktion:

```
size_type capacity() const;
```

zur Verfügung. Die Kapazität eines Vektors (Ergebnis von `capacity()`) ist immer größer gleich seiner Größe (Ergebnis von `size()`). Solange die Kapazität des Vektors noch nicht ausgeschöpft ist, können in diesen Vektor noch weitere Elemente abgelegt werden, ohne dass vom System eine (möglicherweise “teure”) Speicheranforderung durchgeführt werden muss.

Bei der Erzeugung eines Vektors durch einen Konstruktor stimmen Größe und Kapazität zunächst überein! Die einzige Möglichkeit, die Kapazität eines Vektors — unter Beibehaltung seiner Größe — zu erhöhen, ist der Aufruf der Element-Funktion:

```
void reserve(size_type n);
```

Hierdurch wird die Kapazität des Vektors, für den sie aufgerufen wird, auf *n* erhöht, falls *n* größer als die bisherige Kapazität des Vektors ist. Ist *n* kleiner gleich der bisherigen Kapazität, so geschieht nichts!

Beispiel:

```
...
vector<int> iv;    // leeren Vektor erzeugen, size=capacity=0
iv.reserve(1000); /* Kapazitaet vergroessern, size=0, capacity=1000!
                  Die Groesse des Vektors ist immer noch 0 und der
                  Vektor leer, er kann aber bis zu 1000 int's
                  aufnehmen, ohne, dass eine neue
                  Speicheranforderung noetig wird!                */
...
```

Wird ein Vektor mittels **reserve** vergrößert, so wird natürlich neuer Speicher angefordert — i. Allg. wird neuer Speicher für den ganzen Vektor angefordert, die bisherigen Elemente in den neuen Speicherbereich hineinkopiert und anschließend der bisherige Speicherbereich freigegeben! Somit werden ggf. die Adressen (Iterator-Positionen) der bisherigen Elemente ungültig und beim Umkopieren der Elemente von der alten Stelle zur neuen und der Freigabe des alten Speicherbereiches werden bei einem selbstdefinierten Datentyp *T* etliche Konstruktor- und Destruktoraufrufe automatisch durchgeführt!

Bei großen Vektoren kann natürlich die Anforderung von neuem Speicher schiefgehen, in diesem Fall wird standardmäßig mit der Fehlermeldung **out of memory** das Programm beendet!

Die Element-Funktion:

```
void resize(size_type n, T value = T() );
```

ändert die Größe eines Vektors auf *n*.

Man muss folgende Fälle unterscheiden:

1. *n* ist kleiner gleich der bisherigen Größe:
In diesem Fall bleiben die ersten *n* Elemente des Vektors erhalten, die restlichen werden freigegeben (Destruktor) und die Größe des Vektors wird auf *n* verringert (Kapazität bleibt erhalten!).
2. *n* ist größer als die bisherige Größe, aber kleiner gleich der Kapazität des Vektors:
Die bisherigen Elemente des Vektors bleiben erhalten, der Vektor wird bis zur neuen Größe *n* mit Kopien des als zweites Argument angegebenen Wertes aufgefüllt. Ist kein zweites Argument angegeben, so werden die neuen Elemente standardmäßig vorbesetzt (Default-Konstruktor). Die Kapazität des Vektors bleibt erhalten.

3. n ist größer als die bisherige Kapazität:

In diesem Fall wird i. Allg. zunächst mal Platz für einen neuen Vektor mit Größe und Kapazität gleich n reserviert, die bisherigen Elemente werden an den neuen Platz kopiert und die restlichen Elemente mit dem angegebenen Wert (bzw. mit dem T-Standardwert) initialisiert. Der bisherige Speicherbereich wird freigegeben. (Dies ist manchmal eine ziemlich teure Operation!)

Aus dem oben Erläuterten ist ersichtlich, dass jede Operation, in der die Kapazität eines Vektors erhöht werden muss, ziemlich aufwändig ist. Deswegen sollten Kapazitätserhöhungen so selten wie möglich eingesetzt werden, d.h. man sollte Vektoren zunächst mit einer voraussichtlich ausreichenden Kapazität versehen und, falls erforderlich, die Kapazität dann gleich um einen größeren Betrag erhöhen.

Sind bei einem “vollen” Vektor (Kapazitätsgrenze erreicht) etwa noch 10 Elemente anzuhängen, so ist schlecht (obwohl es funktioniert) jedes der anzuhängenden Elemente mit `push_back(...)` (s.u.) anzuhängen — es sind hierbei 10 einzelne Speicheranforderungen nötig! Besser ist es, die Kapazität des Vektors mit `reserve` gleich um einen ausreichenden Betrag (≥ 10) zu erhöhen und dann jedes einzelne Element mit `push_back(...)` anzuhängen — hier ist nur eine Speicherneuanforderung nötig!

Vektor-Iteratoren

Es stehen die üblichen Iteratortypen und Funktionen zur Verfügung, welche Iteratorpositionen liefern.

```
template <class T>
class vector {
    ...
public:
    ...
    // Vorwaerts-Iterator,
    // zeigt auf erstes Element eines Vektors:
    iterator begin();

    // const-Vorwaerts-Iterator,
    // zeigt auf erstes Element eines const-Vektors:
    const_iterator begin() const;

    // Vorwaerts-Iterator,
    // zeigt hinter letztes Element eines Vektors:
    iterator end();

    // const-Vorwaerts-Iterator,
    // zeigt hinter letztes Element eines const-Vektors:
    const_iterator end() const;

    // Rueckwaerts-Iterator,
    // zeigt auf letztes Element eines Vektors:
```



```

reverse_iterator rbegin();

// const-Rueckwaerts-Iterator,
// zeigt auf letztes Element eines const-Vektors:
const_reverse_iterator rbegin() const;

// Rueckwaerts-Iterator,
// zeigt vor erstes Element eines Vektors:
reverse_iterator rend();

// const-Rueckwaerts-Iterator,
// zeigt vor erstes Element eines const-Vektors:
const_reverse_iterator rend() const;
...
};

```

Da es sich bei den Iteratortypen des Containertypes `vector<T>` um Random-Access-Iteratoren handelt, ist mittels eines solchen Iterators wahlfreier Zugriff auf die Elemente des Vektors möglich, d.h. ist neben den für alle Iteratortypen erlaubten Operationen:

<code>iter1 == iter2</code>	Gleichheit zweier Iteratoren, genau dann wahr, wenn beide Iteratoren auf dasselbe Element desselben Containers zeigen.
<code>iter1 != iter2</code>	Ungleichheit zweier Iteratoren <code>!(iter1 == iter2)</code> .
<code>*iter</code>	Zugriff auf das aktuelle Element.
<code>iter->komponente</code>	Zugriff auf eine Komponente des Aktuellen Elementes.
<code>++iter</code>	Weitersetzen des Iterators, liefert neue Position.
<code>iter++</code>	Weitersetzen des Iterators, liefert alte Position.
<code>....::iterator iter2(iter1)</code>	Copy-Konstruktor.

zusätzlich die Anwendung folgender Operatoren auf solche Iteratoren möglich (vgl. Abschnitt 11.2):

<code>--iter</code>	Zurücksetzen des Iterators um eine Position, liefert neue Position.
<code>iter--</code>	Zurücksetzen des Iterators um eine Position, liefert alte Position.
<code>iter1 = iter2</code>	Zuweisen von Iteratoren.
<code>iter[n]</code>	Zugriff auf das <i>n</i> -te Element hinter (bzw. vor, falls <i>n</i> negativ ist) dem, auf welches der Iterator zeigt.
<code>iter += n</code>	Weitersetzen um <i>n</i> Positionen. (Bzw. Zurücksetzen, falls <i>n</i> negativ ist!)

<code>iter + n</code>	Iterator auf das um <code>n</code> Positionen verschobene Element — vorwärts, falls <code>n</code> positiv und rückwärts, falls <code>n</code> negativ ist. (Iterator <code>iter</code> behält seinen Wert, das Ergebnis ist ein Iterator auf das entsprechende Element — nicht das Element selbst!)
<code>n + iter</code>	wie <code>iter + n</code> .
<code>iter - n</code>	Iterator auf das um <code>n</code> Positionen verschobene Element — rückwärts, falls <code>n</code> positiv und vorwärts, falls <code>n</code> negativ ist. (Iterator <code>iter</code> behält seinen Wert, das Ergebnis ist ein Iterator auf das entsprechende Element — nicht das Element selbst!)
<code>iter1 - iter2</code>	Abstand (vom Typ <code>difference_type</code>) zwischen <code>iter1</code> und <code>iter2</code> .
<code>iter1 < iter2</code> <code>iter1 > iter2</code> <code>iter1 <= iter2</code> <code>iter1 >= iter2</code>	liefert, ob <code>iter1</code> vor <code>iter2</code> liegt. liefert, ob <code>iter1</code> hinter <code>iter2</code> liegt. liefert, ob <code>iter1</code> nicht hinter <code>iter2</code> liegt. liefert, ob <code>iter1</code> nicht vor <code>iter2</code> liegt.

Beim wahlfreien Zugriff mittels Iteratoren auf einen Vektor muss man selbst darauf achten, dass man nicht über Anfang bzw. Ende des Vektors hinausläuft, hier kann bzw. muss man ggf. abfragen:

```
...
vector<int> a(100);    // int-Vektor der Laenge 100
vector<int>::iterator a_it=a.begin();
...                  // Iterator auf Anfang von a
...a_it[1000];        // Fehler: Zugriff auf nicht vorhandenes Element,
                      // wird vom Compiler nicht erkannt, kann aber zum
                      // Programmabsturz fuehren!
a_it += 1000;         // Kein Fehler, Iterator zeigt hinter den Vektor!
...*a_it;             // Fehler: Zugriff auf nicht vorhandenes Element,
                      // wird vom Compiler nicht erkannt, kann aber zum
                      // Programmabsturz fuehren!
if ( (a_it >= a.begin()) && (a_it < a.end()) )
{
    ... *a_it;        // Zugriff nur, wenn a_it im erlaubten Bereich!
}
...
```

Vektor-Elementzugriff

Der Zugriff auf ein einzelnes Element des `vector<T>` ist mit folgenden Operatoren bzw. Funktionen möglich:

```
template <class T>
class vector {
    ...
public:
    ...
    // ungepruefter Index-Zugriff
```

```

reference      operator[] (size_type n);
const_reference operator[] (size_type n) const;

// geprüfter Index-Zugriff
reference      at(size_type n);
const_reference at(size_type n) const;

// ungeprüfter Zugriff aufs erste Element (Index: 0)
reference      front();
const_reference front() const;

// ungeprüfter Zugriff aufs letzte Element (Index: size()-1)
reference      back();
const_reference back() const;
...
};

```

Wie bei eingebauten Feldern (Arrays) ist durch den Operator `[]` der Zugriff auf die Elemente eines Vektors durch Indizierung möglich.

Der Index innerhalb der eckigen Klammern muss ganzzahlig sein und der Anwender muss selbst darauf achten, dass der Index im gültigen Bereich (von 0 bis `size()-1`) liegt (ungeprüfter Zugriff):

```

...
vector<int> a(5);    // Vektor der Laenge 5
...
...a[100]...;       // Index zu gross, undefiniertes Verhalten
...

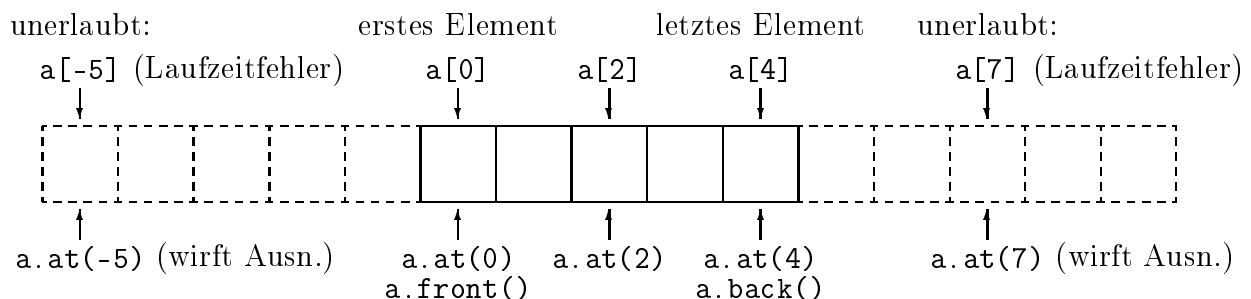
```

Alternativ ist im Standard (funktioniert noch nicht beim GCC-Compiler) der geprüfte Elementzugriff mittels der Element-Funktion

```
reference at(size_type n);
```

vorgesehen. Greift man mit dieser Funktion auf einen Vektor vor dessen Anfang (Argument < 0) oder hinter dessen Ende (Argument $\geq \text{size}()$) zu, so wird eine Ausnahme vom Typ `out_of_range` ausgeworfen!

Der Aufbau des Vektors `vector<T> a(5);` und die Zugriffe auf dessen Elemente können wie folgt veranschaulicht werden (wobei jedes durchgezogene Quadrat ein Element vom Typ `T` und jedes gestrichelte Quadrat ein nicht vorhandenes Element darstellt!):



Wie in Abschnitt 11.3.1 bereits erwähnt, kann man mit den Element-Funktionen `front()` bzw. `back()` auf das erste bzw. letzte Element eines (nicht leeren) Vektors zugreifen.

Ändern eines Vektors

Für Objekte vom Typ `vector<T>` sind folgende Funktionen definiert, mit denen der `vector<T>` abgeändert werden kann (Einfügen, Löschen von Elementen, zwei ganze `vector<T>`en vertauschen):

```
template <class T>
class vector {
    ...
public:
    ...
    void push_back(const T& x);
    void pop_back();

    iterator insert(iterator pos, const T& x);
    void insert(iterator pos, size_type n, const T& x);

    template <class InputIterator>
    void insert(iterator pos, InputIterator anf, InputIterator ende);

    iterator erase(iterator pos);
    iterator erase(iterator anf, iterator ende);

    void clear();
    void swap(vector<T> &);
};
```

Die Element-Funktion:

```
void push_back(const T&);
```

hängt eine Kopie des angegebenen Argumentes vom Typ `T` "hinten" an den Vektor an, d.h. die Vektorlänge wird um eins größer und das neue Element ist das mit dem größten Index.

Man muss zwei Fälle unterscheiden:

1. Die Kapazität des Vektors reicht aus (d.h. die Kapazität ist größer als die bisherige Größe):
Dann wird im vorhandenen Speicherplatz das neue Element angehängt.
2. Die Kapazität des Vektors reicht nicht aus (d.h. die Kapazität ist gleich der bisherigen Größe):
I. Allg. wird dann zunächst mal ausreichender Speicherplatz für den vergrößerten Vektor reserviert, die bisherigen Elemente werden an die neue Stelle kopiert (Copy-Konstruktoren und Destruktoren!), eine Kopie des neuen Elementes wird angehängt und der bisherige Speicherbereich freigegeben.
(Dies ist manchmal eine ziemlich teure Operation!)

Die Element-Funktion:

```
void pop_back();
```

entfernt das “hinterste“ Element (das mit dem größten Index) aus dem Vektor (der Vektor darf nicht leer sein!) und verringert die Größe des Vektors um eins — die Kapazität des Vektors bleibt jedoch erhalten.

Diese Funktionen `push_back`, `pop_back` und `back` sind für Vektoren (von Speicheranforderungen abgesehen) effizient, da die bisherigen Vektorelemente an ihre ursprünglichen Position verbleiben.

Mit folgenden (ineffizienten) Element-Funktionen kann man mitten in einem Vektor neue Elemente einfügen:

```
– iterator insert(iterator pos, value_type x);
```

fügt eine Kopie des angegebenen Elementes an die Position `pos`. Ergebnis ist die Iteratorposition auf das eingefügte Element.

```
– void insert(iterator pos, size_type n, const T& x);
```

fügt an der durch die Iteratorposition `pos` gegebenen Stelle `n` Kopien des angegebenen Elementes `x` ein.

```
– template <class InputIterator>
```

```
void insert(iterator pos, InputIterator anf, InputIterator end);
```

fügt in den Vektor an der Stelle `pos` Kopien der durch die Iteratoren `anf` und `end` gegebenen Sequenz ein. Der Iteratortyp `InputIterator` muss zum Elementtyp des Vektors passen.

Folgende (ebenso ineffizienten) Funktionen löschen Elemente mitten aus einem Vektor:

```
– iterator erase(iterator pos);
```

Löschen des Elementes an Position `pos`. Funktionsergebnis ist die Position des Vektor-Elementes hinter dem gelöschten.

```
– iterator erase(iterator anf, iterator end);
```

Löschen aller Elemente der durch `anf` und `end` gegebenen Teil-Sequenz des Vektors. Funktionsergebnis ist die Iteratorposition hinter das zuletzt gelöschte Element im ursprünglichen Vektor.

Bei all diesen Einfüge- und Löschfunktionen ist der Anwender selbst dafür verantwortlich, dass die beteiligten Iteratoren gültig sind, d.h. tatsächlich auf Elemente des Vektors zeigen.

Nach Einfügen oder Löschen von Elementen mitten in einem `vector<T>` erhalten die Elemente einen anderen Index und eine andere Adresse. Darüberhinaus werden auch bereits vorhandene Iteratoren ungültig, d.h. man muss sie anschließend neu initialisieren!

Die Funktion

```
void clear()
```

löscht alle Elemente des Vektors, dieser hat anschließend die Größe 0.

Mit der (möglicherweise sehr effizient implementierten) Elementfunktion:

```
void swap(vector<T>&);
```

werden zwei Vektoren vertauscht.

Globale Operatoren und Funktionen für Vektoren

Es sind die üblichen Vergleichs-Operatoren für Vektoren als globale Operator-Funktionen:

```
template <class T>
bool operator== (const vector<T>& x, const vector<T>& y);
```

```
template <class T>
bool operator< (const vector<T>& x, const vector<T>& y);
```

```
template <class T>
bool operator!= (const vector<T>& x, const vector<T>& y);
```

```
template <class T>
bool operator> (const vector<T>& x, const vector<T>& y);
```

```
template <class T>
bool operator>= (const vector<T>& x, const vector<T>& y);
```

```
template <class T>
bool operator<= (const vector<T>& x, const vector<T>& y);
```

sowie eine (möglicherweise sehr effizient implementierte) globale Funktion zum Vertauschen zweier Vektoren vorhanden:

```
template <class T>
void swap( vector<T>& x, vector<T>& y);
```

11.3.3 Die Spezialisierung `vector<bool>`

Der Standard sieht eine separate Spezialisierung `vector<bool>` der allgemeinen Vektor-Template-Klasse `vector<T>` für den Datentyp `bool` vor.

Die Funktionalität dieses Types `vector<bool>` umfasst die des allgemeinen Templates `vector<T>`, die Implementierung der Typen und Funktionen ist aber an den Datentyp `bool` angepasst und möglichst effizient.

Zusätzlich verfügt diese Klasse `vector<bool>` über eine Member-Funktion:

```
class vector<bool> {
    ...
public:
    ...
    void flip();
    ...
};
```

welche alle Vektorelemente negiert.

11.3.4 Die Containerklasse deque<T>

Eine `deque<T>` ist ein dynamisches Feld wie ein Vektor, wobei eine `deque<T>` jedoch im Gegensatz zu einem `vector<T>` nach hinten und nach vorne *effizient* wachsen kann:



(Im Prinzip kann man einen Vektor `v` durch die Funktion `v.insert(v.begin(), ...)`; auch vorne vergrößern — nur sind hierbei i. Allg. teure Umbelegungen des Speichers nötig. Eine `deque<T>` ist so implementiert, dass das Verlängern nach hinten und nach vorne effizient ist!)

Zur Verwendung von `deque<T>`'s muss die Headerdatei `<deque>` eingebunden werden! An Operationen und Funktionen stehen für eine `deque<T>` die gleichen wie für einen `vector<T>` (insbesondere Elementzugriff mittels des `[]`-Operators, der Index läuft hierbei von 0 bis `size()-1`) zur Verfügung mit folgenden Ausnahmen:

- Bei einer `deque<T>` kann man die Kapazität nicht beeinflussen und es gibt keine Funktion `capacity` zur Abfrage der Kapazität und keine Funktion `reserve` zur vorsorglichen Erhöhung der Kapazität.
- Es gibt die beiden zusätzlichen Element-Funktionen :
 - `void deque<T>::push_front(const T&);`
zum (effektiven) Einfügen der Kopie des Argumentes vom Typ `T` (gleich `value_type`) am Anfang der `deque` (neues Element hat den Index 0). Hierdurch erhalten alle "alten" Elemente der `deque<T>` einen neuen Index (um eins erhöht).
 - `void deque<T>::pop_front();`
zum (effektiven) Entfernen des ersten Elementes (das mit Index 0) einer (nicht leeren) `deque<T>`. Auch hierbei erhalten alle Elemente einen neuen Index (um eins vermindert).

Die wesentliche deque<T>-Funktionalität

Die wesentlichen Operationen einer `deque<T>` sind:

- Am Ende einfügen (Ende meint: größter Index):

```
void deque<T>::push_back(const T&);
```

- Am Ende löschen:

```
void deque<T>::pop_back();
```

- Auf Element am Ende zugreifen:

```
T& deque<T>::back();
```

(Bei einer konstanten `deque<T>` ist das Ergebnis vom Typ `const T&`!)

- Am Anfang einfügen (Anfang meint: Index 0):

```
void deque<T>::push_front(const T&);
```

- Am Anfang löschen:

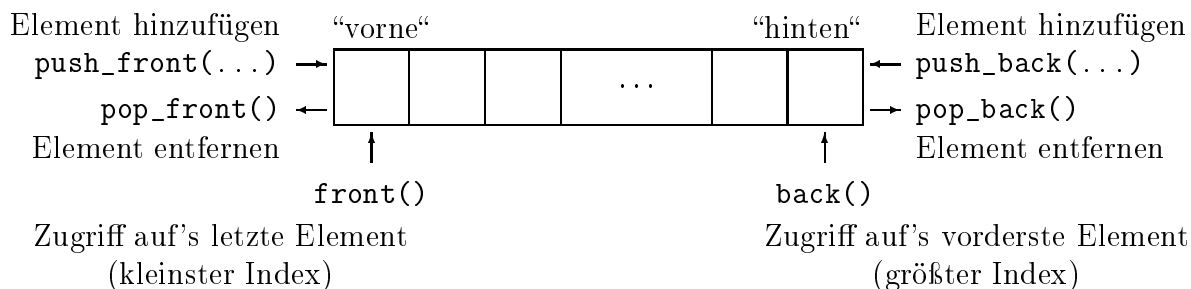
```
void deque<T>::pop_front();
```

- Auf Element am Anfang zugreifen:

```
T& deque<T>::front();
```

(Bei einer konstanten `deque<T>` ist das Ergebnis vom Typ `const T&`!)

Die (wesentliche) Funktionalität einer `deque<T>` wird in folgendem Bild veranschaulicht:



Typen der `deque<T>`-Klasse

In der Template-Klasse `deque<T>` sind (wie bei allen Containern üblich) folgende Typen definiert (vgl. Abschnitt 11.3.1):

```
template <class T>
class deque {
    ...
public:
    ...
    typedef ... reference;
    typedef ... const_reference;
    typedef ... iterator;
    typedef ... const_iterator;
```



```

typedef ... reverse_iterator;
typedef ... const_reverse_iterator;
typedef ... size_type;
typedef ... difference_type;
typedef T    value_type;
typedef ... pointer;
typedef ... const_pointer;
...
};

```

Die Bedeutung dieser Typen ist in Abschnitt 11.3.1 erläutert.

Erzeugen, Zuweisen, Zerstören einer `deque<T>`

Folgender Ausschnitt aus der Klassendefinition der Template-Klasse `deque<T>` zeigt die Möglichkeiten, ein `deque<T>`-Objekt zu erzeugen, zerstören und ihm etwas zuzuweisen:

```

template <class T>
class deque {
    ...
public:
    ...
    explicit deque();                // Standardkonstruktor
    deque( const deque<T>&);          // Copy--Konstruktor
    explicit deque ( size_type n, const T& value = T() );

    template <class InputIterator>    // mit Sequenz initialisieren
    deque( InputIterator anf, InputIterator ende);

    ~deque();                        // Destruktor

    deque<T> & operator=( const deque<T> &); // Zuweisungsoperator

    void assign( size_type n, const T& value); // Zuweisungsfunktion

    template <class InputIterator>    // Zuweisungsfunktion
    void assign( InputIterator anf, InputIterator ende);
    ...
};

```

Neben der in Abschnitt 11.3.1 erläuterten, für alle Container vorhandenen Funktionalität (Standardkonstruktor, Copy-Konstruktor, Initialisierung mit Sequenz, Destruktor und Zuweisungsoperator) gibt es somit (wie bei Vektoren) zusätzlich

- den Konstruktor:

```
explicit deque ( size_type n, const T& value = T() );
```

der eine neue `deque<T>` der Länge `n` erzeugt, wobei jedes der `n` Elemente mit dem angegebenen Wert `value` initialisiert wird. Ist kein Initialisierungswert angegeben, wird der Wert mit dem Standardkonstruktor des Types `T` erzeugt,

- die Zuweisungsfunktion

```
void assign( size_type n, const T& value);
```

welche eine vorhandene `deque<T>` mit einer `deque<T>` aus `n` Elementen mit Wert `value` überschreibt,

- der Zuweisungsfunktion

```
template <class InputIterator>
void assign( InputIterator anf, InputIterator ende);
```

welche eine vorhandene `deque<T>` mit einer `deque<T>`, dessen Elemente sich aus der durch Iteratoren gegebenen Sequenz `[anf, ende)` ergeben, überschreibt. Der Iteratortyp `InputIterator` muss zum Elementtyp des Vektors passen.

Man muss wiederum unterscheiden zwischen

- `deque<T> a(10);`
erzeugt eine `deque<T>` der Länge 10 mit 10 Standard-Elementen vom Typ `T`.
- `deque<T> a[10];`
Erzeugt ein Feld (Array) der Länge 10, wobei jedes Feldelement eine (leere) `deque<T>` vom Typ `T` ist!

Größe einer `deque<T>`

Im Gegensatz zu einem Vektor hat eine `deque<T>` keine Kapazität, sondern nur eine Größe (= Anzahl der abgespeicherten Elemente).

```
template <class T>
class deque {
    ...
public:
    ...
    size_type size() const;        // aktuelle Groesse
    size_type max_size() const;    // maximale Groesse
    bool empty() const;           // deque leer?

    // deque vergrößern/verkleinern
    void resize(size_type n, T value = T());
    ...
};
```

Neben den für alle Containerklassen üblichen Funktionen `size()`, `max_size()` und `empty` gibt es nur noch die Funktion:

```
void resize(size_type n, T value = T() );
```

welche die Größe einer `deque<T>` auf `n` ändert.

Man muss folgende Fälle unterscheiden:

1. `n` ist kleiner gleich der bisherigen Größe:

In diesem Fall bleiben die ersten `n` Elemente der `deque<T>` erhalten, die restlichen werden freigegeben (Destruktor) und die Größe des `deque<T>` wird auf `n` verringert.

2. `n` ist größer als die bisherige Größe:

Die `deque<T>` wird auf Länge `n` vergrößert, die bisherigen Elemente der `deque<T>` bleiben erhalten, die `deque<T>` wird bis zur neuen Größe `n` mit Kopien des als zweites Argument angegebenen Wertes aufgefüllt. Ist kein zweites Argument angegeben, so werden die neuen Elemente standardmäßig vorbesetzt (Default-Konstruktor).

`deque<T>`–Iteratoren

Es stehen die üblichen Iteratortypen und Funktionen zur Verfügung, welche Iteratorpositionen liefern.

```
template <class T>
class deque {
    ...
public:
    ...
    // Vorwaerts-Iterator,
    // zeigt auf erstes Element einer deque:
    iterator begin();

    // const-Vorwaerts-Iterator,
    // zeigt auf erstes Element einer const-deque:
    const_iterator begin() const;

    // Vorwaerts-Iterator,
    // zeigt hinter letztes Element einer deque:
    iterator end();

    // const-Vorwaerts-Iterator,
    // zeigt hinter letztes Element einer const-deque:
    const_iterator end() const;

    // Rueckwaerts-Iterator,
    // zeigt auf letztes Element einer deque:
    reverse_iterator rbegin();
```

```

// const-Rueckwaerts-Iterator,
// zeigt auf letztes Element einer const-deque:
const_reverse_iterator rbegin() const;

// Rueckwaerts-Iterator,
// zeigt vor erstes Element einer deque:
reverse_iterator rend();

// const-Rueckwaerts-Iterator,
// zeigt vor erstes Element einer const-deque:
const_reverse_iterator rend() const;
...
};

```

Wie bei der Klasse `vector<T>` sind auch bei der Klasse `deque<T>` die Iteratoren Random-Access-Iteratoren, so dass wiederum mittels Iteratoren wahlfreier Zugriff auf die Elemente der `deque<T>` möglich ist.

Beim wahlfreien Zugriff mittels Iteratoren auf eine `deque<T>` muss man selbst darauf achten, dass man nicht über Anfang bzw. Ende der `deque<T>` hinausläuft.

`deque<T>`–Elementzugriff

Der Zugriff auf ein einzelnes Element einer `deque<T>` ist mit folgenden Operatoren bzw. Funktionen möglich:

```

template <class T>
class deque {
    ...
public:
    ...
    // ungepruefter Index-Zugriff
    reference      operator[] (size_type n);
    const_reference operator[] (size_type n) const;

    // gepruefter Index-Zugriff
    reference      at(size_type n);
    const_reference at(size_type n) const;

    // ungepruefter Zugriff aufs erste Element (Index: 0)
    reference      front();
    const_reference front() const;

    // ungepruefter Zugriff aufs letzte Element (Index: size()-1)
    reference      back();
    const_reference back() const;
    ...
};

```

Wie bei Vektoren ist durch den Operator `[]` der Zugriff auf die Elemente einer `deque<T>` durch Indizierung möglich.

Der Index innerhalb der eckigen Klammern muss ganzzahlig sein und der Anwender muss selbst darauf achten, dass der Index im gültigen Bereich (von 0 bis `size()-1`) liegt (ungeprüfter Zugriff).

Alternativ ist im Standard (funktioniert noch nicht beim GCC-Compiler) der geprüfte Elementzugriff mittels der Element-Funktion

```
reference at(size_type n);
```

vorgesehen. Greift man mit dieser Funktion auf eine `deque<T>` vor deren Anfang (Argument < 0) oder hinter deren Ende (Argument $\geq \text{size}()$) zu, so wird eine Ausnahme vom Typ `out_of_range` ausgeworfen!

Wie in Abschnitt 11.3.1 bereits erwähnt, kann man mit den Element-Funktionen `front()` bzw. `back()` auf das erste bzw. letzte Element einer (nicht leeren) `deque<T>` zugreifen.

Ändern einer `deque<T>`

Für Objekte vom Typ `deque<T>` sind folgende Funktionen definiert, mit denen eine `deque<T>` abgeändert werden kann (Einfügen, Löschen von Elementen, zwei ganze `deque<T>`s vertauschen):

```
template <class T>
class deque {
    ...
public:
    ...
    void push_back(const T& x);
    void pop_back();

    iterator insert(iterator pos, const T& x);
    void insert(iterator pos, size_type n, const T& x);

    template <class InputIterator>
    void insert(iterator pos, InputIterator anf, InputIterator ende);

    iterator erase(iterator pos);
    iterator erase(iterator anf, iterator ende);

    void clear();
    void swap(deque<T> &);

    void push_front(const T& x);
    void pop_front();
    ...
};
```

Neben den bereits in der Klasse `vector<T>` vorhandenen Funktionen:

- `void push_back(const T&);`

hängt eine Kopie des angegebenen Argumentes vom Typ `T` “hinten” an die `deque<T>` an, d.h. die `deque<T>`-Größe wird um eins größer und das neue Element ist das mit dem größten Index.

- `void pop_back();`

entfernt das “hinterste” Element (das mit dem größten Index) aus der `deque<T>` (darf nicht leer sein!) und verringert die Größe um eins.

- `iterator insert(iterator pos, value_type x);`

fügt eine Kopie des angegebenen Elementes an die Position `pos`. Ergebnis ist die Iteratorposition auf das eingefügte Element.

- `void insert(iterator pos, size_type n, const T& x);`

fügt an der durch die Iteratorposition `pos` gegebenen Stelle `n` Kopien des angegebenen Elementes `x` ein.

- `template <class InputIterator>`

`void insert(iterator pos, InputIterator anf, InputIterator end);`

fügt in die `deque<T>` an der Stelle `pos` Kopien der durch die Iteratoren `anf` und `end` gegebenen Sequenz ein. Der Iteratortyp `InputIterator` muss zum Elementtyp der `deque<T>` passen.

- `iterator erase(iterator pos);`

Löschen des Elementes an Position `pos`. Funktionsergebnis ist die Position des `deque<T>`-Elementes hinter dem gelöschten.

- `iterator erase(iterator anf, iterator end);`

Löschen aller Elemente der durch `anf` und `end` gegebenen Teil-Sequenz der `deque<T>`. Ergebnis ist die Iteratorposition hinter das zuletzt gelöschte Element der ursprünglichen `deque<T>`.

- `void clear();`

löscht alle Elemente der `deque<T>`, diese hat anschließend die Größe 0.

- `void swap(deque<T>&);`

vertauscht den Inhalt zweier `deque<T>`'s.

gibt es zusätzlich die beiden Elementfunktionen:

- `void push_front(const T&);`

hängt eine Kopie des angegebenen Argumentes vom Typ `T` “vorne” (Index 0) an die `deque<T>` an, d.h. die `deque<T>`-Größe wird um eins größer und das neue Element ist das mit dem Index 0.

– `void pop_front();`

entfernt das “vorne“ Element (das mit dem kleinsten Index) aus der `deque<T>` (darf nicht leer sein!) und verringert die Größe um eins.

Einfügen/Entfernen von Elementen vorne und hinten sind bei einer `deque<T>` sehr effizient (konstanter Zeitaufwand) — mitten in einer `deque<T>` sind diese Einfüge- und Löschoperationen wiederum ineffizient.

Der Anwender obiger Funktionen ist selbst dafür verantwortlich, dass die verwendeten Iteratoren gültig sind!

Globale Operatoren und Funktionen für eine `deque<T>`

Es sind die üblichen Vergleichs-Operatoren als globale Operator-Funktionen:

```
template <class T>
bool operator== (const deque<T>& x, const deque<T>& y);
```

```
template <class T>
bool operator< (const deque<T>& x, const deque<T>& y);
```

```
template <class T>
bool operator!= (const deque<T>& x, const deque<T>& y);
```

```
template <class T>
bool operator> (const deque<T>& x, const deque<T>& y);
```

```
template <class T>
bool operator>= (const deque<T>& x, const deque<T>& y);
```

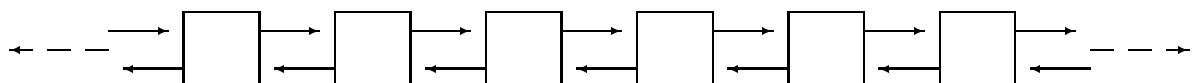
```
template <class T>
bool operator<= (const deque<T>& x, const deque<T>& y);
```

sowie eine (möglicherweise sehr effizient implementierte) globale Funktion zum Vertauschen zweier `deque<T>`’s vorhanden:

```
template <class T>
void swap( deque<T>& x, deque<T>& y);
```

11.3.5 Die Containerklasse `list<T>`

Der Datentyp `list<T>` stellt eine doppelt verkettete Liste von Objekten vom Typ `T` dar, d.h. jedes Listenelement hat neben seinem eigentlichen Inhalt (Typ `T`) einen Verweis auf seinen Vorgänger und einen auf seinen Nachfolger:



Zur Verwendung solcher Listen muss die Headerdatei `<list>` includet werden!

Die interne Realisierung und Verwaltung einer `list` ist vollkommen anders als die eines `vector<T>` oder einer `deque<T>`:

- Es besteht **kein** wahlfreier Zugriff auf die einzelne Elemente, um etwa das dritte Element aufzusuchen muss man sich vorher die ersten beiden Elemente “entlang-hangeln”. Entsprechend gibt es keine Indizierung und die Iteratoren sind keine *Random-Access-Iteratoren*, sondern “nur” noch *Bidirektionale Iteratoren*, (siehe Seite 393).
- Das Löschen und Einfügen von einem oder mehreren Elementen an beliebiger Stelle ist effektiv. Darüberhinaus bleiben bei Einfügen (bzw. Löschen) von Elementen die Adressen der übrigen Elemente sowie Iteratoren gültig!
- Es gibt eine Reihe von speziellen Funktionen, welche die typische Verwendung von Linearen Listen ermöglichen (siehe nächsten Unterabschnitt!).

Die wesentlichen Listenoperationen

Eine zunächst leere Lineare Liste (mit Elementtyp `T` und Namen `liste`) wird durch `list<T> liste;`

erzeugt. (Natürlich kann auch mittels Copy-Konstruktor eine neue `list` als Kopie einer anderen erzeugt werden!)

Man kann (wie bei allen Containerklassen der Standardbibliothek) mit der Elementfunktion `liste.size()` die Anzahl der in der Liste abgespeicherten Elemente ermitteln und mit `liste.empty()` erfahren, ob die Liste leer ist.

Einfügen, Löschen und Zugreifen sind “vorne” und “hinten” möglich (`elem` sei jeweils ein Objekt vom Typ `T`):

- `liste.push_front(elem);`
fügt “vorne” in der Liste eine Kopie von `elem` ein.
- `liste.pop_front(elem);`
entfernt “vorne” ein Element (Liste darf nicht leer sein!).
- `liste.front(elem);`
liefert Referenz auf das “vorderste” Element der Liste (die Liste darf nicht leer sein!). Bei einer konstanten Liste (möglich, wenn etwa eine an sich variable Liste als konstante Referenz an eine Funktion übergeben wird) ist diese Referenz auf das “vorderste” Element eine konstante Referenz, so dass das Element über `front()` nicht abgeändert werden kann!
- `liste.push_back(elem);`
fügt “hinten” in der Liste eine Kopie von `elem` ein.
- `liste.pop_back(elem);`
entfernt “hinten” ein Element (Liste darf nicht leer sein!).

- `liste.back(elem);`
liefert Referenz auf das “hinterste” Element der Liste (die Liste darf nicht leer sein!). Bei einer konstanten Liste (möglich, wenn etwa eine an sich variable Liste als konstante Referenz an eine Funktion übergeben wird) ist diese Referenz auf das “hinterste” Element eine konstante Referenz, so dass das Element über `back()` nicht abgeändert werden kann!

Mittels eines Iterators kann man wie üblich auf beliebige Elemente der Liste zugreifen, etwa alle Elemente der Liste durchlaufen:

```
void fkt( list<int> &liste)
{
    list<int>::iterator iter;    // passenden Iterator definieren
    ...
    for ( iter = liste.begin(); iter != liste.end(); ++iter)
    {
        // auf aktuelles Listenelement mittels *iter zugreifen
        ...
    }
    ...
}
```

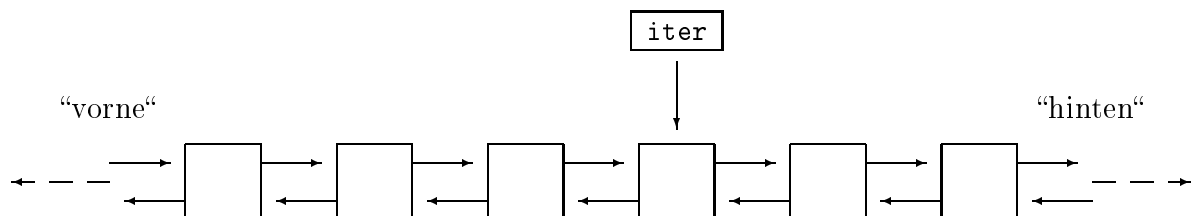
Zeigt der Iterator `iter` auf ein (mittleres) Element der Liste, so zeigt er nach `++iter` auf das nächste bzw. nach `--iter` auf das vorherige (d.h. für diese Iteratoren ist zusätzlich der `---` Operator definiert, der den Iterator um eine Position nach “hinten” verschiebt!).

Ist `iter` eine Iteratorposition in einer Liste `liste<T>` (zwischen `liste.begin()` und `liste.end()` — beide eingeschlossen!), so kann man mit

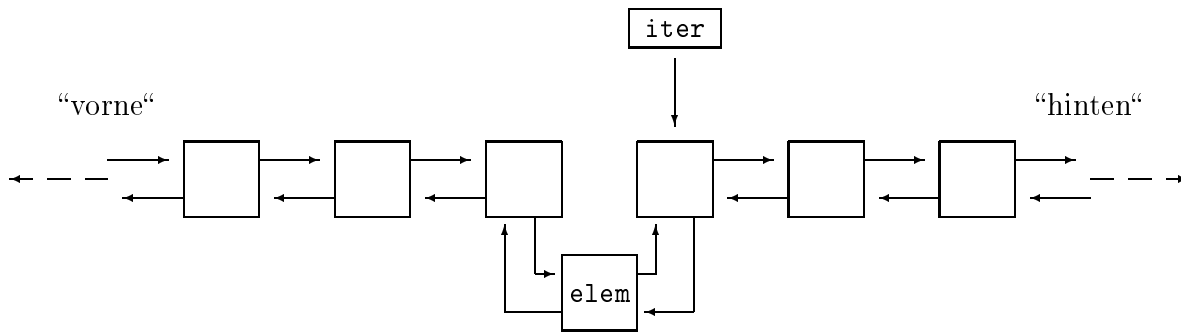
`liste.insert(iter, elem);`

eine Kopie von `elem` (vom Typ `T`) in die Liste einfügen — das neue Element wird hierbei jeweils vor das “Listenelement“, auf welches der Iterator `iter` zeigt, eingefügt:

Liste vor dem Aufruf von `liste.insert(iter, elem)`:



Liste nach dem Aufruf von `liste.insert(iter, elem)`:



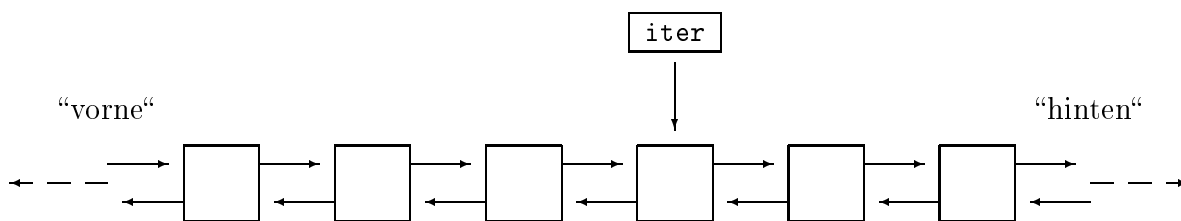
Zeigt `iter` auf den Listenanfang (`iter == liste.begin()`), so wird das neue Element am Listenanfang ("vorne") eingefügt (entspricht `liste.push_front(...)`), zeigt `iter` hinter das Ende der Liste (`iter == liste.end()`), so wird das neue Element am Listende ("hinten") angehängt (entspricht `liste.push_back(...)`).

Ist `iter` eine Iteratorposition in einer (nicht leeren!) Liste `list<T> liste` (zwischen `liste.begin()` einschließlich und `liste.end()` ausschließlich), so wird durch die Funktion:

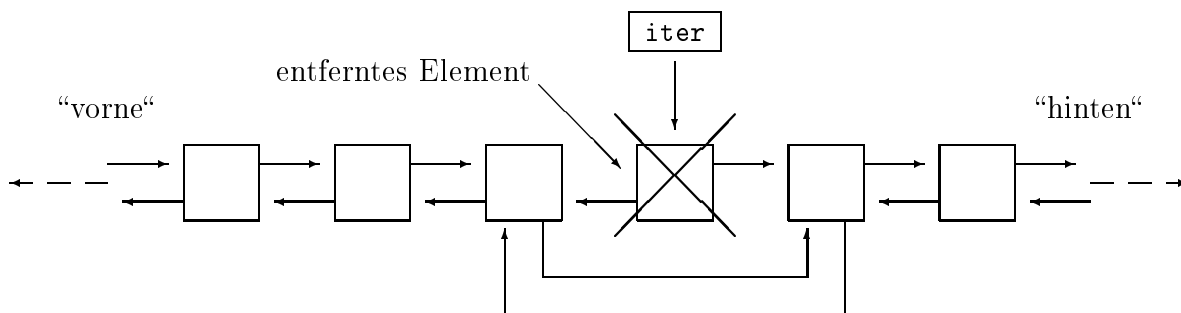
```
liste.erase(iter)
```

das Listenelement, auf welches der Iterator verweist, aus der Liste entfernt:

Liste vor dem Aufruf von `liste.erase(iter)`:



Liste nach dem Aufruf von `liste.erase(iter)`:



(Nach diesem Aufruf der Funktion `erase` "zeigt" der Iterator `iter` somit nicht mehr auf ein Element der aktuellen Liste!)

Die Standardbibliothek stellt eine Reihe von weiteren, typischen Listenoperationen zur Containerklasse `list<T>` zur Verfügung. Diese werden auf Seite 397 erläutert.

Typen der `list<T>`-Klasse

In der Template-Klasse `list<T>` sind (wie bei allen Containern üblich) folgende Typen definiert (vgl. Abschnitt 11.3.1):

```
template <class T>
class list {
    ...
public:
    ...
    typedef ... reference;
    typedef ... const_reference;
    typedef ... iterator;
    typedef ... const_iterator;
    typedef ... reverse_iterator;
    typedef ... const_reverse_iterator;
    typedef ... size_type;
    typedef ... difference_type;
    typedef T    value_type;
    typedef ... pointer;
    typedef ... const_pointer;
    ...
};
```

Die Bedeutung dieser Typen ist in Abschnitt 11.3.1 erläutert.

Erzeugen, Zuweisen, Zerstören einer `list<T>`

Folgender Ausschnitt aus der Klassendefinition der Template-Klasse `list<T>` zeigt die Möglichkeiten, ein `list<T>`-Objekt zu erzeugen, zerstören und ihm etwas zuzuweisen:

```
template <class T>
class list {
    ...
public:
    ...
    explicit list();                // Standardkonstruktor
    list( const list<T>&);          // Copy--Konstruktor
    explicit list ( size_type n, const T& value = T() );

    template <class InputIterator> // mit Sequenz initialisieren
    list( InputIterator anf, InputIterator ende);

    ~list();                        // Destruktor

    list<T> & operator=( const list<T> &); // Zuweisungsoperator
```

```

void assign( size_type n, const T& value); // Zuweisungsfunktion

template <class InputIterator>           // Zuweisungsfunktion
void assign( InputIterator anf, InputIterator ende);
...
};

```

Neben der in Abschnitt 11.3.1 erläuterten, für alle Container vorhandenen Funktionalität (Standardkonstruktor, Copy-Konstruktor, Initialisierung mit Sequenz, Destruktor und Zuweisungsoperator)

gibt es somit (wie bei Vektoren und `deque<T>`'s) zusätzlich

- den Konstruktor:

```
explicit list ( size_type n, const T& value = T() );
```

der eine neue Liste der Länge `n` erzeugt, wobei jedes der `n` Elemente mit dem angegebenen Wert `value` initialisiert wird. Ist kein Initialisierungswert angegeben, wird der Wert mit dem Standardkonstruktor des Types `T` erzeugt,

- die Zuweisungsfunktion

```
void assign( size_type n, const T& value);
```

welche eine vorhandene Liste mit einer Liste aus `n` Elementen mit Wert `value` überschreibt,

- der Zuweisungsfunktion

```
template <class InputIterator>
void assign( InputIterator anf, InputIterator ende);
```

welche eine vorhandene Liste mit einer Liste, dessen Elemente sich aus der durch Iteratoren gegebenen Sequenz `[anf, ende)` ergeben, überschreibt. Der Iteratortyp `InputIterator` muss zum Elementtyp der Liste passen.

Größe einer Liste

Die Funktionen zur Größe einer Liste sind die gleichen wie bei einer `deque<T>`:

```

template <class T>
class list {
...
public:
...
    size_type size() const;           // aktuelle Groesse

```

```

    size_type max_size() const;    // maximale Groesse
    bool empty() const;           // Liste leer?

    // Liste vergroessern/verkleinern:
    void resize(size_type n, T value = T());
    ...
};

```

Neben den für alle Containerklassen üblichen Funktionen `size()`, `max_size()` und `empty` gibt es (wie bei `deque<T>`) nur noch die Funktion:

```
void resize(size_type n, T value = T() );
```

welche die Größe einer Liste auf `n` ändert.

Man muss folgende Fälle unterscheiden:

1. `n` ist kleiner gleich der bisherigen Größe:
In diesem Fall bleiben die ersten `n` Elemente der Liste erhalten, die restlichen werden freigegeben (Destruktor) und die Größe der Liste wird auf `n` verringert.
2. `n` ist größer als die bisherige Größe:
Die Liste wird auf Länge `n` vergrößert, die bisherigen Elemente bleiben erhalten, an die bisherige Liste werden bis zur neuen Größe `n` Kopien des als zweites Argument angegebenen Wertes `value` angehängt. Ist kein zweites Argument angegeben, so werden die neuen Elemente standardmäßig vorbesetzt (Default-Konstruktor).

Listen-Iteratoren

Es stehen die üblichen Iteratortypen und Funktionen zur Verfügung, welche Iteratorpositionen liefern.

```

template <class T>
class list {
    ...
public:
    ...
    // Vorwaerts-Iterator,
    // zeigt auf erstes Element einer Liste:
    iterator begin();

    // const-Vorwaerts-Iterator,
    // zeigt auf erstes Element einer const-Liste:
    const_iterator begin() const;

    // Vorwaerts-Iterator hinter,
    // zeigt letztes Element einer Liste:
    iterator end();

    // const-Vorwaerts-Iterator,

```

```

// zeigt hinter letztes Element einer const-Liste:
const_iterator end() const;

// Rueckwaerts-Iterator,
// zeigt auf letztes Element einer Liste:
reverse_iterator rbegin();

// const-Rueckwaerts-Iterator,
// zeigt auf letztes Element einer const-Liste:
const_reverse_iterator rbegin() const;

// Rueckwaerts-Iterator,
// zeigt vor erstes Element einer Liste:
reverse_iterator rend();

// const-Rueckwaerts-Iterator,
// zeigt vor erstes Element einer const-Liste:
const_reverse_iterator rend() const;
...
};

```

Im Gegensatz zur Klasse `vector<T>` und `deque<T>` sind die Listen-Iteratoren “nur“ Bidirektionale Iteratoren, d.h. neben den für alle Iteratoren verfügbaren Operationen:

<code>iter1 == iter2</code>	Gleichheit zweier Iteratoren, genau dann wahr, wenn beide Iteratoren auf dasselbe Element desselben Containers zeigen.
<code>iter1 != iter2</code>	Ungleichheit zweier Iteratoren <code>!(iter1 == iter2)</code> .
<code>*iter</code>	Zugriff auf das aktuelle Element.
<code>iter->komponente</code>	Zugriff auf eine Komponente des Aktuellen Elementes.
<code>++iter</code>	Weitersetzen des Iterators, liefert neue Position.
<code>iter++</code>	Weitersetzen des Iterators, liefert alte Position.
<code>iterator iter2(iter1)</code>	Copy-Konstruktor.

ist zusätzlich die Anwendung folgender Operatoren auf solche Iteratoren möglich (vgl. Abschnitt 11.2):

<code>--iter</code>	Zurücksetzen des Iterators um eine Position, liefert neue Position.
<code>iter--</code>	Zurücksetzen des Iterators um eine Position, liefert alte Position.
<code>iter1 = iter2</code>	Zuweisung von Iteratoren.

Elementzugriff bei Listen

Neben dem Zugriff auf Listenelemente mittels Iteratoren sind nur folgende Zugriffsfunktionen vorhanden:

```

template <class T>
class list {

```

```

...
public:
    ...
    reference      front();          // erstes Element (hinten!)
    const_reference front() const;    // erstes Element (hinten!)
    reference      back();           // letztes Element (vorne!)
    const_reference back();           // letztes Element (vorne!)
    ...
};

```

Ändern einer Liste

Zur Abänderung der Elemente einer Liste stehen die gleichen Funktionen wie bei einer `deque<T>` zur Verfügung. Im Gegensatz zu `deque<T>`'s sind diese Funktionen für Listen jedoch alle effizient:

```

template <class T>
class list {
    ...
public:
    ...
    void push_back(const T& x);
    void pop_back();
    void push_front(const T& x);
    void pop_front();

    iterator insert(iterator pos, const T& x);
    void insert(iterator pos, size_type n, const T& x);

    template <class InputIterator>
    void insert(iterator pos, InputIterator anf, InputIterator ende);

    iterator erase(iterator pos);
    iterator erase(iterator anf, iterator ende);

    void clear();
    void swap(list<T> &);
    ...
};

```

Erläuterung zu diesen Funktionen:

– `void push_back(const T&);`

hängt eine Kopie des angegebenen Argumentes vom Typ `T` “hinten“ (an der Iteratorposition `end()`) an die Liste an, d.h. die Liste wird um eins länger.

– `void pop_back();`

entfernt das “hinterste“ Element (vor der Iteratorposition `end()`) aus der Liste (darf nicht leer sein!) und verringert die Länge (`size()`) um eins.

– `void push_front(const T&);`

hängt eine Kopie des angegebenen Argumentes vom Typ `T` “vorne“ (vor Iteratorposition `begin()`) an die Liste an.

– `void pop_front();`

entfernt das “vorne“ Element (an Iteratorposition `begin()`) aus der Liste (darf nicht leer sein!).

– `iterator insert(iterator pos, const T& x);`

fügt eine Kopie des angegebenen Elementes an die Position `pos` in die Liste ein. Ergebnis ist die Iteratorposition auf das eingefügte Element.

– `void insert(iterator pos, size_type n, const T& x);`

fügt in die Liste vor der durch die Iteratorposition `pos` gegebenen Stelle `n` Kopien des angegebenen Elementes `x` ein.

– `template <class InputIterator>`

`void insert(iterator pos, InputIterator anf, InputIterator end);`

fügt in die Liste vor der Stelle `pos` Kopien der durch die Iteratoren `anf` und `end` gegebenen Sequenz ein. Der Iteratortyp `InputIterator` muss zum Elementtyp der Liste passen.

– `iterator erase(iterator pos);`

Löschen des Elementes an Position `pos`. Funktionsergebnis ist die Position des Listen-Elementes hinter dem gelöschten.

– `iterator erase(iterator anf, iterator end);`

Löschen aller Elemente der durch `anf` und `end` gegebenen Teil-Sequenz der Liste. Ergebnis ist die Iteratorposition hinter das zuletzt gelöschte Element der ursprünglichen Liste.

– `void clear();`

löscht alle Elemente der Liste, diese hat anschließend die Länge 0.

– `void swap(list<T>&);`

vertauscht den Inhalt zweier Listen.

Der Anwender ist selbst dafür verantwortlich, dass die bei diesen Funktionen verwendeten Iteratoren gültig sind!

Typische Listenoperationen

Es stehen zusätzlich folgende typischen Listenoperationen zur Verfügung:

```
template <class T>
class list {
    ...
public:
    ...
    void splice ( iterator pos, list<T>& x);
    void splice ( iterator pos, list<T>& x, iterator posx);
    void splice ( iterator pos, list<T>& x, iterator anf,
                                     iterator ende);

    void remove(const T& value);

    template <class UnPred>
    void remove_if(UnPred pred);

    void unique();

    template <class BinPred>
    void unique( Binpred pred);

    void merge( list<T> &x);

    template <class Compare>
    void merge( list<T> &x, Compare comp);

    void sort();

    template <class Compare>
    void sort(Compare comp);

    void reverse();
    ...
};
```

Erläuterungen zu diesen Funktionen:

- `void splice (iterator pos, list<T>& x);`
 fügt in die aktuelle Liste vor Position `pos` die Elemente der als zweites Argument angegebenen Liste `x` ein. Die Liste `x` ist anschließend leer! Aktuelle Liste und Liste `x` müssen verschieden sein!
- `void splice (iterator pos, list<T>& x, iterator posx);`
 fügt in die aktuelle Liste vor Position `pos` das an Position `posx` stehende Element der als zweites Argument angegebenen Liste `x` ein. Das in der aktuellen Liste

eingefügte Element wird aus der zweiten Liste **x** entfernt. Aktuelle Liste und Liste **x** dürfen übereinstimmen.

```
- void splice ( iterator pos, list<T>& x, iterator anf,
               iterator ende);
```

fügt in die aktuelle Liste vor Position **pos** die durch die Iteratorpositionen **anf** und **ende** gegebene Teilliste der als zweites Argument angegebenen Liste **x** ein. Die in der aktuellen Liste eingefügten Elemente werden aus der zweiten Liste **x** entfernt.

Aktuelle Liste und Liste **x** dürfen übereinstimmen, in diesem Fall darf die Position **pos** nicht innerhalb der Teilliste [**anf**, **ende**) sein.

```
- void remove(const T& value);
```

löscht in der Liste alle Elemente, welche den angegebenen Wert **value** haben.

```
- template <class UnPred>
void remove_if(UnPred pred);
```

(**pred** ist hierbei ein unäres Prädikat mit Argumenttyp **T**, d.h. ein Funktionsobjekt, welches ein Argument vom Typ **T** und ein Ergebnis vom Typ **bool** hat, vgl. Abschnitt 11.6.3)

löscht in der Liste alle Elemente, welche in das unäre Prädikat **pred** eingesetzt den Wert **true** liefern.

```
- void unique();
```

zu jedem Element der Liste werden unmittelbar folgende, bzgl. des Operators **==** gleichwertigen Elemente gelöscht. (Bei einer sortierten Liste werden somit alle Duplikate entfernt!)

```
- template <class BinPred>
void unique_if(BinPred pred);
```

(**pred** ist hierbei ein binäres Prädikat mit zweifachem Argumenttyp **T**, d.h. ein Funktionsobjekt, welches zwei Argumente vom Typ **T** und ein Ergebnis vom Typ **bool** hat, vgl. Abschnitt 11.6.3)

zu jedem Element (**elem1**) der Liste werden unmittelbar folgende, bzgl. des Prädikates **pred** gleichwertigen Elemente (**elem2**) entfernt (d.h. der "Nachfolger" **elem2** von **elem1** wird gelöscht, falls **pred(elem1, elem2)** gleich **true** ist).

```
- void sort();
```

sortiert die Liste anhand des Vergleichsoperators **<**, d.h. steht nach dem Sortieren ein Element **elem2** (nicht unbedingt unmittelbar) "hinter" einem anderen Element **elem1**, so ist der Vergleich **elem2 < elem1** stets falsch!

Bezüglich **<** gleichwertige Elemente **elem1** und **elem2** (d.h. **elem1 < elem2** und **elem2 < elem1** sind jeweils falsch) bleiben in ihrer ursprünglichen Reihenfolge erhalten. (Stabiles Sortieren!)

```
– template <class Compare>
  void sort(Compare comp);
```

(hierbei ist `Compare` ein Vergleichsfunktionsobjekttyp und `comp` ein Objekt dieses Types, also ein binäres Prädikat mit doppeltem Argumenttyp `T` und Ergebnistyp `bool`, welches zusätzliche Bedingungen erfüllt — vgl. Abschnitt 11.6.3)

sortiert die Liste anhand des Vergleichs `comp` d.h. steht nach dem Sortieren ein Element `elem2` (nicht unbedingt unmittelbar) “hinter“ einem anderen Element `elem1`, so ist der Vergleich `comp(elem2, elem1)` stets falsch!

Bezüglich des Vergleichs gleichwertige Elemente `elem1` und `elem2` (d.h. die beiden Vergleiche `comp(elem1, elem2)` und `comp(elem2, elem1)` sind jeweils falsch) bleiben in ihrer ursprünglichen Reihenfolge erhalten. (Stabiles Sortieren!)

```
– void merge( list<T> &x );
```

Diese Funktion setzt voraus, dass die aktuelle und die als Argument angegebene Liste `x` bzgl. des Operators `<` aufsteigend sortiert sind.

`merge` mischt die als Argument angegebene (sortierte) Liste `x` in die aktuelle (sortierte) Liste ein, d.h. die Elemente von `x` werden so in die aktuelle Liste eingefügt, dass die aktuelle Liste anschließend immer noch (bzgl. `<`) aufsteigend sortiert ist.

Die Liste `x` ist anschließend leer. Aktuelle Liste und Liste `x` dürfen nicht übereinstimmen.

```
– template <class Compare>
  void merge( list<T> &x, Compare comp);
```

(hierbei ist `Compare` ein Vergleichsfunktionsobjekttyp und `comp` ein Objekt dieses Types, also ein binäres Prädikat mit doppeltem Argumenttyp `T` und Ergebnistyp `bool`, welches zusätzliche Bedingungen erfüllt — vgl. Abschnitt 11.6.3)

Diese Funktion setzt voraus, dass die aktuelle und die als Argument angegebene Liste bzgl. des Vergleichs `comp` aufsteigend sortiert sind.

`merge` mischt die als Argument angegebene (sortierte) Liste `x` in die aktuelle (sortierte) Liste ein, d.h. die Elemente von `x` werden so in die aktuelle Liste eingefügt, dass die aktuelle Liste anschließend immer noch (bzgl. `comp`) aufsteigend sortiert ist.

Die Liste `x` ist anschließend leer. Aktuelle Liste und Liste `x` dürfen nicht übereinstimmen.

Globale Operatoren und Funktionen für Listen

Es sind die üblichen Vergleichs-Operatoren als globale Operator-Funktionen:

```

template <class T>
bool operator== (const list<T>& x, const list<T>& y);

template <class T>
bool operator< (const list<T>& x, const list<T>& y);

template <class T>
bool operator!= (const list<T>& x, const list<T>& y);

template <class T>
bool operator> (const list<T>& x, const list<T>& y);

template <class T>
bool operator>= (const list<T>& x, const list<T>& y);

template <class T>
bool operator<= (const list<T>& x, const list<T>& y);

sowie eine (möglicherweise sehr effizient implementierte) globale Funktion zum Ver-
tauschen zweier Listen vorhanden:

template <class T>
void swap( list<T>& x, list<T>& y);

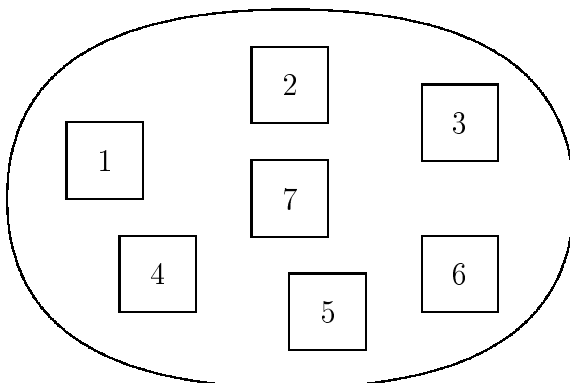
```

11.3.6 Die Containerklassen `set<T>` und `multiset<T>`

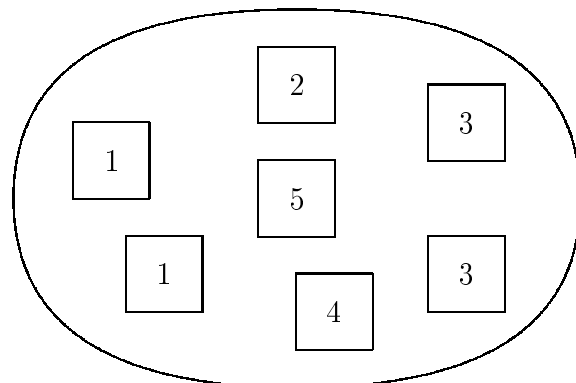
Eine `set<T>` bzw. `multiset<T>` sind Mengenklassen zur Verwaltung von Objekten vom Typ `T` und daraufhin optimiert, Elemente aufzufinden. (Bei einem `vector<T>` und einer `deque<T>` ist der wahlfreie Zugriff — bei einer `list<T>` das Einfügen und Entfernen von Elementen an beliebiger Stelle die Zielrichtung der Implementierung!) Zur Verwendung von `set<T>`'s bzw. `multiset<T>`'s muss die Headerdatei `<set>` includet werden.

Der Unterschied zwischen einer `set<T>` und einer `multiset<T>` ist der, dass in einer `multiset<T>` mehrere Elemente mit gleichem Wert vorhanden sein können, in einer `set<T>` müssen alle Elemente verschiedene Werte haben:

`set<int>`:



`multiset<int>`:



Zur Erzielung der “schnellen Auffindbarkeit“ von Elementen in einer Menge muss standardmäßig der Vergleichsoperator `<` oder ein anderes Vergleichsfunktionsobjekt (siehe etwa Abschnitt 11.7.5) für Objekte des beteiligten Types `T` definiert sein.

Bei der Erzeugung einer `set<T>` oder `multiset<T>` muss (wie üblich) der Elementtyp `T` angegeben werden, zusätzlich kann als zweites Template-Argument der zu Grunde liegende Vergleich als Funktionsobjekttyp angegeben sein — die `set<T>`- und `multiset<T>`-Templates haben also zwei Typparameter! (Auf die Angabe dieses zweiten Template-Argumentes wird in der abgekürzten Schreibweise `multiset<T>` bzw. `set<T>` verzichtet, eigentlich müsste `set<T, Compare>` bzw. `multiset<T, Compare>` geschrieben werden!)

Wird kein Funktionsobjekttyp angegeben, so wird mittels des Operators `<` verglichen (d.h. der Funktionsobjekttyp ist dann standardmäßig der Typ `less<T>`, der zum Operator `<` des Types `T` gehörende Funktionsobjekttyp).

Darüberhinaus werden die Elemente intern in spezieller Art- und Weise (i. Allg. in einer Art ausbalancierter Bäume) abgespeichert, so dass die geforderte “schnelle Auffindbarkeit“ realisiert werden kann.

Typ	Bedeutung
<code>set<T></code>	Menge von (unterschiedlichen) Elementen vom Typ <code>T</code> , Vergleich mittels <code><</code> .
<code>multiset<T></code>	Menge von (auch gleichen) Elementen vom Typ <code>T</code> , Vergleich mittels <code><</code> .

Möchte man etwa für einen Typen `T` einen eigenen Vergleich zu Grunde legen, so muss man hierzu einen “Funktionsobjekttypen“ (Standard-Funktionsobjekte werden in Abschnitt 11.6 ausführlicher behandelt) definieren, etwa:

```
/* von der zum Standard gehoerenden Template-Klasse  binary-function
   abgeleitete Klasse  vergleich,
   zu der der Funktionsaufruf-Operator () ueberladen ist!
```

```

    T ist der zur Elementtyp der set bzw. multiset
*/
```

```
struct vergleich: public binary_function<T,T,bool>
{ public:
    bool operator() ( T a, T b)
    {
        /* hier muss irgendwie der Vergleich der Elemente a und b
           erfolgen und ein Wahrheitswert zurueckgegeben werden
        */
        return ...;
    }
};
```

und diesen Funktionsobjekttypen als zweites Template-Argument beim Mengentyp angeben:

Typ	Bedeutung
<code>set<T, vergleich></code>	Menge von (unterschiedlichen) Elementen vom Typ <code>T</code> , Vergleich mittels eines Objektes des Funktionstypen <code>vergleich</code> .
<code>multiset<T, vergleich></code>	Menge von (möglicherweise gleichen) Elementen vom Typ <code>T</code> , Vergleich mittels eines Objektes des Funktionstypen <code>vergleich</code> .

Die “schnelle Auffindbarkeit“ hat zur Konsequenz, dass die Stelle, an der (intern) ein Element in der Menge abgespeichert wird, durch den Wert des Elementes bestimmt wird. Deshalb können die Elemente einer Menge nicht abgeändert (höchstens entfernt und mit neuem Wert neu eingefügt) werden.

Entsprechend liefert der Zugriff auf Mengenelemente über einen Iterator den Typ `const T&`, so dass auch hier keine Änderung des Elementwertes möglich ist!

Typen der `set<T>`– bzw. `multiset<T>`–Klasse

In den Template-Klassen `set<T>` bzw. `multiset<T>` sind neben den in allen Containerklassen definierten Hilfstypen drei weitere definiert, insgesamt sind es folgende Typen definiert (vgl. Abschnitt 11.3.1):

```
template <class T, class Compare = less<T> >
class set {
    ...
public:
    ...
    // wie in allen Containerklassen:
    typedef ... reference;
    typedef ... const_reference;
    typedef ... iterator;
    typedef ... const_iterator;
    typedef ... reverse_iterator;
    typedef ... const_reverse_iterator;
    typedef ... size_type;
    typedef ... difference_type;
    typedef T    value_type;
    typedef ... pointer;
    typedef ... const_pointer;

    // zusaetzlich:
    typedef T    key_type;
    typedef Compare key_compare;
    typedef Compare value_compare;
    ...
};

template <class T, class Compare = less<T> >
class multiset {
```

```

...
public:
    ...
    // wie in allen Containerklassen:
    typedef ... reference;
    typedef ... const_reference;
    typedef ... iterator;
    typedef ... const_iterator;
    typedef ... reverse_iterator;
    typedef ... const_reverse_iterator;
    typedef ... size_type;
    typedef ... difference_type;
    typedef T    value_type;
    typedef ... pointer;
    typedef ... const_pointer;

    // zusaetzlich:
    typedef T      key_type;
    typedef Compare key_compare;
    typedef Compare value_compare;
    ...
};

```

Der Typ `key_type` ist ein anderer Name für den `value_type`, also den Typen `T` und `value_compare` und `key_compare` sind andere Namen für den dem Vergleich zu Grunde liegenden Funktionsobjekttypen. Die Bedeutung der anderen Typen ist in Abschnitt 11.3.1 erläutert.

Erzeugen, Zuweisen, Zerstören einer `set<T>` bzw. `multiset<T>`

Bei den Konstruktoren zur Klasse `set<T>` und `multiset<T>` (vom Copy-Konstruktor abgesehen) kann gegenüber den bisherigen Containerklassen als zusätzliches Argument ein konkretes Vergleichs-Funktionsobjekt (also ein Objekt des als Template-Parameter angegebenen Funktionsobjekttypes) angegeben werden (dieses wird wohl irgendwie im `private`- oder `protected`-Teil abgespeichert). Standardmäßig wird hier das mit dem zugehörigen Standardkonstruktor erzeugte Objekt des Funktionsobjekttypes genommen!

```

template <class T, class Compare = less<T> >
class set {
    ...
public:
    ...
    // Standardkonstruktor
    explicit set(const Compare& cmp = Compare() );

    // Copy--Konstruktor

```

```

    set( const set<T, Compare>&);

    // mit Sequenz initialisieren
    template <class InputIterator>
    set(InputIterator anf, InputIterator ende,
        const Compare& cmp = Compare());

    // Destruktor
    ~set();

    // Zuweisungsoperator
    set<T,Compare> & operator=(const set<T,Compare> &);
    ...
};

template <class T, class Compare = less<T> >
class multiset {
    ...
public:
    ...
    // Standardkonstruktor
    explicit multiset(const Compare& cmp = Compare() );

    // Copy--Konstruktor
    multiset( const multiset<T, Compare>&);

    // mit Sequenz initialisieren
    template <class InputIterator>
    multiset(InputIterator anf, InputIterator ende,
        const Compare& cmp = Compare());

    // Destruktor
    ~multiset();

    // Zuweisungsoperator
    multiset<T,Compare> & operator=(const multiset<T,Compare> &);
    ...
};

```

Neben den bei allen Containerklassen vorhandenen Konstruktoren und Zuweisungsoperator gibt es keine weiteren Konstruktoren oder Zuweisungsfunktionen.

Größe einer `set<T>` bzw. `multiset<T>`

Bei `set<T>`'s und `multiset<T>`'s kann nur die aktuelle Elementzahl, maximale Elementzahl und "Leerheit" abgefragt werden:


```

template <class T, class Compare = less<T> >
class set {
    ...
public:
    ...
    size_type size() const;        // aktuelle Elementzahl
    size_type max_size() const;    // maximale Elementzahl
    bool empty() const;           // Menge leer?
    ...
};

template <class T, class Compare = less<T> >
class multiset {
    ...
public:
    ...
    size_type size() const;        // aktuelle Elementzahl
    size_type max_size() const;    // maximale Elementzahl
    bool empty() const;           // Multimenge leer?
    ...
};

```

Vergrößern einer Menge oder Multimenge ist nur durch explizites Einfügen eines neuen Elementes möglich (es gibt keine Funktion `resize()`)!

Iteratoren für `set<T>`'s und `multiset<T>`'s

Es stehen die üblichen Iteratortypen und Funktionen zur Verfügung, welche Iteratorpositionen liefern.

```

template <class T, class Compare = less<T> >
class set {
    ...
public:
    ...
    // Vorwaerts-Iterator,
    // zeigt auf erstes Element einer Menge:
    iterator begin();

    // const-Vorwaerts-Iterator,
    // zeigt auf erstes Element einer const-Menge:
    const_iterator begin() const;

    // Vorwaerts-Iterator,
    // zeigt hinter letztes Element einer Menge:
    iterator end();

```

```

    // const-Vorwaerts-Iterator,
    // zeigt hinter letztes Element einer const-Menge:
    const_iterator end() const;

    // Rueckwaerts-Iterator,
    // zeigt auf letztes Element einer Menge:
    reverse_iterator rbegin();

    // const-Rueckwaerts-Iterator,
    // zeigt auf letztes Element einer const-Menge:
    const_reverse_iterator rbegin() const;

    // Rueckwaerts-Iterator,
    // zeigt vor erstes Element einer Menge:
    reverse_iterator rend();

    // const-Rueckwaerts-Iterator,
    // zeigt vor erstes Element einer const-Menge:
    const_reverse_iterator rend() const;
    ...
};

template <class T, class Compare = less<T> >
class multiset {
    ...
public:
    ...
    // Vorwaerts-Iterator,
    // zeigt auf erstes Element einer Multimenge:
    iterator begin();

    // const-Vorwaerts-Iterator,
    // zeigt auf erstes Element einer const-Multimenge:
    const_iterator begin() const;

    // Vorwaerts-Iterator,
    // zeigt hinter letztes Element einer Multimenge:
    iterator end();

    // const-Vorwaerts-Iterator,
    // zeigt hinter letztes Element einer const-Multimenge:
    const_iterator end() const;

    // Rueckwaerts-Iterator,
    // zeigt auf letztes Element einer Multimenge:
    reverse_iterator rbegin();

```

```

    // const-Rueckwaerts-Iterator,
    // zeigt auf letztes Element einer const-Multimenge:
    const_reverse_iterator rbegin() const;

    // Rueckwaerts-Iterator,
    // zeigt vor erstes Element einer Multimenge:
    reverse_iterator rend();

    // const-Rueckwaerts-Iterator,
    // zeigt vor erstes Element einer const-Multimenge:
    const_reverse_iterator rend() const;
    ...
};

```

Bei den Iteratortypen handelt es sich wiederum “nur“ um bidirektionale Iteratoren. Bedeutsam ist jedoch, dass, wenn man mittels eines Iterators auf ein Element einer Menge (`set<T>` oder `multiset<T>`) zugreift, der zurückgegebene Wert als konstant aufgefasst wird, also keine Wertänderung des Mengen-Elementes möglich ist!

Ändern einer `set<T>` bzw. `multiset<T>`

Zur Abänderung einer Menge oder Multimenge stehen nur die üblichen `insert`-, `erase`-, `swap`- und `clear`-Funktionen sowie zwei zusätzliche `insert`- und eine zusätzliche `erase`-Funktion zur Verfügung:

```

template <class T, class Compare = less<T> >
class set {
    ...
public:
    ...
    // uebliche Funktionen:
    iterator insert(iterator pos, const T& x);

    void erase(iterator pos);
    void erase(iterator anf, iterator ende);

    void clear();
    void swap(set<T,Compare> &);

    // neu in set<>:
    pair<iterator,bool> insert(const T& x);

    template <class InputIterator>
    void insert(iterator pos, InputIterator anf, InputIterator ende);

    size_type erase( const T& x);
};

```

```

template <class T, class Compare = less<T> >
class multiset {
    ...
public:
    ...
    // uebliche Funktionen:
    iterator insert(iterator pos, const T& x);

    void erase(iterator pos);
    void erase(iterator anf, iterator ende);

    void clear();
    void swap(multiset<T,Compare> &);

    // neu in multiset<>:
    iterator insert(const T& x);

    template <class InputIterator>
    void insert(InputIterator anf, InputIterator ende);

    size_type erase( const T& x);
    ...
};

```

Erläuterung zu diesen Funktionen:

- Bei der Funktion

```
iterator insert(iterator pos, const T& x);
```

ist (bei `set<T>` und `multiset<T>`) zu beachten, dass die Position des einzufügenden Elementes `x` in die Menge bzw. Multimenge durch den Wert von `x` gegeben ist und nicht durch eine Iteratorposition `pos` vorgegeben werden kann. Aus diesem Grund ist das erste Argument `pos` eigentlich unnötig, wird aber beibehalten, da es jeweils eine entsprechende zusätzliche Funktion `... insert(const T& x);` gibt.

Bei `set<T>`'s ist weiterhin zu beachten, dass das Einfügen eines Elementes `x` nicht funktioniert, wenn ein zu `x` (bezüglich des Vergleichs) gleichwertiges Element bereits in der Menge vorhanden ist. In diesem Fall wird eben kein neues Element eingefügt.

Funktionsergebnis ist, wenn das Einfügen geklappt hat, die Iteratorposition auf das eingefügte Element, bzw., falls das Einfügen nicht geklappt hat (nur bei `set<>` möglich), die Iteratorposition auf das bereits in der Menge vorhandene, zu `x` gleichwertige Element.

- Die Funktionen

```

void erase(iterator pos);
void erase(iterator anf, iterator ende);

void clear();
void set<T,Compare>::swap(set<T,Compare> &);
void multiset<T,Compare>::swap(multiset<T,Compare> &);

```

funktionieren so wie bei allen Containerklassen üblich mit der Einschränkung, dass die `erase`-Funktionen kein Ergebnis haben!

- `template <class InputIterator>`
`void insert(InputIterator anf, InputIterator end);`

fügt in die Menge bzw. Multimenge Kopien der Elemente der durch die Iteratoren `anf` und `end` gegebenen Sequenz ein. Bei einer Menge werden nur die Elemente eingefügt, zu denen es noch kein (bezüglich des Vergleichs) gleichwertiges Element in der Menge gibt!

Der Iteratortyp `InputIterator` muss zum Elementtyp `T` der Menge bzw. Multimenge passen!

- Bei der Funktion:

```
pair<iterator,bool> set<T,Compare>::insert( const T& x);
```

wird, falls noch kein (bezüglich des Vergleiches) zu `x` gleichwertiges Element in der Menge vorhanden ist, eine Kopie des Elementes `x` in die Menge aufgenommen. Ergebnis ist in diesem Fall ein Paar vom Typ `pair<iterator,bool>` mit der Iteratorposition des eingefügten Elementes als erster Komponente und dem Wert `true` als zweiter Komponente.

Falls das Einfügen nicht klappt (weil bereits ein zu `x` gleichwertiges Element in der Menge vorhanden ist), ist im Ergebnis (vom Typ `pair<iterator,bool>`) die erste Komponente die Iteratorposition des bereits in der Menge vorhandenen, zu `x` gleichwertigen Elementes und die zweite Komponente ist `false`.

- Bei der Funktion:

```
iterator multiset<T,Compare>::insert( const T& x);
```

wird in die Multimenge eine Kopie des Elementes `x` aufgenommen und die Iteratorposition auf das eingefügte Element zurückgegeben.

- Die Funktion:

```
size_type erase(const T& x);
```

löscht aus der Menge bzw. Multimenge alle zu `x` (bezüglich des Vergleichs) gleichwertigen Elemente. Funktionsergebnis ist die Anzahl der aus der Menge bzw. Multimenge gelöschten Elemente. (Bei einer Menge kann das Funktionsergebnis somit nur 0 oder 1 sein!)

Der Anwender ist selbst dafür verantwortlich, dass die verwendeten Iteratoren gültig sind!

Sonstige Funktionen für `set<T>`'s bzw. `multiset<T>`'s

Es stehen folgende Funktionen zur Verfügung, welche aufgrund der internen Abspeicherung der Mengenelemente sehr effizient sind:

```
template <class T, class Compare = less<T> >
class set {
    ...
public:
    ...
    key_compare    key_comp() const;
    value_compare value_comp() const;

    iterator find(const T& x) const;
    size_type count(const T& x) const;

    iterator lower_bound(const T& x) const;
    iterator upper_bound(const T& x) const;
    pair<iterator, iterator> equal_range(const T& x) const;
    ...
};
```

```
template <class T, class Compare = less<T> >
class multiset {
    ...
public:
    ...
    key_compare    key_comp() const;
    value_compare value_comp() const;

    iterator find(const T& x) const;
    size_type count(const T& x) const;

    iterator lower_bound(const T& x) const;
    iterator upper_bound(const T& x) const;
    pair<iterator, iterator> equal_range(const T& x) const;
    ...
};
```

Erläuterungen:

- Die Funktionen

```
key_compare    key_comp() const;
value_compare value_comp() const;
```

geben als Ergebnis das (Vergleichs-)Funktionsobjekt zurück, welches der Erzeugung der Menge bzw. Multimenge zu Grunde liegt.

- Die Funktion

```
size_type count(count const T& x) const;
```

liefert als Ergebnis die Anzahl der Elemente der Menge bzw. Multimenge, welche zu **x** gleichwertig sind. (Bei einer `set<T>` kann das Funktionsergebnis also nur 0 oder 1 sein, bei einer `multiset<T>` sind auch größere Funktionsergebnisse möglich!)

- Die Funktion

```
iterator find(const T& x) const;
```

sucht in der Menge bzw. Multimenge nach einem Element gleichwertig zu **x** (bei einer `multiset<T>` nach dem ersten derartigen Element). Gibt dessen Position zurück oder `end()`, falls kein entsprechendes Element in der Menge bzw. Multimenge vorhanden ist!

- Die Funktion

```
iterator lower_bound(const T& x) const;
```

liefert die Position des ersten Elementes der Menge bzw. Multimenge, dessen Wert (bezüglich des Vergleichs) nicht kleiner als der von **x** ist. Gibt es in der Menge bzw. Multimenge kein derartiges Element (weil alle Mengenelemente einen Wert kleiner als **x** haben), wird `end()` zurückgegeben.

(Bei einer Multimenge ist die zurückgegebene Iteratorposition somit die erste Position, an der das Element **x** eingefügt werden könnte. Liegt eine Menge zu Grunde und gibt es in der Menge kein zu **x** gleichwertiges Element, so müsste **x**, falls **x** in die Menge aufgenommen werden sollte, an dieser zurückgegebenen Iteratorposition eingefügt werden. Liegt eine Menge zu Grunde und gibt es in der Menge ein zu **x** gleichwertiges Element, so zeigt die zurückgegebene Iteratorposition auf das gleichwertige Element.)

- Die Funktion

```
iterator upper_bound(const T& x) const;
```

liefert die Position des ersten Elementes der Menge bzw. Multimenge, dessen Wert (bezüglich des Vergleichs) größer als der von **x** ist. Gibt es in der Menge bzw. Multimenge kein derartiges Element, wird `end()` zurückgegeben.

- Die Funktion

```
pair<iterator,iterator> equal_range(const T& x) const;
```

liefert als Ergebnis ein Paar von Iteratorpositionen, wobei die erste Iteratorposition die ist, die auch von der Funktion `lower_bound`, und die zweite Iteratorposition die ist, die auch von der Funktion `upper_bound` geliefert würde.

Elementzugriff für Mengenklassen

Außer über Iteratoren kann man nicht auf Elemente einer Menge oder Multimenge zugreifen.

Somit gibt es auch die bei allen bisherigen Containerklassen vorhandenen Zugriffsfunktionen `front` und `back` nicht!

Globale Operatoren und Funktionen für Mengenklassen

Es sind die üblichen Vergleichs-Operatoren als globale Operator-Funktionen:

```
template <class T, class Compare = less<T> >
bool operator== (const set<T,Compare>& x, const set<T,Compare>& y);
```

```
template <class T, class Compare>
bool operator< (const set<T,Compare>& x, const set<T,Compare>& y);
```

```
template <class T, class Compare = less<T> >
bool operator!= (const set<T,Compare>& x, const set<T,Compare>& y);
```

```
template <class T, class Compare = less<T> >
bool operator> (const set<T,Compare>& x, const set<T,Compare>& y);
```

```
template <class T, class Compare = less<T> >
bool operator>= (const set<T,Compare>& x, const set<T,Compare>& y);
```

```
template <class T, class Compare = less<T> >
bool operator<= (const set<T,Compare>& x, const set<T,Compare>& y);
```

```
template <class T, class Compare = less<T> >
bool operator== (const multiset<T,Compare>& x,
                 const multiset<T,Compare>& y);
```

```
template <class T, class Compare = less<T> >
bool operator< (const multiset<T,Compare>& x,
                 const multiset<T,Compare>& y);
```

```
template <class T, class Compare = less<T> >
bool operator!= (const multiset<T,Compare>& x,
                 const multiset<T,Compare>& y);
```

```
template <class T, class Compare = less<T> >
bool operator> (const multiset<T,Compare>& x,
                 const multiset<T,Compare>& y);
```

```
template <class T, class Compare = less<T> >
bool operator>= (const multiset<T,Compare>& x,
                 const multiset<T,Compare>& y);
```



```
template <class T, class Compare = less<T> >
bool operator<= (const multiset<T,Compare>& x,
               const multiset<T,Compare>& y);
```

sowie die (möglicherweise sehr effizient implementierten) globalen Funktion zum Vertauschen zweier Mengen vorhanden:

```
template <class T, class Compare = less<T> >
void swap( set<T,Compare>& x, set<T,Compare>& y);

template <class T, class Compare = less<T> >
void swap( multiset<T,Compare>& x, multiset<T,Compare>& y);
```

11.3.7 Die Containerklassen `map<Key,T>` und `multimap<Key,T>`

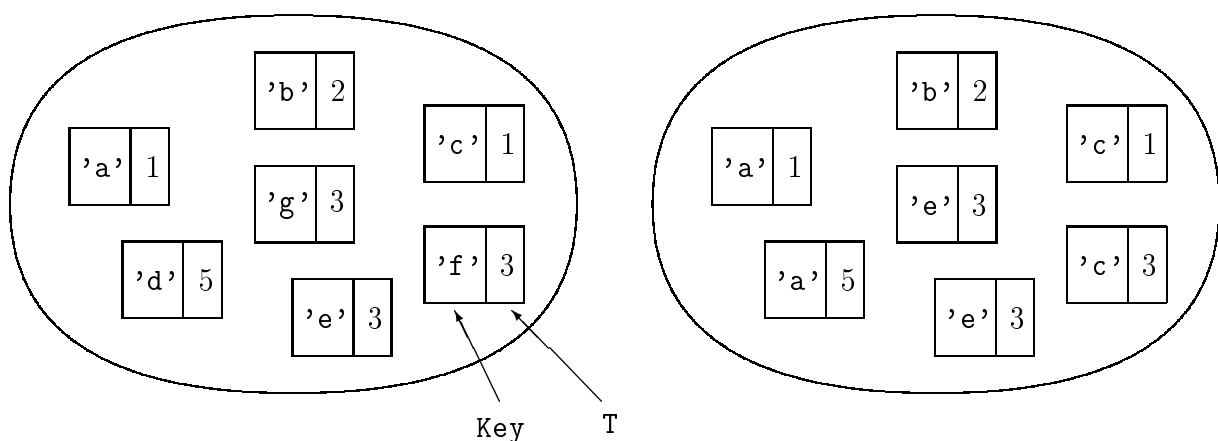
Die Funktionalität der Klassen `map<Key,T>` bzw. `multimap<Key,T>` ist nahezu identisch zu der der Mengenklassen `set<T>` bzw. `multiset<T>` — Unterschied ist (im Wesentlichen) nur, dass die abgespeicherten Elemente Paare des Types `pair<const Key,T>` (die Typen `Key` und `T` sind im Allgemeinen verschieden) sind (wobei die erste Komponente vom Typ `Key` als konstant angesehen wird!) und dass der zu Grunde liegende Vergleich (standardmäßig mit `<` oder auch durch einen anderen, bei der Template-Instanziierung anzugebenden Funktionsobjekttypen) sich nur auf den Schlüsseltyp `Key` bezieht.

Zur Verwendung von `map<Key,T>`'s bzw. `multimap<Key,T>`'s muss die Headerdatei `<map>` inkludiert werden.

Der Unterschied zwischen einer `map<Key,T>` und einer `multimap<Key,T>` ist wiederum, dass bei einer `map<Key,T>` die Schlüssel aller Element-Paare verschieden sein müssen, bei einer `multimap<Key,T>` kann es verschiedene Element-Paare mit dem gleichen Schlüssel geben (d.h. ein- und derselbe Schlüssel darf mehrfach vorkommen):

`map<char,int>`:

`multimap<char,int>`:



Bei der Definition einer `map<Key,T>` bzw. einer `multimap<Key,T>` sind entsprechend zwei Typen anzugeben, nämlich der Schlüsseltyp (`Key`) und der Typ der Werte (`T`), etwa:

```
map<const char *, int> assoc;
```

zur Erzeugung einer (noch leeren) `map` mit Schlüsseln vom Typ `const char *` `const` (das zweite `const` kommt von der `map<Key,T>` bzw. `multimap<Key,T>` selbst, da die Schlüssel — hier also die Zeiger — als `const` angesehen werden!) und Werten vom Typ `int`. Die Schlüssel werden hierbei mittels `<` verglichen.

Möchte man die Schlüssel nicht mittels `<` vergleichen, muss als drittes Template-Argument der entsprechende (Vergleichs-)Funktionsobjekttyp angegeben werden, etwa:

```
#include <map>
#include <cstring>

struct vergleich
    : public binary_function<const char *, const char *, bool>
{
    public:
    bool operator() ( const char *a, const char *b)
    {
        if (strcmp(a,b) < 0)
            return true;

        return false;
    }
};

map<const char *, int, vergleich> assoc;
multimap<const char *, int, vergleich> m_assoc;
```

(Im Folgenden wird wiederum bei der abkürzenden Schreibweise `multimap<Key,T>` bzw. `map<Key,T>` auf die Angabe des dritten Template-Argumentes zum Vergleichsfunktionsobjekttyp verzichtet! Eigentlich müsste dieser wiederum angegeben werden, etwa `map<Key,T,Vergleich>` bzw. `multimap<Key,T,Vergleich>`!)

Intension einer `map<Key,T>` bzw. `multimap<Key,T>` ist es wie bei den Mengenklassen `set<T>`'s bzw. `multiset<T>`'s, das schnelle Auffinden von Elementen, wobei es jedoch (nur bei `map<Key,T>`'s) mittels des `[]`-Operators über die Schlüssel einen speziellen Zugriff auf die Werte der abgespeicherten Elemente gibt, etwa:

```
assoc["hallo"];
```

liefert (eine Referenz auf) den Wert des in `assoc` abgespeicherten Paares, dessen Schlüssel die konstante Zeichenkette "hallo" ist. (Sollte es hierbei in `assoc` noch kein Paar mit dem Schlüssel "hallo" geben, wird eins erzeugt, wobei der zugehörige Wert mit seinem Standardkonstruktor erzeugt wird!)

Mittels einer `map<Key,T>` kann man somit (in gewisser Hinsicht) "Felder" von Objekten vom Typ `T` (hier im Beispiel `int`) simulieren, wobei die "Indizierung" jedoch mit Werten vom Typ `Key` (hier im Beispiel `const char *`) erfolgt. Derartige "Felder" nennt man auch *Assoziative Felder*.

Die "schnelle Auffindbarkeit" der Elemente macht wiederum eine Vergleichsfunktion für Schlüssel und eine spezielle interne Struktur (ausbalancierter Baum) notwendig.

Typen der `map<Key,T>` bzw. `multimap<Key,T>`-Klasse

In den Template-Klassen `map<Key,T>` bzw. `multimap<Key,T>` sind (wie bei `set<T>`'s und `multiset<T>`'s) wiederum einige Typen mehr als in anderen Containerklassen üblich definiert (vgl. Abschnitt 11.3.1):

```
template <class Key, class T, class Compare = less<Key> >
class map {
    ...
public:
    ...
    // wie in allen Containerklassen:
    typedef ...           reference;
    typedef ...           const_reference;
    typedef ...           iterator;
    typedef ...           const_iterator;
    typedef ...           reverse_iterator;
    typedef ...           const_reverse_iterator;
    typedef ...           size_type;
    typedef ...           difference_type;
    typedef pair<const Key,T> value_type;
    typedef ...           pointer;
    typedef ...           const_pointer;

    // zusaetzlich:
    typedef Key            key_type;
    typedef T              mapped_type;
    typedef Compare        key_compare;
    ...
};
```

```
template <class Key, class T, class Compare = less<Key> >
class multimap {
    ...
public:
    ...
    // wie in allen Containerklassen:
    typedef ...           reference;
    typedef ...           const_reference;
    typedef ...           iterator;
    typedef ...           const_iterator;
    typedef ...           reverse_iterator;
    typedef ...           const_reverse_iterator;
    typedef ...           size_type;
    typedef ...           difference_type;
    typedef pair<const Key,T> value_type;
    typedef ...           pointer;
    typedef ...           const_pointer;
```

```

// zusaetzlich:
typedef Key          key_type;
typedef T            mapped_type;
typedef Compare      key_compare;
class               value_comp { ... };
...
};

```

Die in der `map<Key,T>` bzw. `multimap<Key,T>` abgepeicherten Elemente sind Paare (Typ `pair<const Key,T>`) und der `value_type` ist ein anderer Name für diesen Paar-Typ (man beachte: die erste Komponente solcher Paare wird als konstant angesehen!). Der Typ `key_type` ist ein anderer Name für den Schlüsseltyp `Key`, der Typ `mapped_type` ein anderer Name für den zweiten Typ `T` des Paartypes `value_type`.

`key_compare` ist andere Namen für den dem Vergleich der Schlüssel zu Grunde liegenden Funktionsobjekttypen.

Der Typ `value_comp` ist ein Funktionsobjekttyp zum Vergleich zweier Elemente einer `map<Key,T>` bzw. `multimap<Key,T>`, also des Vergleichs zweier Paare vom Typ `pair<const Key,T>`. Dieser Vergleich von Paaren wird auf den Vergleich der Schlüssel mittels `key_compare` zurückgeführt.

Die Bedeutung der anderen Typen ist in Abschnitt 11.3.1 erläutert.

Erzeugen, Zuweisen, Zerstören einer `map<Key,T>` bzw. `multimap<Key,T>`

Bei den Konstruktoren zur Klasse `map<Key,T>` und `multimap<Key,T>` (vom Copy-Konstruktor abgesehen) kann wie bei `set<T>`'s und `multiset<T>`'s als zusätzliches Argument ein konkretes Vergleichs-Funktionsobjekt (also ein Objekt des als Template-Parameter angegebenen Funktionsobjekttypes) angegeben werden (dieses wird wohl irgendwie im `private`- oder `protected`-Teil abgespeichert). Standardmäßig wird hier das mit dem zugehörigen Standardkonstruktor erzeugte Objekt des Funktionsobjekttypes genommen!

```

template <class Key, class T, class Compare = less<Key> >
class map {
    ...
public:
    ...
    // Standardkonstruktor
    explicit map(const Compare& cmp = Compare() );

    // Copy--Konstruktor
    map( const map<Key, T, Compare>&);

    // mit Sequenz initialisieren
    template <class InputIterator>
    map(InputIterator anf, InputIterator ende,
        const Compare& cmp = Compare());

```

```

    // Destruktor
    ~map();

    // Zuweisungsoperator
    map<Key,T,Compare> & operator=( const map<Key, T,Compare> &);
    ...
};

template <class Key, class T, class Compare = less<Key> >
class multimap {
    ...
public:
    ...
    // Standardkonstruktor
    explicit multimap(const Compare& cmp = Compare() );

    // Copy--Konstruktor
    multimap( const multimap<Key,T,Compare>&);

    // mit Sequenz initialisieren
    template <class InputIterator>
    multimap(InputIterator anf, InputIterator ende,
              const Compare& cmp = Compare());

    // Destruktor
    ~multimap();

    // Zuweisungsoperator
    multimap<Key,T,Compare> & operator=(const multimap<Key,T,Compare> &);
    ...
};

```

Neben den bei allen Containerklassen vorhandenen Konstruktoren und Zuweisungsoperator gibt es keine weiteren Konstruktoren oder Zuweisungsfunktionen.

Größe einer `map<Key,T>` bzw. `multimap<Key,T>`

Wie bei den Kengenklassen `set<T>` und `multiset<T>` kann auch bei `map<Key,T>`'s und `multimap<Key,T>`'s nur die aktuelle Elementzahl, maximale Elementzahl und "Leerheit" abgefragt werden:

```

template <class Key, class T, class Compare = less<Key> >
class map {
    ...
public:
    ...

```

```

    size_type size() const;        // aktuelle Elementzahl
    size_type max_size() const;    // maximale Elementzahl
    bool empty() const;           // map leer?
    ...
};

template <class Key, class T, class Compare = less<Key> >
class multimap {
    ...
public:
    ...
    size_type size() const;        // aktuelle Elementzahl
    size_type max_size() const;    // maximale Elementzahl
    bool empty() const;           // multimap leer?
    ...
};

```

Vergrößern einer `map<Key,T>` oder `multimap<Key,T>` ist nur durch explizites Einfügen eines neuen Elementes möglich (wie bei den Mengenklassen gibt es keine Funktion `resize()`)!

Iteratoren für `map<Key,T>`'s und `multimap<Key,T>`'s

Es stehen die üblichen Iteratortypen und Funktionen zur Verfügung, welche Iteratorpositionen liefern.

```

template <class Key, class T, class Compare = less<Key> >
class map {
    ...
public:
    ...
    // Vorwaerts-Iterator,
    // zeigt auf erstes Element einer map:
    iterator begin();

    // const-Vorwaerts-Iterator,
    // zeigt auf erstes Element einer const-map:
    const_iterator begin() const;

    // Vorwaerts-Iterator,
    // zeigt hinter letztes Element einer map:
    iterator end();

    // const-Vorwaerts-Iterator,
    // zeigt hinter letztes Element einer const-map:
    const_iterator end() const;

```

```

    // Rueckwaerts-Iterator,
    // zeigt auf letztes Element einer map:
    reverse_iterator rbegin();

    // const-Rueckwaerts-Iterator,
    // zeigt auf letztes Element einer const-map:
    const_reverse_iterator rbegin() const;

    // Rueckwaerts-Iterator,
    // zeigt vor erstes Element einer map:
    reverse_iterator rend();

    // const-Rueckwaerts-Iterator,
    // zeigt vor erstes Element einer const-map:
    const_reverse_iterator rend() const;
    ...
};

template <class Key, class T, class Compare = less<Key> >
class multimap {
    ...
public:
    ...
    // Vorwaerts-Iterator,
    // zeigt auf erstes Element einer multimap:
    iterator begin();

    // const-Vorwaerts-Iterator,
    // zeigt auf erstes Element einer const-multimap:
    const_iterator begin() const;

    // Vorwaerts-Iterator,
    // zeigt hinter letztes Element einer multimap:
    iterator end();

    // const-Vorwaerts-Iterator,
    // zeigt hinter letztes Element einer const-multimap:
    const_iterator end() const;

    // Rueckwaerts-Iterator,
    // zeigt auf letztes Element einer multimap:
    reverse_iterator rbegin();

    // const-Rueckwaerts-Iterator,
    // zeigt auf letztes Element einer const-multimap:
    const_reverse_iterator rbegin() const;

```

```

// Rueckwaerts-Iterator,
// zeigt vor erstes Element einer multimap:
reverse_iterator rend();

// const-Rueckwaerts-Iterator,
// zeigt vor erstes Element einer const-multimap:
const_reverse_iterator rend() const;
...
};

```

Bei den Iteratortypen handelt es sich wiederum “nur“ um bidirektionale Iteratoren. Da die Elemente, auf welche die Iteratoren zeigen, den Typen `pair<const Key, T>` haben, kann man, wenn man mittels eines Iterators auf ein Element einer `map<Key, T>` oder `multimap<Key, T>` zugreift, den Schlüssel nicht ändern (da Typ `const Key`), den Wert kann man im Allgemeinen sehr wohl ändern (da Typ `T`), falls die `map<Key, T>` bzw. `multimap<Key, T>` selbst nicht konstant ist! Möchte man wirklich in einer `map<Key, T>` oder `multimap<Key, T>` den Schlüssel eines Elementes ändern, so muss man das Element aus der `map<Key, T>` bzw. `multimap<Key, T>` entfernen (etwa mit `erase`, s.u.) und mit verändertem Schlüssel erneut einfügen (etwa mit `insert`, s.u.).

Ändern einer `map<Key, T>` bzw. `multimap<Key, T>`

Zur Abänderung einer `map<Key, T>` bzw. `multimap<Key, T>` stehen die gleichen Funktionen wie bei den Mengenklassen zur Verfügung — es ist nur zu beachten, dass die einzufügenden bzw. zu entfernenden Elemente den Typen `pair<const Key, T>` (alias `value_type`) haben und “Gleichwertigkeit“ immer nur anhand des Vergleichs der Schlüssel festgestellt wird:

```

template <class Key, class T, class Compare = less<Key> >
class map {
    ...
public:
    ...
    iterator insert(iterator pos, const value_type& x);
    pair<iterator, bool> insert(const value_type& x);

    template <class InputIterator>
    void insert(iterator pos, InputIterator anf, InputIterator ende);

    void erase(iterator pos);
    void erase(iterator anf, iterator ende);
    size_type erase(const value_type& x);

    void clear();
    void swap(map<Key, T, Compare> &);
    ...
}

```



```

};

template <class Key, class T, class Compare = less<Key> >
class multimap {
    ...
public:
    ...
    iterator insert(iterator pos, const value_type& x);
    iterator insert(const value_type& x);

    template <class InputIterator>
    void insert(InputIterator anf, InputIterator ende);

    void erase(iterator pos);
    void erase(iterator anf, iterator ende);
    size_type erase( const key_type& x);

    void clear();
    void swap(multimap<Key,T,Compare> &);
    ...
};

```

Erläuterung zu diesen Funktionen:

- Bei der Funktion

```
iterator insert(iterator pos, const value_type& x);
```

ist (bei `map<Key,T>` und `multimap<Key,T>`) zu beachten, dass die Position des einzufügenden Elementes `x` in die `map<Key,T>` bzw. `multimap<Key,T>` durch den Schlüsselwert von `x` gegeben ist und nicht durch eine Iteratorposition `pos` vorgegeben werden kann. Aus diesem Grund ist das erste Argument `pos` eigentlich unnötig, wird aber beibehalten, da es jeweils eine entsprechende zusätzliche Funktion `... insert(const value_type& x);` gibt.

Bei `map<Key,T>`'s ist weiterhin zu beachten, dass das Einfügen eines Elementes `x` nicht funktioniert, wenn ein zu `x` (bezüglich des Vergleichs der Schlüssel) gleichwertiges Element bereits in der `map<Key,T>` vorhanden ist. In diesem Fall wird eben kein neues Element eingefügt.

Funktionsergebnis ist, wenn das Einfügen geklappt hat, die Iteratorposition auf das eingefügte Element, bzw., falls das Einfügen nicht geklappt hat (nur bei `map<Key,T>` möglich), die Iteratorposition auf das bereits in der `map<Key,T>` vorhandene, zu `x` gleichwertige Element.

- Die Funktionen

```
void erase(iterator pos);
void erase(iterator anf, iterator ende);
```

```
void clear();
void map<T,Compare>::swap(map<T,Compare> &);
void multimap<T,Compare>::swap(multimap<T,Compare> &);
```

funktionieren so wie bei allen Conatinerklassen üblich (`erase` hat wie bei den Mengenklassen wiederum kein Ergebnis!).

- `template <class InputIterator>`
`void insert(InputIterator anf, InputIterator end);`

fügt in die `map<Key,t>` bzw. `multimap<Key,T>` Kopien der Elemente der durch die Iteratoren `anf` und `end` gegebenen Sequenz (von Objekten des Types `pair<Key,T>`) ein. Bei einer `map<Key,T>` werden nur die Elemente eingefügt, zu denen es noch kein (bezüglich des Schlüsselvergleichs) gleichwertiges Element in der `map<Key,T>` gibt!

Der Iteratortyp `InputIterator` muss zum Elementtyp `pair<const Key,T>` der `map<Key,T>` bzw. `multimap<Key,T>` passen!

- Bei der Funktion:

```
pair<iterator,bool> map<Key,T,Compare>::insert( const T& x);
```

wird, falls noch kein (bezüglich des Schlüsselvergleiches) zu `x` gleichwertiges Element in der `map<Key,T>` vorhanden ist, eine Kopie des Elementes `x` in die `map<Key,T>` aufgenommen. Ergebnis ist in diesem Fall ein Paar vom Typ `pair<iterator,bool>` mit der Iteratorposition des eingefügten Elementes als erster Komponente und dem Wert `true` als zweiter Komponente.

Falls das Einfügen nicht klappt (weil bereits in der `map<Key,T>` ein Element mit zu `x` gleichwertigem Schlüssel vorhanden ist), ist im Ergebnis (vom Typ `pair<iterator,bool>`) die erste Komponente die Iteratorposition des bereits in der `map<Key,T>` vorhandenen, zu `x` gleichwertigen Elementes und die zweite Komponente ist `false`.

- Bei der Funktion:

```
iterator multimap<Key,T,Compare>::insert( const T& x);
```

wird in die Multimenge eine Kopie des Elementes `x` aufgenommen und die Iteratorposition auf das eingefügte Element zurückgegeben.

- Die Funktion:

```
size_type erase(const key_type& x);
```

löscht aus der `map<Key,T>` bzw. `multimap<Key,T>` alle Elemente mit zu `x` gleichwertigem Schlüssel. Funktionsergebnis ist die Anzahl der aus der `map<Key,T>` bzw. `multimap<Key,T>` gelöschten Elemente. (Bei einer `map<Key,t>` kann das Funktionsergebnis somit nur 0 oder 1 sein!)

Bei der Anwendung dieser Funktionen ist der Anwender selbst dafür verantwortlich, dass die verwendeten Iteratoren gültig sind!

Sonstige Funktionen für `map<Key,T>`'s bzw. `multimap<Key,T>`'s

Es stehen die gleichen Funktionen wie bei den Mengenklassen zur Verfügung. Auch bei `map<Key,t>`'s und `multimap<Key,T>`'s sind diese aufgrund der internen Abspeicherung der Elemente sehr effizient.

Zu Beachten ist, dass die Argumente der Funktionen vom Typ `key_type` sind und dass bei Vergleichen und Gleichheit dieses Argument mit dem jeweiligen Schlüssel der Elemente der `map<Key,T>` bzw. `multimap<Key,T>` verglichen wird.

```
template <class Key, class T, class Compare = less<Key> >
class map {
    ...
public:
    ...
    key_compare    key_comp() const;
    value_compare  value_comp() const;

    iterator       find(const key_type& x);
    const_iterator find(const key_type& x) const;

    size_type count(const key_type& x) const;

    iterator       lower_bound(const key_type& x);
    const_iterator lower_bound(const key_type& x) const;

    iterator       upper_bound(const key_type& x);
    const_iterator upper_bound(const key_type& x) const;

    pair<iterator, iterator> equal_range(const key_type& x);

    pair<const_iterator, const_iterator>
        equal_range(const key_type& x) const;
    ...
};
```

```
template <class Key, class T, class Compare = less<Key> >
class multimap {
    ...
public:
    ...
    key_compare    key_comp() const;
    value_compare  value_comp() const;

    iterator       find(const key_type& x);
    const_iterator find(const key_type& x) const;

    size_type count(const key_type& x) const;
```

```

iterator      lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;

iterator      upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;

pair<iterator, iterator> equal_range(const key_type& x);

pair<const_iterator, const_iterator>
    equal_range(const key_type& x) const;
    ...
};

```

Diese Funktionen funktionieren im Wesentlichen wie bei den Mengenklassen. Da man über die Iteratoren (theoretisch) den Wert eines Elementes einer `map<Key,T>` bzw. `multimap<Key,T>` ändern kann (der Schlüssel kann nicht geändert werden!), gibt es zu allen Funktionen, welche einen Iterator oder ein Paar von Iteratoren als Ergebnis liefern, eine entsprechende konstante und eine nicht konstante Memberfunktion. Bei den konstanten Memberfunktionen werden im Ergebnis `const_iteratoren` geliefert, damit über diese Iteratoren die Elemente nicht geändert werden können.

- Die Funktion

```
key_compare key_comp() const;
```

gibt als Ergebnis das (Vergleichs-)Funktionsobjekt zum Schlüsselvergleich zurück, welches der Erzeugung der `map<Key,T>` bzw. `multimap<Key,T>` zu Grunde liegt.

- Die Funktion

```
value_compare value_comp() const;
```

gibt als Ergebnis das auf den Schlüsselvergleich zurückgeführte Vergleichsfunktionsobjekt zum Vergleich von Elementen der `map<Key,T>` bzw. `multimap<Key,T>` zurück.

- Die Funktion

```
size_type count(const key_type& x) const;
```

liefert als Ergebnis die Anzahl der Elemente der `map<Key,T>` bzw. `multimap<Key,T>` zurück, deren Schlüssel zu `x` gleichwertig sind. (Bei einer `map<Key,T>` kann das Funktionsergebnis also nur 0 oder 1 sein, bei einer `multimap<Key,T>` sind auch größere Funktionsergebnisse möglich!)

- Die Funktion

```
iterator      find(const key_type& x);
const_iterator find(const key_type& x) const;
```

sucht in der `map<Key,T>` bzw. `multimap<Key,T>` nach einem Element, dessen Schlüssel gleichwertig zu `x` (bei einer `multimap<Key,T>` nach dem ersten derartigen Element). Gibt dessen Position zurück oder `end()`, falls kein entsprechendes Element in der `map<Key,T>` bzw. `multimap<Key,T>` vorhanden ist!

- Die Funktion

```
iterator      lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
```

liefert die Position des ersten Elementes der `map<Key,T>` bzw. `multimap<Key,T>`, dessen Schlüssel (bezüglich des Vergleichs) nicht kleiner als `x` ist. Gibt es in der `map<Key,T>` bzw. `multimap<Key,T>` kein derartiges Element, wird `end()` zurückgegeben.

- Die Funktion

```
iterator      upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;
```

liefert die Position des ersten Elementes der `map<Key,T>` bzw. `multimap<Key,T>`, dessen Schlüssel (bezüglich des Schlüsselvergleichs) größer als `x` ist. Gibt es in der `map<Key,T>` bzw. `multimap<Key,T>` kein derartiges Element, wird `end()` zurückgegeben.

- Die Funktion

```
pair<iterator,iterator> equal_range(const key_type& x);
pair<const_iterator,const_iterator>
    equal_range(const key_type& x) const;
```

liefert als Ergebnis ein Paar von Iteratorpositionen, wobei die erste Iteratorposition die ist, die auch von der Funktion `lower_bound`, und die zweite Iteratorposition die ist, die auch von der Funktion `upper_bound` geliefert würde.

Direkter Elementzugriff in der Klasse `map<Key,T>`

Wie bereits erwähnt kann bei einer `map<Key,T>` (bei einer `multimap<Key,T>` nicht möglich!) mittels `[]` über einen Schlüssel direkt auf den zum Schlüssel gehörenden Elementwert zugegriffen werden:

```
template <class Key, class T, class Compare = less<Key> >
class map {
    ...
public:
    ...
    T& operator[](const key_type& x);
    ...
};
```

liefert Referenz auf den Wert des Elementes zum Schlüssel `x`.

Sollte noch kein Element mit diesem Schlüssel vorhanden sein, so wird eins eingefügt, wobei der zugehörige Wert mittels dessen Standard-Konstruktor erzeugt wird.

Somit kann es keinen “unzulässigen Index“ (vom Typ `key_type`) geben — ggf. wird ein entsprechendes Element erzeugt und eingefügt!

Diese Operatorfunktion ist eine nicht konstante Funktion, kann also nicht für eine konstante `map<Key,T>` aufgerufen werden.

Ansonsten ist der Zugriff auf Elemente einer `map<Key,T>` bzw. `multimap<Key,T>` nur über entsprechende Iteratoren möglich — wie bei den Mengenklassen gibt es keine Zugriffsfunktionen `front()` bzw. `back()`.

Globale Operatoren und Funktionen für `map<Key,T>`’s und `multimap<Key,T>`’s

Es sind die üblichen Vergleichs-Operatoren als globale Operator-Funktionen:

```
template <class Key, class T, class Compare = less<Key> >
bool operator== (const map<Key,T,Compare>& x,
                 const map<Key,T,Compare>& y);
```

```
template <class Key, class T, class Compare = less<Key> >
bool operator< (const map<Key,T,Compare>& x,
                const map<Key,T,Compare>& y);
```

```
template <class Key, class T, class Compare = less<Key> >
bool operator!= (const map<Key,T,Compare>& x,
                 const map<Key,T,Compare>& y);
```

```
template <class Key, class T, class Compare = less<Key> >
bool operator> (const map<Key,T,Compare>& x,
                const map<Key,T,Compare>& y);
```

```
template <class Key, class T, class Compare = less<Key> >
bool operator>= (const map<Key,T,Compare>& x,
                 const map<Key,T,Compare>& y);
```

```
template <class Key, class T, class Compare = less<Key> >
bool operator<= (const map<Key,T,Compare>& x,
                 const map<Key,T,Compare>& y);
```

```
template <class Key, class T, class Compare = less<Key> >
bool operator== (const multimap<Key,T,Compare>& x,
                 const multimap<Key,T,Compare>& y);
```

```
template <class Key, class T, class Compare = less<Key> >
bool operator< (const multimap<Key,T,Compare>& x,
                const multimap<Key,T,Compare>& y);
```

```
template <class Key, class T, class Compare = less<Key> >
bool operator!= (const multimap<Key,T,Compare>& x,
                const multimap<Key,T,Compare>& y);
```

```
template <class Key, class T, class Compare = less<Key> >
bool operator> (const multimap<Key,T,Compare>& x,
               const multimap<Key,T,Compare>& y);
```

```
template <class Key, class T, class Compare = less<Key> >
bool operator>= (const multimap<Key,T,Compare>& x,
                const multimap<Key,T,Compare>& y);
```

```
template <class Key, class T, class Compare = less<Key> >
bool operator<= (const multimap<Key,T,Compare>& x,
                const multimap<Key,T,Compare>& y);
```

sowie die (möglicherweise sehr effizient implementierten) globalen Funktion zum Vertauschen von zwei `map<Key,T>`'s bzw. `multimaps<Key,T>`'s vorhanden:

```
template <class Key, class T, class Compare = less<Key> >
void swap( map<Key,T,Compare>& x, map<Key,T,Compare>& y);
```

```
template <class Key, class T, class Compare = less<Key> >
void swap( multimap<Key,T,Compare>& x, multimap<Key,T,Compare>& y);
```

11.4 Container-Adapter

Wie bereits erwähnt, verfügt der Standard über weitere Containerklassen, welche aber mittels der bereits erwähnten implementiert sind und eine neue, eingeschränkte Schnittstelle zu der bereits vorhandenen Containerklasse zur Verfügung stellen.

Diese zusätzlichen Containerklassen werden auch *Container-Adapter* genannt.

I. Allg. kann bei der Definition eines derartigen Container-Adapters angegeben werden, auf welchem der Standardcontainer dieser beruhen soll (es sind nicht immer alle Standardcontainer geeignet!).

Container-Adapter haben selbst keine Iteratoren, so dass man sie auch nicht über Iteratoren bearbeiten kann. (Hier müsste man auf die Iteratoren des dem Container-Adapters zu Grunde liegenden Standard-Containers ausweichen, wodurch aber die Schnittstelle des Container-Adapters umgangen würde!)

11.4.1 Der Containeradapter `queue<T>`

Ein Objekt der Klasse `queue<T>` stellt einen klassischen Datenpuffer von Objekten vom Typ `T` dar, in dem nach dem Fifo-Prinzip (*First in first out*) Elemente (vom Typ `T`) "hinten" eingefügt (`push`) und "vorne" wieder entfernt (`pop`) werden können. Zur Verwendung von `queue<T>`'s muss die Headerdatei `<queue>` eingebunden werden.

Die wesentlichen `queue<T>`-Operationen

Erzeugt wird eine (zunächst leere) `queue<T>` mit Elementtyp `T` und Namen `puffer` mittels des Konstruktors:

```
queue<T> puffer;
```

Durch die Funktion

```
puffer.push(elem);
```

wird eine Kopie von `elem` vom Typ `T` (hinten) in den Puffer aufgenommen.

Die Funktion

```
puffer.front();
```

gibt als Ergebnis die Referenz auf das vorderste (zuerst eingefügte) Element in `puffer` zurück. Das Element verbleibt in `puffer`.

Die Funktion

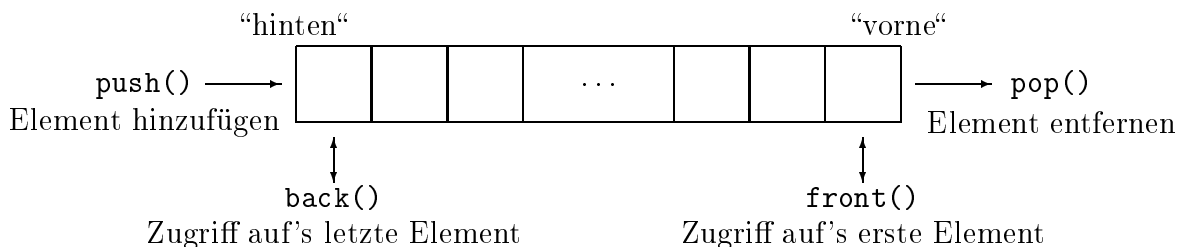
```
puffer.pop();
```

entfernt das vorderste (zuerst eingefügte) Element aus `puffer`.

Neben diesen, traditionellerweise für Fifo's vorhandenen Operationen verfügt eine `queue<T>` noch über einen Zugriff (liefert Referenz auf das Element, das Element kann hiermit aber nicht entfernt werden!) auf das zuletzt eingefügte Element (also das Element "hinten"):

```
puffer.back()
```

Eine Übersicht über diese Standard-Verwendung einer `queue<T>` gibt folgendes Bild:



`queue<T>`'s als Container-Adapter

Die `queue<T>`-Klasse ist nicht als eigenständige Klasse realisiert, sondern die Funktionalität einer `queue<T>` wird auf bereits vorhandene Containerklassen (etwa auf `list<T>` oder `deque<T>`) zurückgeführt.

Man sagt: die `queue<T>`-Klasse ist nur ein "Adapter" für die andere Containerklasse, d.h. diese andere Containerklasse wird durch eine spezielle, nämlich die `queue<T>`-, Schnittstelle zur Verfügung gestellt!

Die vorhandene Containerklasse muss hierzu über die Element-Funktionen `front`, `back`, `push_back` und `pop_front` verfügen.

Die `queue<T>`-Operation `push` wird hierbei auf die Container-Operation `push_back` zurückgeführt, die `queue<T>`-Operation `pop` auf `pop_front` und die `queue<T>`-Elementzugriffs-Operation `front` bzw. `back` auf die Container-Operationen `front` bzw. `back`.

Als "Grundlage" für eine `queue<T>` kann also jede Containerklasse mit (mindestens)

diesen vier Operationen `push_back`, `pop_front`, `front` und `back` genommen werden. Aus der Standardbibliothek sind dies die Containerklassen `list<T>` und `deque<T>`. Bei der Erzeugung einer `queue<T>` kann als zweites Argument die Containerklasse angegeben werden, auf welcher die `queue<T>` beruhen soll, also etwa:

```
// T irgendein Typ
queue<T,list<T> > puffer1; // auf list<T> beruhender Puffer
queue<T,deque<T> > puffer2; // auf deque<T> beruhender Puffer
queue<T> puffer3;           // auf deque<T> beruhender Puffer
                           // deque<T> ist Standardvorgabe
```

(Nebenbei bemerkt:

einige Compiler meckern, wenn man das Leerzeichen in `queue<T,list<T> >` fortlässt, denn in `queue<T,list<T>>` fassen sie das `>>` am Schluss als Shift-Operator auf!)

Die genaue `queue<T>`-Schnittstelle

Die vom Standard vorgegebene Template-Deklaration der Klasse `queue<T>` ist wie folgt:

```
template <class T, class Container = deque<T> >
class queue {
public:
    typedef typename Container::value_type    value_type;
    typedef typename Container::size_type     size_type;
    typedef          Container                container_type;

protected:
    Container c;

public:
    // Standardkonstruktor
    explicit queue(const Container& = Container());

    bool      empty() const { return c.empty(); }
    size_type size() const  { return c.size(); }

    value_type&      front()      { return c.front(); }
    const value_type& front() const { return c.front(); }

    value_type&      back()       { return c.back(); }
    const value_type& back() const { return c.back(); }

    void push(const value_type& x) { c.push_back(x); }
    void pop()                     { c.pop_front(); }
};

template <class T, class Container>
```

```

bool operator== (const queue<T,Container>& x,
                 const queue<T,Container>& y);

template <class T, class Container>
bool operator< (const queue<T,Container>& x,
               const queue<T,Container>& y);

template <class T, class Container>
bool operator!= (const queue<T,Container>& x,
                const queue<T,Container>& y);

template <class T, class Container>
bool operator> (const queue<T,Container>& x,
               const queue<T,Container>& y);

template <class T, class Container>
bool operator>= (const queue<T,Container>& x,
                const queue<T,Container>& y);

template <class T, class Container>
bool operator<= (const queue<T,Container>& x,
                const queue<T,Container>& y);

```

Erläuterungen:

```

- template <class T, class Container = deque<T> >
  class queue { ... };

```

An dieser Template-Definition sieht man, dass man als zweites Template-Argument die Containerklasse angeben kann, auf die der Container-Adapter `queue<T>` beruhen soll. Standardmäßig wird hierzu die Containerklasse `deque<T>` (mit gleichem Elementtyp) verwendet.

```

- typedef typename Container::value_type  value_type;
  typedef typename Container::size_type   size_type;
  typedef          Container              container_type;

```

Eingebettete Typnamen:

`value_type` Elementtyp der `queue<T>` (zurückgeführt auf den Elementtyp des zu Grunde liegenden Containertypes),
`size_type` Größentyp der `queue<T>` (zurückgeführt auf den Größentyp des zu Grunde liegenden Containertypes),
`container_type` Typ des zu Grunde liegenden Containers.

```

- protected:
  Container c;

```

Der zu Grunde liegender Container ist als `protected`-Komponente abgespeichert.

- `explicit queue(const Container& = Container());`

Standard-Konstruktor, legt eine leere `queue<T>` an. Als Argument kann ein konkreter Container des zu Grunde liegenden Containertypes angegeben werden. Ansonsten wird der zu Grunde liegende Container mittels des entsprechenden Standardkonstruktors erzeugt.

- `bool empty() const { return c.empty(); }`

liefert, ob die `queue` leer ist. (Zurückgeführt auf die Funktion `empty()` des zu Grunde liegenden Containers!)

- `size_type size() const { return c.size(); }`

liefert die aktuelle Anzahl der Elemente. (Zurückgeführt auf die Funktion `size()` des zu Grunde liegenden Containers!)

- `value_type& front() { return c.front(); }`
`const value_type& front() const { return c.front(); }`

liefert Referenz auf das vorderste (zuerst gespeicherte) Element (die `queue` darf nicht leer sein)! (Zurückgeführt auf die Funktion `front()` des zu Grunde liegenden Containers!)

Bei der konstanten Member-Funktion wird eine Referenz auf ein konstantes Element zurückgegeben.

- `value_type& back() { return c.back(); }`
`const value_type& back() const { return c.back(); }`

liefert Referenz auf das hinterste (zuletzt gespeicherte) Element (die `queue` darf nicht leer sein)! (Zurückgeführt auf die Funktion `back()` des zu Grunde liegenden Containers!)

Bei der konstanten Member-Funktion wird eine Referenz auf ein konstantes Element zurückgegeben.

- `void push(const value_type& x) { c.push_back(x); }`

fügt eine Kopie des angegebenen Elementes (hinten) in die `queue` ein. (Zurückgeführt auf die Funktion `push_back()` des zu Grunde liegenden Containers.)

- `void pop() { c.pop_front(); }`

entfernt das vorderste, bereits am längsten in der `queue<T>` abgespeicherte Element aus der `queue<T>`. (Zurückgeführt auf die Funktion `pop_front()` des zu Grunde liegenden Containers.)

- Die üblichen Vergleichsoperatoren werden ebenfalls auf den Vergleich des jeweils zu Grunde liegenden Containers zurückgeführt.
- Es gibt keine spezielle Elementfunktion `swap` zum Vertauschen zweier `queue<T>`'s.
- Es gibt keinen speziellen Zuweisungsoperator, da die Funktionalität des für jede Klasse generierten Standard-Zuweisungsoperators genügt.

- Es gibt keinen speziellen Destruktor, da die Funktionalität des für jede Klasse generierten Standard-Destruktors genügt.

11.4.2 Der Containeradapter `priority_queue<T>`

Eine `priority_queue<T>` ist in ihrer Funktionalität ähnlich zu einer `queue`, d.h. man kann mit `push()` Elemente “hineinstecken“, mit `top()` auf das “vorderste“ Element zugreifen und mit `pop()` dieses “vorderste“ Elemente aus der `priority_queue<>` löschen.

Zur Verwendung einer `priority_queue<T>` muss wiederum die Headerdatei `<queue>` eingebunden werden. (In der Headerdatei `<queue>` wird sowohl der Container-Adapter-Typ `queue<T>` als auch der Container-Adapter-Typ `priority_queue<T>` definiert!) Der Unterschied zwischen einer `priority_queue<T>` und einer `queue<T>` ist der, dass eine Priorität (Vergleich der Elemente) festlegt, welches das im Augenblick “vorderste“ Element ist. (Bei einer `queue<T>` ist es immer das am längsten in der `queue<T>` stehende Element!)

Der der Priorität zu Grunde liegende Vergleich ist standardmäßig durch den Operator `<` für `T`-Elemente gegeben — kann aber (wie bei `set<T>`'s und `map<Key,T>`'s) als zusätzliches (drittes) Template-Argument angegeben werden.

Wie bei einer `queue<T>` wird auch eine `priority_queue<T>` mit einer anderen Container-Klasse (standardmäßig `vector<T>`) realisiert — eine `priority_queue<T>` ist somit auch nur ein “Adapter“ (spezielle Schnittstelle) für eine andere Containerklasse. (Diese andere Containerklasse muss aufgrund der Sortierung nach Priorität über *Random-Access-Iteratoren* verfügen sowie über die Funktionen `front()`, `push_back()` und `pop_back()`. i Von den Standard-Containern kommen somit nur `vector<T>` und `deque<T>` in Frage! Darüberhinaus wird dieser zu Grunde liegende Container über entsprechende Algorithmen intern als ein *Heap* — eine zur schnellen Auffindung des größten Elementes besonders geeignete Datenstruktur — verwaltet.)

Als zweites Template-Argument kann wiederum die zu Grunde liegende Containerklasse angegeben werden:

```
// int-priority-queue, mittels vector<int> realisiert
// und < als Vergleichsoperation:
priority_queue<int> pq1;

// int-priority-queue, mittels list<int> realisiert
// und < als Vergleichsoperation:
priority_queue<int, list<int> > pq2;

// Definition einer Vergleichsklasse:
struct vergleich: binary_function<int, int, bool> {
    bool operator(int i, int j) { ... }
};

// int-priority-queue, mittels deque<int> realisiert und
// mit Klasse vergleich als Vergleichsfunktionsobjekttyp:
priority_queue<int, deque<int>, vergleich> pq3;
```

Die wesentlichen `priority_queue`-Operationen

Erzeugt wird eine (zunächst leere) `priority_queue<T>` mit Elementtyp `T` und Namen `puffer` mittels des Konstruktors:

```
priority_queue<T> puffer;
```

(Diese `priority_queue` ist mittels `vector<T>` realisiert und hat `<` als Vergleichsoperation!)

Durch die Funktion

```
puffer.push(elem);
```

wird eine Kopie von `elem` vom Typ `T` aufgenommen und anhand seines Wertes in der `priority_queue` abgespeichert. Diese Funktion wird (indirekt) auf die Funktion `push_back()` der zu Grunde liegenden Containerklasse zurückgeführt.

Die Funktion

```
puffer.top();
```

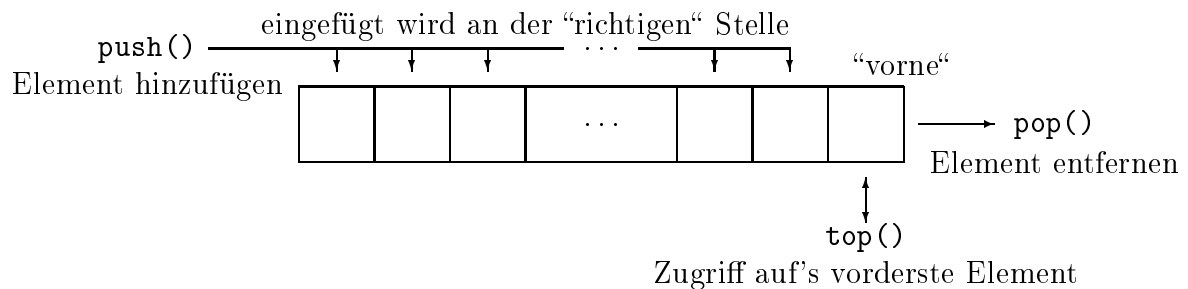
gibt als Ergebnis die **konstante** Referenz auf das vorderste (mit höchster Priorität) Element in `puffer` zurück. Das Element verbleibt in `puffer`. Diese Funktion wird auf die Funktion `front()` der zu Grunde liegenden Containerklasse zurückgeführt.

Die Funktion

```
puffer.pop();
```

entfernt das vorderste (mit höchster Priorität) Element aus `puffer`. Diese Funktion wird (indirekt) auf die Funktion `pop_back()` der zu Grunde liegenden Containerklasse zurückgeführt.

Eine Übersicht über diese Standard-Verwendung einer `priority_queue<T>` gibt folgendes Bild:



Die genaue Schnittstelle zur Klasse `priority_queue<T>`

Die vom Standard vorgegebene Template-Deklaration der Klasse `priority_queue<T>` ist wie folgt:

```
template <class T, class Container = vector<T>,
          class Compare = less< Container::value_type> >
class priority_queue {
public:
    typedef typename Container::value_type    value_type;
    typedef typename Container::size_type     size_type;
    typedef          Container                container_type;
```

```

protected:
    Container c;
    Compare comp;

public:
    explicit priority_queue( const Compare& cmp = Compare(),
                           const Container& c = Container());

    template <class InputIterator>
    priority_queue ( InputIterator anf, InputIterator ende,
                   const Compare& cmp = Compare(),
                   const Container& c = Container());

    bool      empty() const { return c.empty(); }
    size_type size()  const { return c.size(); }

    const value_type& top() const { return c.front(); }

    void push(const value_type& x);
    void pop();
};

```

Erläuterungen:

```

- template <class T, class Container = vector<T>,
           class Compare = less< Container::value_type> >
  class priority_queue { ... };

```

Template-Definition, erstes Argument ist der Elementtyp, zweites Argument der zu Grunde liegende Containertyp (standardmäßig `vector<T>`), drittes Argument der Vergleichsfunktionsobjektyp (standardmäßig: Vergleich des Container-Elementtypes mit <).

```

- typedef typename Container::value_type  value_type;
  typedef typename Container::size_type   size_type;
  typedef          Container              container_type;

```

Eingebettete Typnamen (wie bei `queue<T>`):

`value_type` Elementtyp der `queue<T>` (zurückgeführt auf den Elementtyp des zu Grunde liegenden Containertypes),
`size_type` Größentyp der `queue<T>` (zurückgeführt auf den Größentyp des zu Grunde liegenden Containertypes),
`container_type` Typ des zu Grunde liegenden Containers.

```

- protected:
    Container c;
    Compare comp;

```

Der zu Grunde liegender Container und das Vergleichsfunktionsobjekt sind als `protected`-Komponenten abgespeichert.

- `priority_queue(const Compare& cmp = Compare(),
 const Container& c = Container());`

Standard-Konstruktor, legt eine zunächst leere `priority_queue<T>` an, wobei als Argument das konkrete Vergleichsfunktionsobjekt `cmp` und der konkrete zu Grunde liegende Container angegeben werden können. Defaultmäßig werden diese Argumente mit den entsprechenden Standardkonstruktoren erzeugt!

- Es gibt einen weiteren Konstruktor

```
template <class InputIterator>
priority_queue ( InputIterator anf, InputIterator ende,
                 const Compare& cmp = Compare(),
                 const Container& c = Container());
```

der die `priority_queue<T>` mit den Elementen der durch die Iteratoren gegebenen Sequenz `[anf,ende)` vorbesetzt (Iteratortyp `InputIterator` muss zum Elementtyp passen!).

Auch hier können das konkrete Vergleichsfunktionsobjekt `cmp` und der konkrete zu Grunde liegende Container angegeben werden. Defaultmäßig werden diese mit den entsprechenden Standardkonstruktoren erzeugt!

- `size_type empty() const { return c.empty();}`

liefert, ob die `priority_queue<T>` leer ist.

Diese Funktion wird auf die entsprechende Funktion `empty` der Containerklasse zurückgeführt, welche der `priority_queue<T>` zu Grunde liegt.

- `size_type size() const { return c.size();}`

liefert die aktuelle Anzahl der Elemente.

Diese Funktion wird auf die entsprechende Funktion `size()` der Containerklasse zurückgeführt, welche der `priority_queue<T>` zu Grunde liegt.

- `const value_type& top() const { return c.front();}`

liefert konstante Referenz auf das vorderste Element (das mit der höchsten Priorität). Dieses würde als nächstes (mit `pop`) entfernt werden.

Die `priority_queue<T>` darf nicht leer sein!

Diese Funktion wird auf die entsprechende Funktion `front()` der Containerklasse zurückgeführt, welche der `priority_queue<T>` zu Grunde liegt.

- `void push(const value_type& elem);`

fügt eine Kopie des angegebenen Elementes (an der der Priorität entsprechenden Stelle) in die `priority_queue<T>` ein.

Diese Funktion wird (indirekt) auf die Funktion `push_back()` der Containerklasse zurückgeführt, welche der `priority_queue` zu Grunde liegt.

- `void pop();`
entfernt aus der `priority_queue<T>` das vorderste Element (das mit der höchsten Priorität).
Diese Funktion wird indirekt auf die Funktion `pop_back()` der Containerklasse zurückgeführt, welche der `priority_queue<T>` zu Grunde liegt.
- Es gibt **keine** Vergleichsoperatoren `==` bzw. `<` (und demnach auch nicht die anderen Vergleichsoperatoren!) und keine spezielle Funktion `swap()` zum Vertauschen zweier `priority_queue<T>`'s.
- Es gibt keinen speziellen Zuweisungsoperator, da die Funktionalität des für jede Klasse generierten Standard-Zuweisungsoperators genügt.
- Es gibt keinen speziellen Destruktor, da die Funktionalität des für jede Klasse generierten Standard-Destruktors genügt.

11.4.3 Der Containeradapter `stack<T>`

Die Klasse `stack<T>` stellt einen Kellerspeicher, also einen nach dem *Lifo*-Prinzip (*Last in first out*) funktionierenden Speicherbereich. Das zuletzt (mit der Funktion `push()`) abgespeicherte Element (vom Typ `T`) ist dasjenige, auf das man (mit der Funktion `top`) Zugriff hat und welches (mit der Funktion `pop`) als nächstes aus dem Speicher entfernt wird.

Zur Verwendung von `stack`'s muss die Headerdatei `<stack>` eingebunden werden.

Wie bei `queue<T>`'s und `priority_queue<T>`'s handelt es sich um einen Adapter (besondere Schnittstelle) für eine andere, zu Grunde liegende Containerklasse (standardmäßig: `deque<T>`), welche man wiederum als zweites Template-Argument bei der Definition des `stack`'s angeben kann:

```
stack<int> stack1;           // int-stack, auf deque<int> beruhend
stack<int,list<int> > stack2; // int-stack, auf list<int> beruhend
```

Als zu Grunde liegende Containerklasse kann man jede verwenden, welche über die Funktionen `back()`, `push_back()` und `pop_back()` verfügt. Von den Standard-Containerklassen sind dies `vector<T>`, `deque<T>` und `list<T>`.

Die wesentlichen `stack<T>`-Operationen

Erzeugt wird ein (zunächst leerer) `stack` mit Elementtyp `T` und Namen `puffer` mittels des Konstruktors:

```
stack<T> puffer;
```

(Dieser Stack ist mittels `deque<T>` realisiert!)

Durch die Funktion

```
puffer.push(elem);
```

wird eine Kopie des angegebenen Elementes `elem` (vom Typ `T`) "oben" in den Kellerspeicher eingefügt. (Diese Funktion wird auf die Funktion `push_back()` der zu Grunde liegenden Containerklasse zurückgeführt!)

Die Funktion

```
puffer.top();
```

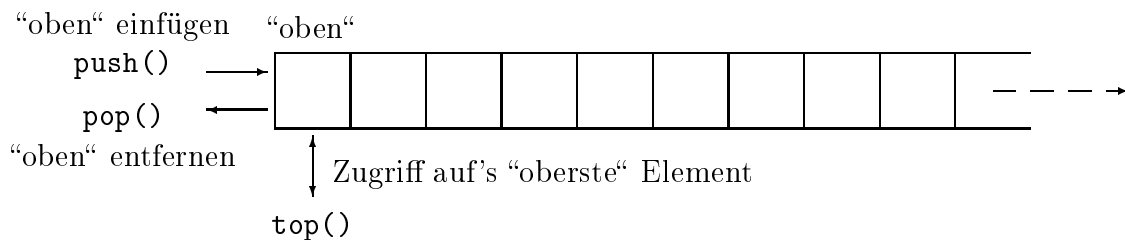
liefert als Ergebnis eine Referenz auf das “oberste” (zuletzt eingefügte) Element des Kellerspeichers. (Diese Funktion wird auf die Funktion `back()` der zu Grunde liegenden Containerklasse zurückgeführt!)

Die Funktion

```
puffer.pop();
```

entfernt das “oberste” (zuletzt eingefügte) Element des Kellerspeichers, die Funktion hat `void`-Rückgabotyp. (Diese Funktion wird auf die Funktion `pop_back()` der zu Grunde liegenden Containerklasse zurückgeführt!)

Die wesentliche Schnittstelle zu einem `stack` ist in folgenden Bild veranschaulicht:



Die genaue Schnittstellen zur `stack<T>`-Klasse

Die vom Standard vorgegebene Template-Deklaration der Klasse `stack<T>` ist wie folgt:

```
template <class T, class Container = deque<T> >
class stack {
public:
    typedef typename Container::value_type value_type;
    typedef typename Container::size_type size_type;
    typedef          Container          container_type;

protected:
    Container c;

public:
    explicit stack( const Container& = Container());

    bool          empty() const { return c.empty(); }
    size_type size() const { return c.size(); }

    value_type&    top()          { return c.back(); }
    const value_type& top() const { return c.back(); }

    void push(const value_type& x) { c.push_back(x); }
    void pop()                    { c.pop_back(); }
};
```

```

template <class T, class Container>
bool operator== (const stack<T,Container>& x, const stack<T,Container>& y);

template <class T, class Container>
bool operator< (const stack<T,Container>& x, const stack<T,Container>& y);

template <class T, class Container>
bool operator!= (const stack<T,Container>& x, const stack<T,Container>& y);

template <class T, class Container>
bool operator> (const stack<T,Container>& x, const stack<T,Container>& y);

template <class T, class Container>
bool operator>= (const stack<T,Container>& x, const stack<T,Container>& y);

template <class T, class Container>
bool operator<= (const stack<T,Container>& x, const stack<T,Container>& y);

```

Erläuterungen:

```

- template <class T, class Container = vector<T> >
  class stack { ... };

```

An dieser Template-Definition sieht man, dass man als zweites Template-Argument die Containerklasse angeben kann, auf die der Container-Adapter `stack<T>` beruhen soll. Standardmäßig wird hierzu die Containerklasse `vector<T>` (mit gleichem Elementtyp) verwendet.

```

- typedef typename Container::value_type  value_type;
  typedef typename Container::size_type   size_type;
  typedef          Container              container_type;

```

Eingebettete Typnamen:

`value_type` Elementtyp des `stack<T>` (zurückgeführt auf den Elementtyp des zu Grunde liegenden Containertypes),
`size_type` Größentyp des `stack<T>` (zurückgeführt auf den Größentyp des zu Grunde liegenden Containertypes),
`container_type` Typ des zu Grunde liegenden Containers.

```

- protected:
  Container c;

```

Der zu Grunde liegender Container ist als `protected`-Komponente abgespeichert.

```

- explicit stack( const Container& = Container() );

```

Standard-Konstruktor, legt einen leere `stack<T>` an. Als Argument kann ein konkreter Container des zu Grunde liegenden Containertypes angegeben werden. Ansonsten wird der zu Grunde liegende Container mittels des entsprechenden Standardkonstruktors erzeugt.

– `bool empty() const { return c.empty(); }`

liefert, ob die `stack` leer ist. (Zurückgeführt auf die Funktion `empty()` des zu Grunde liegenden Containers!)

– `size_type size() const { return c.size(); }`

liefert die aktuelle Anzahl der Elemente. (Zurückgeführt auf die Funktion `size()` des zu Grunde liegenden Containers!)

– `value_type& top() { return c.front(); }`
– `const value_type& top() const { return c.front(); }`

liefert Referenz auf das vorderste (zuerst gespeicherte) Element (die `stack` darf nicht leer sein)! (Zurückgeführt auf die Funktion `front()` des zu Grunde liegenden Containers!)

Bei der konstanten Member-Funktion wird eine Referenz auf ein konstantes Element zurückgegeben.

– `void push(const value_type& x) { c.push_back(x); }`

fügt eine Kopie des angegebenen Elementes (hinten) in die `stack<T>` ein. (Zurückgeführt auf die Funktion `push_back()` des zu Grunde liegenden Containers.)

– `void pop() { c.pop_back(); }`

entfernt das hinterste, zuletzt in den `stack<T>` abgespeicherte Element aus dem `stack<T>`. (Zurückgeführt auf die Funktion `pop_back()` des zu Grunde liegenden Containers.)

– Die üblichen Vergleichsoperatoren werden ebenfalls auf den Vergleich des jeweils zu Grunde liegenden Containers zurückgeführt.

– Es gibt keine spezielle Elementfunktion `swap` zum Vertauschen zweier `stack<T>`'s.

– Es gibt keinen speziellen Zuweisungsoperator, da die Funktionalität des für jede Klasse generierten Standard-Zuweisungsoperators genügt.

– Es gibt keinen speziellen Destruktor, da die Funktionalität des für jede Klasse generierten Standard-Destruktors genügt.

11.5 Bitsets

In der Standardbibliothek ist eine Template-Klasse `bitset<N>` zur Verwaltung einer (festen) Menge von Flaggen (etwa Zuständen *gut/schlecht*, *an/aus* oder *wahr/falsch* ...) vorgesehen.

Der zu Grunde liegende Datentyp ist "Feld" vom Typ `bool` und der Template-Parameter ist die Größe dieses Feldes:

```
template <size_t N>
class bitset {
    ...
};
```

Die Länge eines konkreten Bitsets wird bei seiner Erzeugung über das Template-Argument festgelegt und kann nicht mehr verändert werden. Da das Template-Argument bereits vom Compiler verarbeitet wird, muss das Argument und somit die Größe des Bitsets bereits zur Compilierzeit feststehen — d.h. es muss ein ganzzahliges Literal, eine Präprozessor-Konstante oder eine globale ganzzahlige Konstante sein.

Ein Bitset ist also kein Vektor `vector<bool>` vom Typ `bool`, der bekanntlich dynamisch wachsen könnte — dafür sind für Bitsets aber eine Reihe von Bitoperationen definiert, welche es für einen `vector<bool>` nicht gibt.

Zur Verwendung von Bitsets muss die Headerdatei `<bitset>` includet werden.

11.5.1 Konstruktion und Größe eines Bitset

Ein Ausschnitt aus der Template-Definition von Bitsets:

```
template <size_t N>
class bitset {
    ...
public:
    // Konstruktoren:

    // parameterloser Konstruktor:
    bitset();

    // bitset mit unsigned long initialisieren:
    bitset(unsigned long wert);

    // bitset mit C++-String initialisieren:
    explicit bitset(const string &str, size_type pos=0,
                    size_type len=npos);
    ...
    // Größe des Bitsets
    size_t size();
    ...
};
```

Bei allen Konstruktoren muss die Länge des Bitsets über das Template-Argument für den Bitsettyp angegeben werden:

```
unsigned long ul = 1;
string init_string("1000101110");

bitset<50> flagg1;           // Laenge 50, Standardkonstruktor
bitset<10> flagg2( ul);      // Laenge 10, mit unsigned int initialisiert
bitset<10> flagg3( init_string); // Laenge 10, mit String "1000101110"
...                          // initialisiert
```

Der parameterlose Konstruktor `bitset<N>::bitset()` setzt alle Bits auf 0.

Der `unsigned long`-Konstruktor `bitset<N>::bitset(unsigned long wert)` initialisiert die “ersten” Bits des Bitsets mit dem Bitmuster, mit welchem der angegebene `unsigned int`-Wert intern abgespeichert ist. (Zum Verständnis ist es hilfreich, sich die “ersten” Bits als die am weitesten “rechts” stehenden vorzustellen!)

Sollte das Bitset größer sein, als Bits in einem `unsigned long` vorhanden sind, wird “links” mit 0-Bits aufgefüllt.

Sollte das Bitset kleiner sein, als Bits in einem `unsigned long` vorhanden sind, werden nur die niederwertigen (“rechten”) Bits des Wertes zur Initialisierung genommen.

In obigem Beispiel dürfte das Bitmuster des Bitsets `flagg2` also 0000000001 lauten.

Beim String-Konstruktor

```
bitset<N>::bitset( const string &str, string::size_type pos=0,
                  string::size_type len=npow);
```

wird das Bitset anhand des an Position `pos` beginnenden Teilstrings der Länge `len` des Strings `str` initialisiert.

Der Typ `string::size_type` ist hierbei der zur String-Klasse definierte vorzeichenlose ganzzahlige Typ, mit dem in der String-Klasse Stringposition angegeben werden und die Konstante `npos` ist der ebenfalls in der String-Klasse definierte Wert zur Indikation von *zu großer Index* (vgl. Kapitel 10).

Anstelle eines C++-Strings kann hier auch ein Objekt einer anderen, für eine andere Zeichenart als Spezialisierung der Template-Klasse `basic_string<T>` gewonnenen String-Klasse verwendet werden!

Wie bei Strings üblich, wird eine `out_of_range`-Ausnahme ausgelöst, falls `pos` größer als die Länge des Strings `str.size()` ist und falls `len` zu groß ist, wird nur der an Position `pos` beginnende Rest des Strings `str` genommen.

Die Defaultargumente 0 für `pos` und `npos` für `len` sorgen dafür, dass standardmäßig der ganze String zur Initialisierung verwendet wird.

Jedes Zeichen des zur Initialisierung verwendeten Teils des C++-Strings muss hierbei entweder die Ziffer ‘0’ oder die Ziffer ‘1’ sein — ansonsten wird eine Ausnahme vom im Standard definierten Typ `invalid_argument` ausgelöst.

Sei `n` die Länge des zur Initialisierung verwendeten Teilstrings und `N` die Länge des Bitsets und sei `M = min{n, N}` das Minimum von `n` und `N`, so werden die (von rechts) die ersten `M`-Bits des Bitsets mit den (von links) ersten `M` Zeichen des Strings initialisiert,

wobei das erste Bit (am weitesten “rechts“ stehend, “kleinster“ Index 0 im Bitset) mit dem letzten Zeichen (am weitesten “rechts“ stehend, größter Index $M-1$ im Teilstring) initialisiert wird (aus der Ziffer ‘0’ wird hierbei der Bitwert 0, aus der Ziffer ‘1’ der Bitwert 1), anschließend wird der Index im Bitset um eins erhöht und im Teilstring um eins erniedrigt usw., bis genau M Bits initialisiert wurden.

Ist N größer als M , erhalten die restlichen Bits (“links“) im Bitset jeweils den Wert 0.

Als Copy-Konstruktor ist, da nicht neudefiniert, für Bitsets der standardmäßig definierte verfügbar.

Ein Destruktor ist nicht vorgesehen.

Die Member-Funktion

```
size_t bitset<N>::size();
```

gibt die Größe des Bitsets (Anzahl der Bits, also den Template-Parameter des Bitset-types) als Ergebnis zurück.

11.5.2 Operationen für Bitset

Manipulierende Member-Funktionen für Bitsets

Mit folgenden Member-Operator-Funktionen zur Klasse `bitset<N>` kann man die Bits eines `bitsets` verändern:

```
- bitset<N>& bitset<N>::operator&=( const bitset<N>& arg);
```

Bitweise *UND*-Zuweisung. In

```
bitset<N> a, b;
...
a &= b;
...
```

wird im Bitset `a` jedes Bit gelöscht (auf 0 gesetzt), dessen korrespondierendes Bit in `b` (gleiche Position) den Wert 0 hat. Die restlichen Bits in `a` bleiben unverändert. Ergebnis ist das aktuelle Bitset nach der Zuweisung. (Beide Bitsets müssen den gleichen Typ, also die gleiche Länge haben!)

```
- bitset<N>& bitset<N>::operator|=( const bitset<N>& arg);
```

Bitweise *ODER*-Zuweisung. In

```
bitset<N> a, b;
...
a |= b;
...
```

wird im Bitset `a` jedes Bit auf 1 gesetzt, dessen korrespondierendes Bit in `b` auch den Wert 1 hat. Die restlichen Bits in `a` bleiben unverändert. Ergebnis ist das aktuelle Bitset nach der Zuweisung. (Beide Bitsets müssen den gleichen Typ, also die gleiche Länge haben!)

- `bitset<N>& bitset<N>::operator^=(const bitset<N>& arg);`

Bitweise *EXKLUSIVE ODER*-Zuweisung. In

```
bitset<N> a, b;
...
a ^= b;
...
```

wird im Bitset `a` jedes Bit negiert (von 0 auf 1 gesetzt bzw. umgekehrt), dessen korrespondierendes Bit in `b` den Wert 1 hat. Die restlichen Bits in `a` bleiben unverändert. Ergebnis ist das aktuelle Bitset nach der Zuweisung. (Beide Bitsets müssen den gleichen Typ, also die gleiche Länge haben!)

- `bitset<N>& bitset<N>::operator<<=(size_t pos);`

Shift des Bitmusters des Bitsets um `pos` Positionen nach links, wobei rechts 0-Bits nachgeschoben werden. Ergebnis ist das aktuelle Bitset nach der Verschiebung.

- `bitset<N>& bitset<N>::operator>>=(size_t pos);`

Shift des Bitmusters des Bitsets um `pos` Positionen nach rechts, wobei links 0-Bits nachgeschoben werden. Ergebnis ist das aktuelle Bitset nach der Verschiebung.

- `bitset<N>& bitset<N>::set();`

Setzt alle Bits des Bitsets auf 1 und gibt das Bitset selbst als Ergebnis zurück.

- `bitset<N>& bitset<N>::set(size_t pos, int val = 1);`

Setzt das Bit an Position `pos` des Bitsets auf den gegebenen Wert `val` (hat `val` den Wert 0, so wird das Bit auf 0 gesetzt, hat `val` einen Wert $\neq 0$, so wird das Bit auf 1 gesetzt) und gibt das Bitset selbst als Ergebnis zurück. Falls `pos` zu groß ist, wird eine *out_of_range*-Ausnahme ausgelöst.

- `bitset<N>& bitset<N>::reset();`

Setzt alle Bits des Bitsets auf 0 und gibt das Bitset selbst als Ergebnis zurück.

- `bitset<N>& bitset<N>::reset(size_t pos);`

Setzt das Bit an Position `pos` des Bitsets auf den Wert 0 und gibt das Bitset selbst als Ergebnis zurück. Falls `pos` zu groß ist, wird eine *out_of_range*-Ausnahme ausgelöst.

- `bitset<N>& bitset<N>::flip();`

Komplementiert jedes Bit des Bitsets und gibt das Bitset selbst als Ergebnis zurück.

```
– bitset<N>& bitset<N>::flip(size_t pos);
```

Komplementiert das Bit an Position `pos` des Bitsets und gibt das Bitset selbst als Ergebnis zurück. Falls `pos` zu groß ist, wird eine `out_of_range`-Ausnahme ausgelöst.

Der Operator `~=` ist für Bitsets nicht vorgesehen!

Nicht manipulierende Member-Funktionen für Bitsets

Folgende Member-Funktionen ändern das aktuelle Bitset nicht:

```
– bitset<N> bitset<N>::operator<< (size_t pos) const;
```

liefert das um `pos` Positionen nach links verschobene, rechts mit 0-Bits aufgefüllte Bit-Muster des aktuellen Bitsets als (temporäres) Ergebnis vom Typ `bitset<N>`.

```
– bitset<N> bitset<N>::operator>> (size_t pos) const;
```

liefert das um `pos` Positionen nach rechts verschobene, links mit 0-Bits aufgefüllte Bit-Muster des aktuellen Bitsets als (temporäres) Ergebnis vom Typ `bitset<N>`.

```
– bitset<N> bitset<N>::operator~ (size_t pos) const;
```

liefert das komplementierte Bit-Muster des aktuellen Bitsets als (temporäres) Ergebnis vom Typ `bitset<N>`.

```
– size_t bitset<N>::count() const;
```

liefert die Anzahl der Bits des Bitsets mit Wert 1.

```
– bool bitset<N>::any() const;
```

liefert `true`, falls mindestens ein Bit des Bitsets den Wert 1 hat.

```
– bool bitset<N>::none() const;
```

liefert `true`, falls kein einziges Bit des Bitsets den Wert 1 hat.

```
– bool bitset<N>::test(size_t pos) const;
```

liefert `true`, falls das Bit mit der Position `pos` im Bitset den Wert 1 hat, sonst `false`. Wirft Ausnahme `out_of_range`, falls `pos` zu groß ist.

```
– bool bitset<N>::operator==( const bitset<N> &s) const;
```

testet das aktuelle Bitset mit dem als Argument angegebenen Bitset (der gleichen Länge!) auf Gleichheit.

```
– bool bitset<N>::operator!=( const bitset<N> &s) const;
```

testet das aktuelle Bitset mit dem als Argument angegebenen Bitset (der gleichen Länge!) auf Ungleichheit.

Typumwandlungen für Bitsets

Die Member-Funktion

```
unsigned long bitset<N>::to_ulong() const;
```

gibt, falls möglich, das in ein `unsigned long` umgewandelte Bitmuster des Bitsets zurück. Falls der resultierende Wert zu groß für ein `unsigned long` ist, wird eine `overflow_error`-Ausnahme ausgelöst.

Die Member-Funktion

```
string bitset<N>::to_string() const;
```

gibt einen C++-String mit dem Bitmuster als Ziffernfolge mit den Ziffern '0' und '1' zurück. (Die Umwandlungsfunktion ist invers zum entsprechenden Konstruktor!) Diese Umwandlung `to_string` ist selbst ein Template, so dass als Ergebnis auch ein Objekt einer anderen Spezialisierung von `basic_string<T>` zurückgegeben werden kann.

Globale Funktionen für Bitsets

Zusätzlich zu den Member-Operationen gibt es folgende global definierte Operationen:

```
- bitset<N> operator& ( const bitset<N>& s, const bitset<N>& t );
```

bildet (als temporäres Ergebnis vom Typ `bitset<N>`) die bitweise *UND*-Verknüpfung der beiden Argumente (vgl. `operator&=`).

```
- bitset<N> operator| ( const bitset<N>& s, const bitset<N>& t );
```

bildet (als temporäres Ergebnis vom Typ `bitset<N>`) die bitweise *ODER*-Verknüpfung der beiden Argumente (vgl. `operator|=`).

```
- bitset<N> operator^ ( const bitset<N>& s, const bitset<N>& t );
```

bildet (als temporäres Ergebnis vom Typ `bitset<N>`) die bitweise *EXKLUSIVE ODER*-Verknüpfung der beiden Argumente (vgl. `operator^=`).

Mittels der durch den Konstruktor `bitset<N>::bitset(unsigned long ul)` definierten Typumwandlung von `unsigned long` nach `bitset` kann bei diesen Operationen einer der Operanden auch den Typ `unsigned long` haben.

Ein- und Ausgabeoperatoren sind ebenfalls definiert:

```
ostream& operator<<( ostream& strm, const bitset<N>& x );
```

wandelt das Bitset mit der Element-Funktion `to_string` in einen String um und gibt diesen aus Ziffern '0' und '1' bestehenden String auf den Ausgabestrom `strm` aus und liefert (wie üblich) den Ausgabestrom als Ergebnis.

```
istream& operator>>(istream& strm, bitset<N>& x);
```

liest bis zu **N** Zeichen vom angegebenen Eingabestrom **strm**, speichert diese in einem temporären String **s** und weist dem Bitset **x** ein mittels des Konstruktors **bitset<N>(s)** aus dem String erzeugtes temporäres Bitset zu. Zurückgegeben wird (wie üblich) der Eingabestrom, ggf. mit gesetzter Fehlerflagge **ios_base::failbit**. Das Lesen von Zeichen endet

- spätestens nach **N** gelesenen Zeichen,
- beim Ende des Eingabestroms (*End of File*),
- vor dem nächsten Zeichen, welches keine Ziffer '0' oder '1' ist.

11.5.3 Der Referenz-Hilfstyp für Bitsets

Um auf ein einzelnes Bit eines Bitsets zugreifen zu können (um diesem etwa einen Wert zuzuweisen), gibt es in der Bitset-Klasse einen eingebetteten Typen **reference** (also eine Referenz auf ein einzelnes Bit!):

```
template <size_t N>
class bitset {
    ...
public:

    // eingebettete Hilfsklasse
    class reference {
    private:
        friend class bitset;
        reference();           // privater Konstruktor!!!
    public:

        ~reference();
        reference& operator=(bool x);
        reference& operator=(const reference& r);
        operator bool() const;
        bool operator~() const;
        reference& flip();
    }; // Ende des Rumpfes der eingebetteten Hilfsklasse

    ...
    reference operator[](size_t pos);
    ...
};
```

Die Tatsache, dass der einzige Konstruktor der Klasse **reference** privat und die Klasse **bitset<N>** befreundet ist, bewirkt, dass nur in Memberfunktionen der Klasse **bitset<N>** eine **reference** erzeugt werden kann!

Die einzige **bitset<N>**-Member-Funktion, die dann eine solche **reference** nach außen weitergibt, ist der Indexzugriff:

```
reference bitset<N>::operator[](size_t pos);
```

die für ein Bitset aufgerufen:

```
bitset<100> flaggs(...)
int i;
...
...flaggs[i]...
...
```

eine Referenz auf das *i*-te Bit des Bitsets zurückgibt.

Dieser Index-Zugriff ist im Standard nicht weiter spezifiziert, ich würde erwarten, dass eine *out_of_range*-Ausnahme ausgeworfen wird, falls der Index *pos* zu groß ist.

Die für den Typ *reference* (als Member-Funktionen) definierten Operationen:

- *reference& operator=(bool x);*
Zuweisung eines booleschen Wertes, Ergebnis ist die aktuelle *reference*,
- *reference& operator=(const reference& r);*
Zuweisung einer anderen *reference*, Ergebnis ist die aktuelle *reference*,
- *bitset<N>::operator bool() const;*
Umwandlung der *reference* nach *bool* (Wert des aktuellen Bits),
- *bool operator~() const;*
Umwandlung der *reference* nach *bool* (negierter Wert des aktuellen Bits) und
- *reference& flip();*
Bitwert der *reference* negieren, Ergebnis ist die aktuelle *reference*

machen somit im Zusammenhang mit dem Indexzugriff *operator[]* für Bitsets folgende Anwendungen möglich:

```
bitset<100> bits(...);
int i, j;
bool flagge;
...
bits[i] = flagge;           // reference::operator=(bool)
bits[i] = bits[j];         // reference::operator=(const reference&)
...
if ( bits[i] ){ ... }      // reference::operator bool()
if ( ~bits[i] ){ ... }     // bool reference::operator~()
...
bits[i].flip();            // reference::flip()
...
```

In den meisten Implementierungen dürfte es auch eine Version des Indexzugriffs für konstante Bitsets geben (vom Standard nicht explizit vorgesehen) :

```
bool bitset<N>::operator[](size_t pos) const;
```

welche den Wert des entsprechenden Bits als Wahrheitswert zurückgibt (`out_of_range`-Ausnahme, falls `pos` zu groß ist).

11.6 Funktionsobjekte

Viele Algorithmen benötigen eine Funktion als Argument — etwa ein Sortieralgorithmus das Sortierkriterium, anhand dessen sortiert werden soll.

Natürlich kann hier eine gewöhnliche Funktion oder auch ein Funktionszeiger als Argument übergeben werden — der Algorithmus läuft dann mit der als Argument spezifizierten Funktion ab.

Wie in Abschnitt 5.2 gesehen, ist es in C++ möglich, eine Klasse zu definieren, für welche der “Funktionsausfrufs-Operator“ `()` überladen ist, etwa:

```
class fkt
{
    ...
public:
    fkt(...);          // Konstruktor
    ...
    int operator() (const char * s1, const char * s2) { ... }
    ...
};
```

Hier wird also zur Klasse `fkt` die Operatorfunktion `operator` mit `int`-Resultat und zwei `const char *`-Parametern definiert.

Ist nun `compare` ein Objekt dieses Types `fkt` — also etwa wie folgt definiert:

```
fkt compare(...);
```

so kann diese Operatorfunktion `fkt::operator()(...)` für das Funktionsobjekt `compare` wie folgt aufgerufen werden:

```
... compare("hallo", "HALLO") ...;
```

wobei dies vom Compiler umgesetzt wird zu:

```
... compare.operator() ("hallo", "HALLO") ...;
```

Dieses Objekt `compare` kann somit wie ein Funktionsname — also wie eine Funktion — verwendet werden! (Deshalb der Name: *Funktionsobjekt*!)

Man muss zwischen dem Funktionsobjekttyp `class fkt` und dem eigentlichen Funktionsobjekt `compare` unterscheiden!

Schreibt man nun eine Funktion (i. Allg. eine Template-Funktion, denn bei Templates kommt es nicht auf den exakten Typ eines Argumentes an!), etwa zum Sortieren von Zeichenketten, so kann man das Vergleichskriterium als Template-Parameter angeben:

```
template <class Vergleich>
... sort( ..., Vergleich op)
{
    ...
    ... op( string1, string2);
    ...
}
```

Man kann dann bei der Konkretisierung dieser Template-Funktion eine “normale” Funktion (oder einen Funktionszeiger) angeben:

```
... sort( ..., strcmp);
```

bei dieser Instanziierung der Template-Funktion bekommt der Template-Typparameter `Vergleich` implizit den Wert `int (*)(char*, char*)` (nämlich den Typ von `strcmp`) und der Template-interne Aufruf `op(string1, string2)` wird umgesetzt zu

```
strcmp(string1, string2).
```

Man kann aber auch ein derartiges Funktionsobjekt `fkt compare`; angeben:

```
... sort( ..., compare);
```

bei dieser Instanziierung bekommt der Template-Typparameter `Vergleich` implizit den Wert `class fkt` (nämlich den Funktionsobjekttyp des Funktionsobjektes `compare`) und der Template-interne Aufruf `op(string1, string2)` wird umgesetzt zu

```
compare(string1, string2)
```

was aber vom Compiler zu

```
compare.operator() (string1, string2)
```

erweitert wird. In diesem Fall wird also zu dem Funktionsobjekt `compare` die Operatorfunktion `operator()` mit den angegebenen Argumenten aufgerufen.

Bei einer expliziten Instanziierung der Template-Funktion muss in den spitzen Klammern des Templates der entsprechende Typ (Funktionstyp oder Funktionsobjekttyp) angegeben werden:

```
...sort< int (*)(char *, char *) > ( ..., strcmp);
...sort< fkt >(..., compare);
```

Die am meisten verwendeten Funktionsobjekte “haben zwei oder einen Parameter” — man kann aber Funktionsobjekte mit “beliebiger Parameterzahl” vereinbaren. (“*n* Parameter haben” bedeutet: das Funktionsobjekt kann wie eine Funktion mit *n* Parametern aufgerufen werden!).

Standard-Parameter und Überladung sind hierbei auch möglich.

Bei einigen Anwendungen von Funktionsobjekten bei Template-Klassen in der Standardbibliothek muss man echte Funktionsobjekttypen angeben, da innerhalb der Template-Klasse mittels eines Konstruktors aus dem Funktionsobjekttyp (ein oder mehrere) konkrete Funktionsobjekte erzeugt werden — in diesem Fall kann man keine “normalen” Funktionstypen verwenden (diese Typen sind keine Klassen und haben keinen Konstruktor!).

11.6.1 Basisklassen zu Standard-Funktionsobjekten

In der Standardbibliothek (genauer, in der Headerdatei `<functional>`) gibt es eine Reihe von vordefinierten Template-Funktionsobjekten (also für unterschiedlichste Typen verfügbar) — die meisten von ihnen “haben zwei Parameter“, die anderen “haben einen Parameter“.

Die mit zwei Parametern sind alle von der Klasse:

```
template <class T1, class T2, class T3>
struct binary_function {
    typedef T1 first_argument_type;
    typedef T2 second_argument_type;
    typedef T3 result_type;
};
```

abgeleitet und die mit einem Parameter von

```
template <class T1, class T2>
struct unary_function {
    typedef T1 first_argument_type;
    typedef T2 result_type;
};
```

Hierdurch sind die Gemeinsamkeiten aller binären Funktionen (nämlich zwei Argumente von möglicherweise unterschiedlichem Typ und ein Ergebnistyp) und aller unären Funktionen (nämlich ein Argument und ein Ergebnistyp) in einer entsprechenden Klasse zusammengefasst und unterschiedliche derartige Funktionen können mittels gewisser Techniken gleich behandelt werden. (Leitet man von diesen Klassen eigene Funktionsobjekte ab, so können auch diese wie die vom Standard vorgegebenen verwendet werden!)

11.6.2 Standard-Operatoren als Funktionsobjekte

Im Standard (Headerdatei `<functional>`) sind einige Templates definiert, die zu den standardisierten arithmetischen Operationen entsprechende Funktionsobjekte erzeugen, welche man ggf. an Algorithmen übergeben kann:

Name		Wirkung	Name		Wirkung
plus	binär	$\text{arg1} + \text{arg2}$	minus	binär	$\text{arg1} - \text{arg2}$
multiplies	binär	$\text{arg1} * \text{arg2}$	divides	binär	$\text{arg1} / \text{arg2}$
modulus	binär	$\text{arg1} \% \text{arg2}$	negate	unär	$-\text{arg}$

Ist beispielsweise `T` ein Datentyp, für den die Multiplikation mittels `*` definiert ist, so ist

```
multiplies<T> mul;
```

ein (binäres) Funktionsobjekt (also wie eine Funktion mit zwei Argumenten zu verwenden) und der “Aufruf“

```
mul(elem1, elem2)
```

für zwei Objekte `elem1` und `elem2` vom Typ `T` wird (durch das Template) umgesetzt zu

`elem1*elem2`.

Nochmals zur Verdeutlichung: `multiplies<T>` ist der Funktionsobjekttyp und `mul` ist das Funktionsobjekt!

11.6.3 Prädikate

Ein Prädikat ist ein Funktionsobjekt mit einem Ergebnis vom Typ `bool`.

Man muss zwischen einstelligen (mit einem Argument) und zweistelligen (mit zwei Argumenten) Prädikaten unterscheiden.

Auch hier gibt es in der Standardbibliothek (Headerdatei `<functional>`) Templates, mit denen man aus den Standardoperatoren, welche einen Wahrheitswert liefern, entsprechende Prädikate erzeugt, welche man ggf. an Algorithmen übergeben kann:

Name		Wirkung	Name		Wirkung
<code>equal_to</code>	binär	<code>arg1 == arg2</code>	<code>not_equal_to</code>	binär	<code>arg1 != arg2</code>
<code>greater</code>	binär	<code>arg1 > arg2</code>	<code>less</code>	binär	<code>arg1 < arg2</code>
<code>greater_equal</code>	binär	<code>arg1 >= arg2</code>	<code>less_equal</code>	binär	<code>arg1 <= arg2</code>
<code>logical_and</code>	binär	<code>arg1 && arg2</code>	<code>logical_or</code>	binär	<code>arg1 arg2</code>
<code>logical_not</code>	unär	<code>!arg</code>			

Ist beispielsweise `T` ein Datentyp, für den der Vergleichsoperator `<` definiert ist, so ist `less<T>` kleiner;

ein Funktionsobjekt (also wie eine Funktion zu verwenden) und der “Aufruf“

`kleiner(elem1, elem2)`

für zwei Objekte `elem1` und `elem2` vom Typ `T` wird (durch das Template) umgesetzt zu

`elem1<elem2`.

Bei einigen Algorithmen, etwa zur Sortierung, wird eine Vergleichsoperation oder eine Vergleichsfunktion zugrundegelegt.

Ein solcher Vergleich (mit einer Funktion oder einem Operator) kann auch als binäres Prädikat angesehen werden, doch ist nicht jedes binäre Prädikat als Vergleich geeignet — es muss im Allgemeinen zusätzlich einige Eigenschaften erfüllen, siehe etwa Abschnitt 11.7.5.

11.6.4 Binder, Funktionsadapter, Negierer

Gegenstand dieses Abschnitts sind einige Templates aus der Standardbibliothek (Headerdatei `<functional>`), mit denen man aus gewissen Funktionen bzw. Funktionsobjekten sich etwas anders verhaltende Funktionen bzw. Funktionsobjekte erzeugen kann.

Binder

Die beiden Template-Funktionen `bind1st` und `bind2nd` “erzeugen“ jeweils aus einem Objekt der Klasse `binary_function` ein Objekt der Klasse `unary_function`.

Sei `BinOp` ein `binary_function` mit `T1` und `T2` als erstem bzw. zweiten Argumenttyp und einem Ergebnis vom Typ `T`.

1. Ist `elem1` ein Objekt vom Typ `T1`, so ist

```
bind1st( BinOp, elem1)
```

ein unäres Funktionsobjekt vom Typ `unary_function<T2,T>` (d.h. ein Argument vom Typ `T2` und Ergebnis vom Typ `T`).

Der Ausdruck `bind1st(BinOp, elem1)` kann etwa einem Algorithmus als unäres Funktionsobjekt (Argument vom Typ `T2` und Ergebnis vom Typ `T`) übergeben werden und wann immer innerhalb des Algorithmus diese “Funktion“ für ein Argument `obj` vom Typ `T2` aufgerufen wird, wird dieser Aufruf intern umgesetzt zu:

```
BinOp( elem1, obj)
```

D.h. `bind1st` macht aus der binären Funktion `BinOp` eine unäre Funktion, indem als erstes Argument in `BinOp` immer ein- und dasselbe Objekt, nämlich `elem1` eingesetzt wird! (Beim Aufruf `bind1st(BinOp, elem1)` wird also das Objekt `elem1` als erstes Argument an die binäre Funktion `BinOp` “gebunden“, so dass nur noch ein “freier“ Parameter — vom Typ `T2` — übrig bleibt!)

2. Ist `elem2` ein Objekt vom Typ `T2`, so ist

```
bind2nd( BinOp, elem2)
```

ein unäres Funktionsobjekt vom Typ `unary_function<T1,T>` (d.h. ein Argument vom Typ `T1` und Ergebnis vom Typ `T`).

Der Ausdruck `bind2nd(BinOp, elem2)` kann etwa einem Algorithmus als unäres Funktionsobjekt (Argument vom Typ `T1` und Ergebnis vom Typ `T`) übergeben werden und wann immer innerhalb des Algorithmus diese “Funktion“ für ein Argument `obj` vom Typ `T1` aufgerufen wird, wird dieser Aufruf intern umgesetzt zu:

```
BinOp( obj, elem2)
```

D.h. `bind2nd` macht aus der binären Funktion `BinOp` eine unäre Funktion, indem als zweites Argument in `BinOp` immer ein- und dasselbe Objekt, nämlich `elem2` eingesetzt wird! (Beim Aufruf `bind2nd(BinOp, elem2)` wird also das Objekt `elem2` als zweites Argument an die binäre Funktion `BinOp` “gebunden“, so dass nur noch ein “freier“ Parameter — vom Typ `T1` — übrig bleibt!)

Funktionsadapter

Der sogenannten *Funktionszeigeradapter* macht aus gewöhnlichen (unären oder binären) Funktionen bzw. Funktionszeigern Funktionsobjekte, also Objekte vom Typ `unary_function` bzw. `binary_function` (damit sind die Algorithmen, welche auf Komponenten dieser Klassen `unary_function` bzw. `binary_function` beruhen, auch mit gewöhnlichen Funktionen bzw. Funktionszeigern aufrufbar!).

Ist `fkt` eine unäre Funktion (oder ein Zeiger auf eine solche Funktion) mit Argument vom Typ `T1` und Ergebnis vom Typ `T`, so ist `ptr_fun(fkt)` ein Objekt der Klasse `unary_function<T1,T>` mit gleicher Funktionalität, d.h. der “Aufruf“ des Funktionsobjektes wird auf den Aufruf der entsprechenden Funktion `fkt` zurückgeführt!

Ist `fkt` eine binäre Funktion (oder ein Zeiger auf eine solche Funktion) mit zwei Argumenten, erstes Argument vom Typ `T1`, zweites Argument vom Typ `T2` und Ergebnis vom Typ `T`, so ist `ptr_fun(fkt)` ein Objekt der Klasse `binary_function<T1,T2,T>` mit gleicher Funktionalität, d.h. der “Aufruf“ des Funktionsobjektes wird auf den Aufruf der entsprechenden Funktion `fkt` zurückgeführt!

Beispiel:

Ist ein Algorithmus wie folgt definiert (und entsprechend deklariert):

```
template <class T1, class T2, class T>
... algorithmus( ..., binary_function<T1, T2, T> op)
{ T1 arg1;
  T2 arg2;
  T3 erg;
  ...
  erg = op( arg1, arg2);
  ...
}
```

und will man den Algorithmus für die Datentypen `T1=T2=char *` und `T=int` spezialisieren, so muss beim Aufruf des Algorithmus als Funktions-Argument ein Objekt vom Typ `binary_function<char *, char *, int>` angeben.

Als Argument kann etwa **nicht** die C-Bibliotheks-Funktion `strcmp` angegeben werden:

```
...algorithmus(..., strcmp); // Fehler: strcmp keine binary_function!
```

denn obwohl sie richtige Ergebnis- und Argumenttypen besitzt, ist `strcmp` nun mal kein Objekt der Klasse `binary_function<char *, char *, int>`.

Der Algorithmus kann aber wie folgt aufgerufen werden:

```
...algorithmus(..., ptr_fun(strcmp) ); // ok
```

wobei der `algorithmus`-interne Aufruf

```
op( arg1, arg2)
```

umgesetzt wird zu

```
strcmp(arg1, arg2)
```

Die sogenannten *Elementfunktionsadapter* `mem_fun` und `mem_fun_ref` erzeugen aus (einer argumentlosen bzw. einargumentigen) Elementfunktion einer Klasse wiederum

Funktionsobjekte vom Typ `unary_function` bzw. `binary_function`, so dass auch eine Elementfunktion (mit keinem oder einem zusätzlichen Argument) als Argument bei der Instantiierung eines durch ein Template realisierten Algorithmus angegeben werden kann.

Es ist zu beachten, dass eine Elementfunktion `e_fkt(...)` einer Klasse `K` immer in der Form `obj.e_fkt(...)` bzw. `p->e_fkt(...)` aufgerufen wird, wobei `obj` ein Objekt der Klasse `K` bzw. `p` ein Zeiger auf ein `K`-Objekt ist! Somit können solche Elementfunktionen nicht ohne weiteres als Funktionsobjekt einem Algorithmus übergeben werden, da Funktionsobjekte innerhalb des Algorithmus in der “normalen Funktionsaufrufsform“ `fkt(...)` aufgerufen werden!

Zur genauen Beschreibung des Verhaltens von `mem_fun` bzw. `mem_fun_ref` und der Typen der von ihnen erzeugten Funktionsobjekte muss etwas präziser auf die Typen der Elementfunktionen eingegangen werden:

Eine Klasse sei wie folgt definiert:

```
class K {
    ...
public:
    T e_fkt0();           // Elementfunktion ohne Argument
    T e_fkt1(A);          // Elementfunktion mit einem Argument vom Typ A
    ...
};
```

Die Funktion `e_fkt0` erhält bei ihrem Aufruf `obj.e_fkt0()` (`obj` ein Objekt der Klasse `K`) kein explizites Argument, hat aber als implizites Argument das Objekt `obj` selbst! Ergebnistyp ist `T`.

1. Der Template-Aufruf:

```
men_fun( & K::e_fkt0 )
```

“macht“ aus `K::e_fkt0` (also der Elementfunktion `e_fkt0` der Klasse `K`) ein unäres Funktionsobjekt des Types

```
unary_function<K*,T>
```

also eine “gewöhnliche aufzurufende unäre Funktion“ mit einem (expliziten) Argument vom Typ `K*` (also Zeiger auf Klassenobjekt) und Ergebnis vom Typ `T`. Der Aufruf dieses Funktionsobjektes mit einem Zeiger `p` auf ein `K`-Objekt als Argument wird hierbei umgesetzt zu

```
p->e_fkt0()
```

d.h. der “Aufruf“ des Funktionsobjektes (mit Zeigerargument) bewirkt den Aufruf der Elementfunktion (über einen Zeiger auf das Klassenobjekt)!

2. Der Template-Aufruf:

```
men_fun_ref( & K::e_fkt0 )
```

“macht“ aus `K::e_fkt0` (also der Elementfunktion `e_fkt0` der Klasse `K`) ein unäres Funktionsobjekt des Types

`unary_function<K,T>`

also eine “gewöhnlich aufzurufende unäre Funktion“ mit einem Argument vom Typ `K` (also Klassenobjekt) und Ergebnis vom Typ `T`. Der Aufruf dieses Funktionsobjektes

mit einem `K`-Objekt `obj` als Argument wird hierbei umgesetzt zu

`obj.e_fkt0()`

d.h. der “Aufruf“ des Funktionsobjektes (mit `K`-Argument) bewirkt den Aufruf der Elementfunktion (über das Klassenobjekt)!

Bei der Funktion `e_fkt1(A)` verhält es sich ähnlich! Die Funktion `e_fkt1` erhält bei ihrem Aufruf `obj.e_fkt0(a)` (`obj` ein Objekt der Klasse `K`) ein explizites Argument `a` vom Typ `A` und hat als zusätzliches implizites Argument das Objekt `obj` selbst! Ergebnistyp ist `T`.

1. Der Template-Aufruf:

`men_fun(& K::e_fkt1)`

“macht“ aus `K::e_fkt1` (also der Elementfunktion `e_fkt1` der Klasse `K`) ein binäres Funktionsobjekt des Types

`binary_function<K*,A,T>`

also eine “gewöhnliche aufzurufende, binäre Funktion“ mit erstem Argument vom Typ `K*` (also Zeiger auf Klassenobjekt), zweitem Argument vom Typ `A` und Ergebnis vom Typ `T`. Der Aufruf dieses Funktionsobjektes mit einem Zeiger `p` auf ein `K`-Objekt und ein `A`-Element `a` als Argument wird hierbei umgesetzt zu

`p->e_fkt0(a)`

d.h. der “Aufruf“ des Funktionsobjektes (mit Zeigerargument `p` und weiterem Argument `a`) bewirkt den Aufruf der Elementfunktion (über einen Zeiger auf das Klassenobjekt) und Argument `a`!

2. Der Template-Aufruf:

`men_fun_ref(& K::e_fkt1)`

“macht“ aus `K::e_fkt1` (also der Elementfunktion `e_fkt0` der Klasse `K`) ein binäres Funktionsobjekt des Types

`binary_function<K,A,T>`

also eine “gewöhnliche aufzurufende, binäre Funktion“ mit erstem Argument vom Typ `K` (also Klassenobjekt), zweitem Argument vom Typ `A` und Ergebnis vom Typ `T`. Der Aufruf dieses Funktionsobjektes mit einem `K`-Objekt `obj` und einem `A`-Element `a` als Argument wird hierbei umgesetzt zu

`obj.e_fkt0(a)`

d.h. der “Aufruf“ des Funktionsobjektes (mit `K`-Argument) bewirkt den Aufruf der Elementfunktion (über das Klassenobjekt) und Argument `a`!

Negierer

Die Negierer `not1` bzw. `not2` negieren einstellige bzw. zweistellige Prädikate.

D.h. sie machen aus Funktionsobjekten vom Typ `unary_function<T,bool>` bzw. `binary_function<T1,T2,bool>` gleichartige Funktionsobjekte (vom gleichen Typ), bei denen jedoch als Funktionsergebnis jeweils der andere (negierte) Wahrheitswert geliefert wird!

So ist beispielsweise

```
equal_to<double> gleich
```

ein zweistelliges Prädikat vom Typ

```
binary_function<double,double,bool>
```

welches zwei Argumente vom Typ `double` mittels des Operators `==` vergleicht (siehe Abschnitt 11.6.3), dieses Funktionsobjekt kann wie folgt aufgerufen werden:

```
double x, y;
...
... gleich(x,y); // Aufruf des Funktionsobjektes
...
```

Die Anwendung des Negierers

```
not2(gleich)
```

erzeugt aus diesem Funktionsobjekt `gleich` ein neues Funktionsobjekt vom gleichen Typ

```
binary_function<double,double,bool>
```

das neue Funktionsobjekt liefert aber jeweils die zu `gleich` negierten Ergebnisse, d.h. in den Fällen, wo `gleich(x,y)` zu zwei `double`-Werten `x` und `y` den Wahrheitswert `true` liefert, liefert `not2(gleich)(x,y)` den Wahrheitswert `false` und umgekehrt! (Somit ist das Funktionsobjekt `not2(gleich)` gleichwertig zum Prädikat `not_equal_to<double>`, siehe Abschnitt 11.6.3.)

11.7 Algorithmen

In den in Abschnitt 11.3 beschriebenen Containerklassen werden Elemente auf unterschiedlichste Art abgespeichert und verwaltet.

In vielen Problemstellungen der EDV müssen derartig abgespeicherte Elemente bearbeitet, etwa

- eine Operation/Funktion für jedes Element ausgeführt,
- ein(alle) Element(e) mit einer gewissen Eigenschaft ermittelt,
- sortiert,
- Teilmengen erstellt,
- gewisse Elemente gelöscht

– ...

werden.

Für viele derartige Problemstellungen kommt es nicht auf die interne Abspeicherung der Elemente an, sondern es ist erforderlich, der Reihe nach auf die einzelnen Elemente zugreifen zu können.

Genau diesen sequentiellen Zugriff auf Elemente eines Standardcontainers bieten jedoch die Iteratoren. (Iteratoren sind für Containeradapter, wie bereits früher erwähnt, nicht definiert! Aus diesem Grund sind die Algorithmen nicht für Containeradapter verwendbar!)

So sind im Standard eine ganze Reihe von Algorithmen (als Templates) definiert, die eine (oder mehrere) durch Iteratoren `[anf, end)` (siehe Abschnitt 11.2) gegebene Sequenz(en) von Elementen bearbeiten.

Ist etwa `[anf, end)` eine Sequenz von Elementen irgendeines Types, etwa Typ `T`, und `elem` ein Objekt vom Typ `T`, so sucht beispielsweise der Algorithmus

```
find(anf, end, elem)
```

in der Sequenz nach dem ersten Element, welches mit `elem` übereinstimmt. Bei einem Treffer wird die Iteratorposition auf den Treffer zurückgegeben, ansonsten wird `end` zurückgeliefert zur Indikation, dass kein Treffer vorliegt.

Der Algorithmus

```
count(anf, end, elem)
```

gibt als Ergebnis die Anzahl der Elemente zurück, deren Werte mit `elem` übereinstimmen.

Bei derartigen Algorithmen kommt es gar nicht auf die eventuell dahinterstehende Containerklasse an, sondern nur auf den Iteratorzugriff — sie können somit für beliebige Container oder auch auf Teile eines Containers angewendet werden:

```
#include <vector>
#include <list>
#include <algorithm>    // fuer Algorithmen erforderlich

int erg;
vector<int> iv;
...
// ermittle, wie oft der Wert 7 im Vektor iv vorkommt
erg = count( iv.begin(), iv.end(), 7);
...
list<double> dl;
double x;
...
//ermittle, wie oft der Wert x in der Liste dl vorkommt
erg = count( dl.begin(), iv.end(), x);
...
// finde erste Position des Wertes 7 im Vektor iv
vector<int>::iterator iter1, iter2;
iter1 = find( iv.begin(), iv.end(), 7);
```

```
//falls gefunden, ermittle Position des naechsten Treffers
if ( iter != iv.end())
{ // ++iter1 zeigt hinter den ersten Treffer
  iter2 = find (++iter1, iv.end(), 7);
  ...
}
...
```

Zur Verwendung dieser Algorithmen muss (wie im obigen Fragment geschehen) die Headerdatei `<algorithm>` eingebunden werden.

11.7.1 Übersicht über die Algorithmen der Standardbibliothek

In diesem Abschnitt werden tabellarisch nur die Namen und eine umgangssprachliche Beschreibung der Algorithmen angegeben. Hier kann man somit (hoffentlich) einen Einblick in die Mächtigkeit dieser Algorithmen gewinnen — Details (insbesondere Argumente) sind in den folgenden Abschnitten zu finden.

Nichtmodifizierende Algorithmen für Sequenzen

(Details sind im Abschnitt 11.7.3 zu finden!)

<code>for_each()</code>	für jedes Element eine (das Element nicht ändernde) Operation durchführen
<code>find()</code>	erstes Auftreten eines gewissen Wertes finden
<code>find_if()</code>	erstes Element finden, welches ein gewisses Prädikat erfüllt
<code>find_first_of()</code>	irgendeinen Wert einer Sequenz in einer anderen Sequenz finden
<code>adjacent_find()</code>	benachbarte gleiche Elemente finden
<code>count()</code>	Häufigkeit eines gewissen Wertes ermitteln
<code>count_if()</code>	Häufigkeit ermitteln, wie oft ein gewisses Prädikat erfüllt ist
<code>mismatch()</code>	erstes Element einer Sequenz finden, welches nicht mit dem entsprechenden Element einer anderen Sequenz übereinstimmt
<code>equal()</code>	ermitteln, ob in zwei Sequenzen die gleichen Elemente in gleicher Reihenfolge vorhanden sind
<code>search()</code>	erstes Auftreten einer Teilsequenz finden
<code>find_end()</code>	letztes Auftreten einer Teilsequenz finden
<code>search_n()</code>	erste Teilsequenz mit <code>n</code> gleichen Werten finden

Modifizierende Algorithmen für Sequenzen

(Details sind im Abschnitt 11.7.4 zu finden!)

<code>transform()</code>	für jedes Element eine (das Element ändernde) Operation durchführen
<code>copy()</code>	Sequenz vorwärts kopieren
<code>copy_backward()</code>	Sequenz rückwärts kopieren
<code>swap()</code>	zwei Elemente vertauschen
<code>iter_swap()</code>	zwei über Iteratoren gegebene Elemente vertauschen
<code>swap_ranges()</code>	Elemente zweier Sequenzen vertauschen
<code>replace()</code>	Elemente mit einem gewissen Wert ersetzen
<code>replace_if()</code>	Elemente, die ein gewisses Prädikat erfüllen, ersetzen
<code>replace_copy()</code>	Elemente kopieren und dabei alle Elemente mit einem gewissen Wert ersetzen
<code>replace_copy_if()</code>	Elemente kopieren und dabei alle Elemente, die ein gewisses Prädikat erfüllen, ersetzen
<code>fill()</code>	alle Elemente durch einen gewissen Wert ersetzen
<code>fill_n()</code>	<i>n</i> Elemente durch einen gewissen Wert ersetzen
<code>generate()</code>	alle Elemente durch das Ergebnis einer (für jedes Element erneut ausgeführten) Operation ersetzen
<code>generate_n()</code>	<i>n</i> Elemente durch das Ergebnis einer (für jedes Element erneut ausgeführten) Operation ersetzen
<code>remove()</code>	alle Elemente mit einem gewissen Wert entfernen
<code>remove_if()</code>	alle Elemente, die ein gewisses Prädikat erfüllen, entfernen
<code>remove_copy()</code>	Elemente kopieren und dabei Elemente mit einem gewissen Wert fortlassen
<code>remove_copy_if()</code>	Elemente kopieren und dabei Elemente, die ein gewisses Prädikat erfüllen, fortlassen
<code>unique()</code>	aufeinanderfolgende Duplikate entfernen
<code>unique_copy()</code>	Elemente kopieren und dabei aufeinanderfolgende Duplikate fortlassen
<code>reverse()</code>	Reihenfolge der Elemente umdrehen
<code>reverse_copy()</code>	Elemente in umgekehrter Reihenfolge kopieren
<code>rotate()</code>	Elemente rotieren
<code>rotate_copy()</code>	Elemente in rotierter Reihenfolge kopieren
<code>random_shuffle()</code>	Reihenfolge der Elemente ändern (Mischen)
<code>partition()</code>	Elemente, die ein gewisses Prädikat erfüllen, nach vorne platzieren
<code>stable_partition()</code>	wie <code>partition()</code> , Elemente, welche das Prädikat erfüllen, nach vorne platzieren, bleiben aber untereinander in gleicher Reihenfolge

Algorithmen und Sortierung

(Details sind in Abschnitt 11.7.5 zu finden!)

<code>sort()</code>	Sequenz sortieren
<code>stable_sort()</code>	Sequenz stabil sortieren (gleichwertige Elemente behalten untereinander ihre Reihenfolge)
<code>partial_sort()</code>	ersten Teil einer Sequenz sortieren
<code>partial_sort_copy()</code>	Sequenz sortieren und ersten Teil kopieren
<code>nth_element()</code>	das n -te Element an die richtige Stelle sortieren
<code>lower_bound()</code>	erste Position finden, in der ein gewisses Element in eine sortierte Sequenz eingefügt werden könnte
<code>upper_bound()</code>	letzte Position finden, in der ein gewisses Element in eine sortierte Sequenz eingefügt werden könnte
<code>equal_range()</code>	Teilsequenz mit zu einem gewissen Wert gleichwertigen Elementen in einer sortierten Sequenz finden
<code>binary_search()</code>	gewissen Wert in einer sortierten Sequenz finden
<code>merge()</code>	zwei sortierte Sequenzen zu einer sortierten Sequenz vereinigen
<code>inplace_merge()</code>	zwei hintereinanderliegende sortierte Sequenzen zu einer sortierten Sequenz mischen
<code>next_permutation()</code>	nächste Permutation in lexikographischer Reihenfolge ermitteln
<code>prev_permutation()</code>	vorherige Permutation in lexikographischer Reihenfolge ermitteln

Mengen-Algorithmen

(Details sind im Abschnitt 11.7.6 zu finden!)

<code>includes()</code>	feststellen, ob eine Sequenz Teil einer anderen ist
<code>set_union()</code>	erzeugt zu zwei als Sequenzen gegebenen Mengen deren sortierte Vereinigungsmenge
<code>set_intersection()</code>	erzeugt zu zwei als Sequenzen gegebenen Mengen deren sortierte Schnittmenge
<code>set_difference()</code>	erzeugt zu zwei als Sequenzen gegebenen Mengen die sortierte Menge der Elemente, welche in der ersten, aber nicht in der zweiten Menge enthalten sind
<code>set_symmetric_difference()</code>	erzeugt zu zwei als Sequenzen gegebenen Mengen die sortierte Menge der Elemente, welche in einer Menge, aber nicht in beiden Mengen enthalten sind

Algorithmen für Heaps

Ein Heap ist eine Datenstruktur, welche es ermöglicht, das größte Element schnell aufzufinden bzw. zu entfernen und ein neues Element schnell einzufügen.

(Details sind in Abschnitt 11.7.7 zu finden!)

<code>make_heap()</code>	aus Sequenz einen Heap machen
<code>push_heap()</code>	ein Element zum Heap hinzufügen
<code>pop_heap()</code>	größtes Element aus dem Heap entfernen
<code>sort_heap()</code>	Heap sortieren (ist anschließend kein Heap mehr!)

Minimum, Maximum und Vergleich

(Details sind in Abschnitt 11.7.8 zu finden!)

<code>min()</code>	kleineren von zwei Werten ermitteln
<code>max()</code>	größeren von zwei Werten ermitteln
<code>min_element()</code>	kleinste Element einer Sequenz ermitteln
<code>max_element()</code>	größtes Element einer Sequenz ermitteln
<code>lexicographical_compare()</code>	zwei Sequenzen lexikographisch vergleichen

11.7.2 Gemeinsame Bezeichnung für alle Algorithmen

In den folgenden Abschnitten werden die Algorithmen der Standardbibliothek erläutert.

Die Algorithmen sind als Templates realisiert, so dass von den eigentlichen Typen abstrahiert wird und es nur auf die Funktionalität ankommt.

Der Zugriff auf die konkreten Elemente erfolgt über Iteratoren und ggf. werden im Algorithmus andere Funktionen (Funktionsobjekte, Prädikate, Vergleichsfunktionen) aufgerufen.

Im Allgemeinen müssen diese Iteratoren und ggf. Funktionsobjekte als Template-Argumente bei der konkreten Instantiierung des Algorithmus angegeben werden.

Unterschiedliche Algorithmen benötigen unterschiedliche Arten von Iteratoren bzw. Funktionsobjekten.

Zur Vereinfachung gelten in den nächsten Abschnitten für Iteratoren folgende Schreibweisen:

- **InIt:**
Bezeichnung für Input-Iterator-Typ.
- **OutIt:**
Bezeichnung für Output-Iterator-Typ.
- **ForIt:**
Bezeichnung für Vorwärts-Iterator-Typ.
- **BiIt:**
Bezeichnung für Bidirektionalen-Iterator-Typ.

- **RanIt**:
Bezeichnung für Random-Access-Iterator-Typ.

Durch Iteratoren werden im Allgemeinen eine Sequenz von Objekten eines gewissen Types beschrieben. Es wird davon ausgegangen, dass die entsprechenden Iteratorpositionen zulässig sind.

Für Funktionsobjekte gelten folgende Schreibweisen:

- **BinOp**
binärer Funktionsobjekt-Typ (zwei Argumenttypen, ein Ergebnistyp).
- **UnOp**
unärer Funktionsobjekt-Typ (ein Argumenttyp, ein Ergebnistyp).
- **BinPred**
binärer Prädikat-Typ (zwei Argumenttypen, Ergebnistyp `bool`).
- **UnPred**
unärer Prädikat-Typ (ein Argumenttyp, Ergebnistyp `bool`).
- **Comp**
Vergleichsfunktions-Typ (zwei gleiche Argumenttypen, Ergebnistyp `bool`).

Es wird davon ausgegangen, dass die Funktionsobjekte zu den durch die Iteratoren gelieferten Objekten “passen”.

Sollten in einem Algorithmus mehrere verschiedene Typen der gleichen Kategorie als Template-Parameter verwendet werden (etwa zwei unterschiedliche Input-Iterator-Typen), so werden Zahlen (zur Durchnummerierung) angehängt.

Die Namen `T`, `T1` ... stehen für beliebige Typen, die Typen stehen im Allgemeinen aber in Beziehung der Elementtypen entsprechender Iteratoren.

11.7.3 Nichtmodifizierende Algorithmen für Sequenzen

Der Algorithmus `for_each`:

```
template <class InIt, class UnOp>
UnOp for_each(InIt anf, InIt ende, UnOp f);
```

führt die unäre Operation `f` für jedes Element der durch die Input-Iteratoren `anf` und `ende` gegebenen Sequenz von Objekten aus, wobei das Funktionsergebnis der jeweiligen Operation ignoriert wird. (Elementtyp der Iteratoren und Argumenttyp des unären Operators sollten übereinstimmen!) Funktionsergebnis ist der unäre Operator `f` selbst.

Komplexität:

Ist n die Anzahl der Elemente des Sequenz `[anf, ende)`, so wird die Operation `f` genau n mal ausgeführt.

Der unäre Operator `f` sollte sein Argument nicht verändern, so dass die Elemente der durch die Iteratoren gegebenen Sequenz unverändert bleiben.

Beispiel:

Alle Elemente eines Containers auf der Standardausgabe ausgeben (für Containererelemente muss der Ausgabeoperator << definiert sein!):

```
...
// universelle Ausgabefunktion:
template <class T>
void print(T &obj)
{ cout << obj << ' ';}
...
void fkt(list<int> liste, deque<double> deq )
{
    ...
    // Alle Elemente der Liste ausgeben
    for_each(liste.begin(), liste.end(), print<int>);
    ...
    // Alle Elemente der Deque ausgeben
    for_each(deq.begin(), deq.end(), print<double>);
    ...
}
...
```

Soll mit den Elementen der Sequenz nicht nur etwas gemacht, sondern aus ihnen auch etwas berechnet werden, so muss das Resultat der Berechnung im Ergebnis des `for_each`-Algorithmus, also der unären Operation, zurückgegeben werden — eine normale Funktion scheidet somit als unäre Operation aus, es muss schon ein Funktionsobjekt mit Datenkomponenten sein, in dem das Resultat des Algorithmus untergebracht werden kann.

Beispiel:

Es soll der Maximalwert der Elemente einer Sequenz ermittelt werden. Elemente werden mittels < verglichen. Bei der Berechnung des Maximalwertes einer leeren Sequenz wird ein Fehler ausgeworfen.

```
...
class FEHLER {};      // Fehlerklasse:
...
template <class T>
class groesstesOP {
private:
    T maximalwert;
    bool leer;        // bislang noch kein Element gesehen
public:
    // Konstruktor
    groesstesOP() : leer(true) {}

    // Funktionsaufruf: siehe Element wert an:
    void operator()( T& wert)
```

```

    { if ( leer )
      { maximalwert = wert;
        leer = false;
      }
      else if ( maximalwert < wert)
        maximalwert = wert;
    }

    operator T()
    { // maximalwert zurueckgeben
      if ( leer ) throw FEHLER();
      return maximalwert;
    }
};

void f(vector<int> v, deque<double>d )
{ ...
  cout << "Maximalwert des Vektors: "
        << for_each(v.begin(), v.end(), groesstesOP<int>() ) << endl;
  ...
  cout << "Maximalwert der Deque: "
        << for_each(d.begin(), d.end(), groesstesOP<double>() ) << endl;
  ...
}
...

```

Der Algorithmus find:

```

template <class InIt, class T>
InIt find( InIt anf, InIt ende, const T &wert);

```

Sucht in der durch die Input-Iteratoren **anf** und **ende** gegebenen Sequenz nach dem ersten Auftreten eines zu **wert** gleichwertigen Elementes. Ergebnis ist die Iteratorposition auf den Treffer bzw. **ende**, falls kein Treffer vorliegt.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz [**anf**, **ende**), so werden höchstens n Vergleiche mit **wert** ausgeführt.

Nach dem zweiten Treffer in einer Sequenz (etwa in einer Liste) kann man beispielsweise wie folgt suchen:

```

...
void f( liste<double> liste, double wert)
{
  liste<double>::iterator it1, it2;
  ...
  // ersten Treffer suchen:
  it1 = find(liste.begin(), liste.end(), wert);
}

```

```

// falls zumindest ein Treffer vorliegt:
if ( it1 != liste.end() )
{ // zweiten Treffer suchen, Suche hinter dem
  // ersten Treffer beginnen:
  it2 = find ( ++it1, liste.end(), wert);

  // it2 zeigt, falls ungleich liste.end(),
  // auf den zweiten Treffer
  ...
}
...
}
...

```

Der Algorithmus `find_if`:

```

template <class InIt, class UnPred>
InIt find_if( InIt anf, InIt ende, UnPred pred);

```

Sucht in der durch die Input-Iteratoren `anf` und `ende` gegebenen Sequenz nach dem ersten Element, das in das unäre Prädikat `pred` eingesetzt ein Ergebnis “`!=false`“ (i. Allg. also `true`) liefert (für das also das Prädikat zutrifft!).

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so wird das Prädikat `pred` höchstens n mal ausgewertet.

Der Algorithmus `find_first_of`:

```

template <class ForIt, class ForIt2>
ForIt find_first_of( ForIt anf, ForIt ende, ForIt2 anf2, ForIt2 ende2);

```

Sucht in der durch die Vorwärtsiteratoren `anf` und `ende` gegebenen Sequenz `[anf, ende)` nach dem ersten Element, welches mit einem Element in der durch die beiden anderen Vorwärtsiteratoren gegebenen zweiten Sequenz `[anf2, ende2)` übereinstimmt. Ergebnis ist die Iteratorposition des Treffers oder das Ende `ende` der ersten Sequenz. Die Iteratortypen der ersten und der zweiten Sequenz können verschieden sein, die Elementtypen sollten gleich sein.

Beispiel:

Suche in einer Liste nach dem ersten Element, welches mit einem Element eines Vektors übereinstimmt:

```

...
void f(list<int> liste, vector<int> vec)
{
  list<int>::iterator list_it;
  ...
}

```

```

list_it = find_first_of(liste.begin(), liste.end(),
                        vec.begin(), vec.end());

// list_it zeigt, falls ungleich liste.end(),
// auf ersten Treffer!
...
}
...

```

In der zweiten Version

```

template <class ForIt, class ForIt2, class BinPred>
ForIt find_first_of( ForIt anf, ForIt ende, ForIt2 anf2, ForIt2 ende2,
                    BinPred pred);

```

muss das binäre Prädikat für das (zu suchende) Element der ersten Sequenz mit einem Element der zweiten Sequenz den Wahrheitswert *true* ergeben — d.h. das Ergebnis des Algorithmus ist die Iteratorposition des ersten Elementes *elem1* der ersten Sequenz [*anf*, *ende*), zu dem es ein Element *elem2* der zweiten Sequenz [*anf2*, *ende2*) gibt mit *pred(elem1, elem2)* ist *true*. Falls kein Treffer vorliegt, wird wiederum die Iteratorposition *ende* auf das Ende der ersten Sequenz zurückgegeben.

Komplexität:

Ist *n* die Anzahl der Elemente der ersten Sequenz [*anf*, *ende*) und *n2* die Anzahl der Elemente der zweiten Sequenz [*anf2*, *ende2*), so werden höchstens $n \cdot n2$ Vergleiche (mittels *==* bzw. mittels *pred*) durchgeführt.

Der Algorithmus *adjacent_find*:

sucht in einer durch Iteratorpositionen gegebenen Sequenz [*anf*, *ende*) nach dem ersten Paar zweier aufeinanderfolgender Elemente, die (gemeinsam) eine gewisse Bedingung erfüllen. In der ersten Form des Algorithmus (zwei Parameter) müssen die Elemente gleich sein (d.h. die Bedingung ist: der Vergleich mittels *==* liefert *true*) und in der zweiten Form ist als drittes Argument ein binäres Prädikat anzugeben, welches für die benachbarten Elemente aufgerufen den Wahrheitswert *true* liefert:

```

template <class ForIt>
ForIt adjacent_find( ForIt anf, ForIt ende);

template <class ForIt, class BinPred>
ForIt adjacent_find( ForIt anf, ForIt ende, BinPred pred);

```

Ergebnis des Algorithmus ist die Iteratorposition auf das erste Element eines Trefferpaares bzw. die Iteratorposition *ende* auf das Ende der Sequenz.

Komplexität:

Ist *n* die Anzahl der Elemente der Sequenz [*anf*, *ende*), so werden höchstens $n - 1$ Vergleiche (mittels *==* bzw. *pred*) durchgeführt.

Der Algorithmus `count`:

```
template <class InIt, class T>
iterator_traits<InIt>::difference_type count( InIt anf, InIt ende,
                                              const T& wert);
```

zählt, wie oft in der durch die Iteratorpositionen `anf` und `ende` gegebenen Sequenz `[anf, ende)` ein zum angegebenen Wert `wert` gleichwertiges (bzgl. `==`) Element vorkommt.

Ergebnis des Algorithmus ist die entsprechende Anzahl, insbes. der Zahlwert 0, falls keine Übereinstimmung gefunden wurde. Der Typ des Ergebnisses ist der zum Iteratortyp `InIt` passende ganzzahlige Typ `iterator_traits<InIt>::difference_type` (also nicht unbedingt `int`).

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so werden genau n Vergleiche mit dem angegebenen Wert `wert` durchgeführt.

Der Algorithmus `count_if`:

```
template <class InIt, class UnPred>
iterator_traits<InIt>::difference_type count_if( InIt anf, InIt ende,
                                                UnPred pred);
```

zählt, wieviele Elemente in der durch die Iteratorpositionen `anf` und `ende` gegebenen Sequenz `[anf, ende)` sind, für die das unäre Prädikat `pred` der Wahrheitswert `true` liefert.

Das Ergebnis ist die entsprechende Anzahl (ggf. auch 0) und der Typ des Ergebnisses ist wiederum `iterator_traits<InIt>::difference_type`, also der der zum Iteratortyp `InIt` passende ganzzahlige Typ.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so wird das Prädikat `pred` genau n mal ausgewertet.

Der Algorithmus `equal`:

```
template <class InIt, class InIt2>
bool equal( InIt anf, InIt ende, InIt2 anf2);
```

Durch die als Parameter angegebenen Iteratorpositionen sind zwei Sequenzen gegeben, die erste `[anf, ende)` und die zweite beginnt mit der Iteratorposition `anf2`. (Die Iteratortypen `InIt` und `InIt2` können verschieden sein, die Elementtypen der Iteratoren sollten gleich sein. Es wird davon ausgegangen, dass die zweite Sequenz mindestens soviele Elemente hat wie die erste Sequenz — ansonsten ggf. Laufzeitfehler!). Der Algorithmus `equal` prüft auf elementweise Übereinstimmung (bzgl. `==`): (erstes Element der ersten Sequenz wird mit erstem Element der zweiten Sequenz verglichen, zweites Element der ersten Sequenz wird mit zweitem Element der zweiten Sequenz verglichen usw.). Tritt bis zum Ende der ersten Sequenz keine Diskrepanz auf, wird `true` als Ergebnis geliefert, ansonsten `false`.

```
template <class InIt, class InIt2, BinPred>
bool equal( InIt anf, InIt ende, InIt2 anf2, BinPred pred);
```

funktioniert wie die obige Version mit dem Unterschied, dass die korrespondierenden Elemente der beiden Sequenzen anstelle von `==` mit dem binären Prädikat `pred` “verglichen” werden (Ergebnis von `pred(elem1, elem2)` gleich `true` wird als “Gleichheit” der Elemente `elem1` und `elem2` interpretiert!).

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so werden höchstens n Vergleiche (mittels `==` bzw. `pred`) durchgeführt.

Der Algorithmus mismatch:

```
template <class InIt, class InIt2>
pair<InIt, InIt2> mismatch( InIt anf, InIt ende, InIt2 anf2);

template <class InIt, class InIt2, class BinPred>
pair<InIt, InIt2> mismatch( InIt anf, InIt ende, InIt2 anf2,
                           BinPred pred);
```

sind dual zu den `equal`-Algorithmen, gesucht wird das erste Paar korrespondierender Elemente der beiden Sequenzen, die bzgl. `==` (1. Version) bzw. bzgl. des binären Prädikates `pred` “verschieden” sind.

Funktionsergebnis ist ein Paar von Iteratoren auf die beiden unterschiedlichen Elemente der beiden Sequenzen — der erste Iterator des zurückgelieferten Iteratorpaares zeigt auf das Element der ersten Sequenz, der zweite auf das korrespondierende (aber vom Element der ersten Sequenz “verschiedene”) Element der zweiten Sequenz.

Sollten alle korrespondierenden Elemente der beiden Sequenzen “gleich” sein, wird das Iteratorpaar `(ende, it)` zurückgegeben, wobei `ende` das Ende der ersten Sequenz ist und `it` die korrespondierende Iteratorposition der zweiten Sequenz.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so werden höchstens n Vergleiche (mittels `==` bzw. `pred`) durchgeführt.

Der Algorithmus search:

```
template <class ForIt, class ForIt2>
ForIt search( ForIt anf, ForIt ende, ForIt2 anf2, ForIt2 ende2);
```

sucht in der Sequenz `[anf, ende)` nach einer Teilsequenz, welche elementweise mit der zweiten Sequenz `[anf2, ende2)` (bzgl. `==`) übereinstimmt (Iteratortypen können verschieden, Elementtypen sollten gleich sein).

Ergebnis ist die Anfangsposition der ersten passenden Teilsequenz in `[anf, ende)` bzw. `ende`, falls keine passende Teilsequenz in `[anf, ende)` vorhanden ist.

In der Version:

```
template <class ForIt, class ForIt2, BinPred>
ForIt search( ForIt anf, ForIt ende, ForIt2 anf2, ForIt2 ende2,
              BinPred pred);
```


übernimmt das binäre Prädikat `pred` die Rolle des Vergleichs mit `==`.

Komplexität:

Ist n die Anzahl der Elemente der ersten Sequenz `[anf, ende)` und $n2$ die Anzahl der Elemente der zweiten Sequenz `[anf2, ende2)`, so werden höchstens $n \cdot n2$ Vergleiche (mittels `==` bzw. `pred`) durchgeführt.

Der Algorithmus `find_end`:

Die beiden `find_end` Algorithmen entsprechen den obigen `search`-Algorithmen mit dem Unterschied, dass jeweils die Anfangs-Iteratorposition der letzten passenden Teilsequenz zurückgegeben wird:

```
template <class ForIt, class ForIt2>
ForIt find_end( ForIt anf, ForIt ende, ForIt2 anf2, ForIt2 ende2);

template <class ForIt, class ForIt2, BinPred>
ForIt find_end( ForIt anf, ForIt ende, ForIt2 anf2, ForIt2 ende2,
                BinPred pred);
```

Komplexität:

Ist n die Anzahl der Elemente der ersten Sequenz `[anf, ende)` und $n2$ die Anzahl der Elemente der zweiten Sequenz `[anf2, ende2)`, so werden höchstens $n2 \cdot (n - n2 + 1)$ Vergleiche (mittels `==` bzw. `pred`) durchgeführt.

Der Algorithmus `search_n`:

```
template <class ForIt, class Size, class T>
ForIt search_n( ForIt anf, ForIt ende, Size size, Const T& wert);

template <class ForIt, class Size, class T, class BinPred>
ForIt search_n( ForIt anf, ForIt ende, Size size, Const T& wert,
                BinPred pred);
```

sucht in der durch die Iteratorpositionen gegebenen Sequenz `[anf, ende)` nach der ersten Teilsequenz der Länge `size` (`Size` ist hierbei ein ganzzahliger Typ) von Elementen (`elem`), die bezüglich `==` (1. Version) bzw. des binären Prädikates `pred` (2. Version) mit dem angegebenen Wert `wert` übereinstimmen (d.h. es muss `elem == wert` bzw. `pred(elem, wert)` jeweils *true* sein).

Ergebnis ist die Anfangs-Iteratorposition auf die gefundene Teilsequenz bzw. `ende`, falls keine passende Teilsequenz gefunden wurde.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so werden höchstens $n \cdot \text{size}$ Vergleiche durchgeführt.

11.7.4 Modifizierende Algorithmen für Sequenzen

Der Algorithmus `copy`

```
template <class InIt, class OutIt>
OutIt copy( InIt anf, InIt ende, OutIt out);
```

kopiert der Reihe nach (von vornbe nach hinten) alle Elemente der Sequenz `[anf, ende)` in den Ausgabe-Iterator `out`+

Der Ausgabe-Iterator kann etwa auf die Anfangsposition eines Containers zeigen (Container muss groß genug sein), die Elemente des Containers werden dann durch die Elemente der Sequenz überschrieben:

```
...
vector<int> * fkt(list<int> liste)
{
    vector<int> *vp = new vector<int>( liste.size() );
    // vp zeigt jetzt auf einen Vektor, der alle Elemente
    // von liste aufnehmen koennte!
    // Elemente des Vektors sind Standard-int's

    // Elemente der Liste in den Vektor kopieren:
    copy ( liste.begin(), liste.end(), vp->begin() );

    return vp;
}
...
```

Zu beachten in diesem Beispiel ist, dass der Vektor von Anfang an groß genug ist, um alle Elemente der Liste aufzunehmen — ein Vergrößern des Vektors über den Iterator ist nicht möglich (Laufzeitfehler).

Sollen neue Elemente aufgenommen werden, muss man **Insertter** verwenden, welche ja auch Output-Iteratoren sind:

```
...
vector<int> * fkt(list<int> liste)
{
    vector<int> *vp = new vector<int>;
    // vp zeigt zunaechst auf einen leeren Vektor

    // Instertter auf den Vektor erzeugen:
    back_insert_iterator< vector<int> > b_ins(*vp);

    // Elemente der Liste ueber den Insertter in den
    // Vektor kopieren, Vektor wird hierdurch groesser!
    copy ( liste.begin(), liste.end(), b_ins );

    return vp;
}
...
```

Ergebnis des `copy`-Algorithmus ist der Output-Iterator nach dem Kopieren. Dieser Algorithmus `copy` gehört zu den modifizierenden Algorithmen, da sich der Zielbereich (`out` und das, was dahinter ist) und der Quellbereich (`[anf, ende)`) überschneiden dürfen. Die Iteratorposition `out` selbst darf nur nicht innerhalb der Sequenz `[anf, ende)` liegen — kann aber “kleiner“ als `anf` oder größer gleich `ende` sein! Beispiel: Elemente eines Vektors um eine Position nach vorne verschieben, erstes Element geht verloren, letztes Element steht doppelt am Ende:

```
...
vector<int> vec ...;
...
copy ( ++vec.begin(), vec.end(), vec.begin() );
...
```

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so werden genau n Zuweisungen durchgeführt.

Der Algorithmus `copy_backward`

```
template <class BiIt, class BiIt2>
BiIt2 copy_backward( BiIt anf, BiIt ende, BiIt2 out);
```

funktioniert wie `copy`, nur dass die Sequenz `[anf, ende)` rückwärts durchlaufen wird, vor die Iteratorposition `out` kopiert wird und nach jedem Kopieren die Iteratorposition `out` um eins erniedrigt wird (die Reihenfolge der Elemente bleibt also beim Kopieren erhalten und wird nicht umgedreht).

Der Zielbereich muss groß genug sein, um durch die kopierten Elemente überschrieben zu werden.

Quellbereich und Zielbereich des Kopierens dürfen wieder überlappen, der Ziel-Iterator `out` darf nur nicht innerhalb des Quellbereichs `[anf, ende)` liegen.

Beispiel: Elemente eines Vektors um eine Position nach hinten verschieben, letztes Element geht verloren, erstes Element wird verdoppelt:

```
...
vector<int> vec ...;
...
copy_backward ( vec.begin(), --vec.end(), vec.end() );
...
```

Diese Verschiebung um eine Position nach hinten ist mit dem normalen (Vorwärts-) `copy`-Algorithmus

```
copy( vec.begin(), --vec.end(), ++vec.begin() );
```

nicht möglich, da hier der Ziel-Iterator (`++vec.begin()`) innerhalb der zu kopierenden Sequenz (`[vec.begin(), --vec.end()`) wäre.

Zum Umdrehen der Kopierreihenfolge kann man bei `copy` und `copy_backward` Rückwärts-Iteratoren für die zu kopierende Sequenz verwenden:

```

...
vector<int> * fkt(list<int> liste)
{
    vector<int> *vp = new vector<int>( liste.size() );
    // vp zeigt jetzt auf einen Vektor, der alle Elemente
    // von liste aufnehmen koennte!
    // Elemente des Vektors sind Standard-int's

    // Elemente der Liste in den Vektor kopieren:
    copy ( liste.rbegin(), liste.rend(), vp->begin() );

    return vp;
}
...

```

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so werden genau n Zuweisungen durchgeführt.

Der Algorithmus transform

```

template <class InIt, class OutIt, class UnOp>
OutIt transform( InIt anf, InIt ende, OutIt out, UnOp op);

```

wendet auf jedes Element der Sequenz `[anf, ende)` die (unäre) Operation `op` an und schreibt das Ergebnis in den Ausgabe-Iterator `out` (der anschließend um eins erhöht wird).

Der Elementtyp des Input-Iterator-Types `InIt` muss dem Argumenttyp des Operatortypes `UnOp` entsprechen und der Ergebnistyp von `UnOp` muss Elementtyp des Ausgabe-Iterator-Types `OutIt` sein.

Der Ausgabe-Iterator muss wieder entweder auf einen Container zeigen, der groß genug ist, um das Ergebnis der jeweiligen Operation aufnehmen zu können, oder ein `InsertIterator` sein.

Ergebnis des Algorithmus ist der Ausgabe-Iterator nach der Durchführung aller Operationen.

Wenn die Elementtypen der beiden Iteratortypen `InIt` und `OutIt` identisch sind, können sich wiederum Quellbereich (`[anf, ende)`) und Zielbereich (`out` und das, was "dahinter" ist) überlappen (`out` darf nur nicht innerhalb von `[anf, ende)` liegen!).

In der zweiten Version

```

template <class InIt, class InIt2, class OutIt, class BinOp>
OutIt transform( InIt anf, InIt ende, InIt2 anf2, OutIt out, BinOp op);

```

wird auf jedes Element der ersten Sequenz `[anf, ende)` und das korrespondierende (gleiche Position) Element der zweiten Sequenz, deren Anfang die Iteratorposition `anf2` ist, gemeinsam die binäre Operation `op` angewendet und das Ergebnis dieser Operation auf den Ausgabe-Iterator `out` kopiert.

Es wird davon ausgegangen, dass die zweite Sequenz mindestens so viele Elemente wie die erste Sequenz enthält (sonst ggf. Laufzeitfehler, vgl. die Algorithmen `equal` aus Seite 467 und `mismatch` auf Seite 468).

Die Elementtypen der Iteratoren und des Operation müssen zusammenpassen:

Sei die binäre Operation eine Operation mit erstem Argumenttyp `T1`, zweitem Argumenttyp `T2` und Ergebnistyp `T`, so muss der Elementtyp von `InIt` gleich `T1`, er von `InIt2` gleich `T2` und der von `OutIt` gleich `T` sein!

Ist `T1` gleich `T`, so können sich (mit der üblichen Einschränkung: `out` darf nicht innerhalb von `[anf, ende)` liegen) die erste Sequenz `[anf, ende)` und der Zielbereich (`out` und “dahinter“) überlappen.

Ergebnis des Algorithmus ist der Ausgabe-Iterator nach der Durchführung aller Operationen.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so werden genau n (unäre bzw. binäre) Operationen `op` durchgeführt.

Der Algorithmus `swap`

```
template <class T>
void swap ( T& a, T& b);
```

vertauscht die beiden Argumente.

Der Algorithmus `iter_swap`

```
template <class ForIt, class ForIt2>
void iter_swap( ForIt it, ForIt2 it2);
```

vertauscht die beiden durch die Iteratorpositionen `it` und `it2` gegebenen Elemente. Die Iteratortypen können verschieden sein, der Elementtyp der beiden Iteratortypen muss gleich sein.

Der Algorithmus `swap_ranges`

```
template <class ForIt, class ForIt2>
ForIt2 swap_ranges( ForIt anf, ForIt ende, ForIt2 anf2);
```

vertauscht jedes Element der ersten Sequenz `[anf, ende)` mit dem jeweils korrespondierenden (an gleicher Position stehend) der zweiten Sequenz, auf deren Anfang der Iterator `anf2` zeigt. Es wird davon ausgegangen, dass die zweite Sequenz mindestens so viele Elemente enthält, wie die erste (sonst ggf. Laufzeitfehler).

Die Iteratortypen können verschieden, die Elementtypen der Iteratoren sollten gleich sein.

Ergebnis des Algorithmus ist die Iteratorposition der zweiten Sequenz hinter dem letzten vertauschten Element.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so werden genau n Vertauschungen durchgeführt.

Der Algorithmus replace

```
template <class ForIt, class T>
void replace( ForIt anf, ForIt ende, const T& wert, const T& neuerwert);
```

ersetzt in der Sequenz [anf, ende) jedes Element, welches mit dem angegebenen Wert **wert** (bzgl. ==) übereinstimmt, durch den neuen Wert **neuerwert**.

Elementtyp des Iterators und Typ T sollten übereinstimmen.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz [anf, ende), so werden genau n Vergleiche (mittels ==) durchgeführt.

Der Algorithmus replace_if

```
template <class ForIt, class UnPred, class T>
void replace_if( ForIt anf, ForIt ende, UnPred pred, const T& neuerwert);
```

ersetzt in der Sequenz [anf, ende) jedes Element, auf welches das unäre Prädikat **pred** angewendet den Wahrheitswert **true** liefert, durch den neuen Wert **neuerwert**.

Die beteiligten Typen müssen zusammenpassen, d.h. Elementtyp des Iterators, Argumenttyp des Operators und Typ T müssen gleich sein.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz [anf, ende), so werden genau n Vergleiche (mittels **pred**) durchgeführt.

Der Algorithmus replace_copy

```
template <class ForIt, class OutIt, class T>
OutIt replace_copy( ForIt anf, ForIt ende,
                    OutIt out, const T& wert, const T& neuerwert);
```

erstellt im Ausgabe-Iterator **out** eine Kopie der Sequenz [anf, ende), wobei jedes Element der Sequenz [anf, ende), welches mit dem Wert **wert** übereinstimmt, in der Kopie durch den neuen Wert **neuerwert** ersetzt wird.

Ergebnis der Algorithmus ist der Ausgabe-Iterator nach der Erstellung der Kopie.

Die Elementtypen der Iteratoren und Typ T sollten übereinstimmen.

Mit der üblichen Einschränkung: **out** darf nicht innerhalb von [anf, ende) liegen, können sich wiederum Quellbereich und Zielbereich überlappen.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz [anf, ende), so werden genau n Vergleiche (mittels ==) durchgeführt.

Der Algorithmus replace_copy_if

```
template <class ForIt, class OutIt, class UnPred, class T>
OutIt replace_copy_if( ForIt anf, ForIt ende,
                       OutIt out, UnPred pred, const T& neuerwert);
```

erstellt im Ausgabe-Iterator `out` eine Kopie der Sequenz `[anf, ende)`, wobei jedes Element der Sequenz `[anf, ende)`, auf welches das unäre Prädikat `pred` angewendet den Wahrheitswert `true` liefert, in der Kopie durch den neuen Wert `neuerwert` ersetzt wird.

Ergebnis der Algorithmus ist der Ausgabe-Iterator nach der Erstellung der Kopie. Die beteiligten Typen müssen zusammenpassen, d.h. Elementtypen der Iteratoren, Argumenttyp des Operators und Typ `T` müssen gleich sein.

Mit der üblichen Einschränkung: `out` darf nicht innerhalb von `[anf, ende)` liegen, können sich wiederum Quellbereich und Zielbereich überlappen.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so werden genau n Vergleiche (mittels `pred`) durchgeführt.

Der Algorithmus `fill`

```
template <class ForIt, class T>
void fill( ForIt anf, ForIt ende, const T& wert);
```

ersetzt jedes Element der Sequenz `[anf, ende)` durch den angegebenen Wert `wert`. Elementtyp des Iteratortypes und der Typ `T` müssen übereinstimmen.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so werden genau n Zuweisungen durchgeführt.

Der Algorithmus `fill_n`

```
template <class OutIt, class Size, class T>
void fill_n( OutIt out, Size size, const T& wert);
```

erzeugt im Ausgabe-Iterator `out` eine Sequenz von `size` aufeinanderfolgenden Elementen mit Wert `wert`. `Size` ist hierbei ein ganzzahliger Typ.

Der Elementtyp des Iterators `out` und der Typ `T` müssen übereinstimmen.

Als Ausgabe-Iterator `out` man wiederum einen Iterator auf (einen hinreichend großen) Container oder einen Inserter verwenden.

Komplexität:

Es werden genau `size` Zuweisungen durchgeführt.

Der Algorithmus `generate`

```
template <class ForIt, class Generator>
void generate( ForIt anf, ForIt ende, Generator gen);
```

Der Generator `Generator gen` ist hierbei eine parameterlose Funktion, welche einen Wert vom Elementtyp des Iterators als Ergebnis liefert.

Der Algorithmus `generate` ruft für jedes Element der Sequenz `[anf, ende)` einmal diesen Generator `gen` (ohne Argument) auf und ersetzt das Sequenzelement durch den vom Iterator gelieferten Wert.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so erfolgen genau n Aufrufe des Generators `gen` und entsprechende Zueisungen.

Der Algorithmus `generate_n`

```
template <class OutIt, class Size, class Generator>
void generate_n( OutIt out, Size size, Generator gen);
```

Der Generator `Generator gen` ist hierbei eine parameterlose Funktion, welche einen Wert vom Elementtyp des Iterators als Ergebnis `is` liefert.

Der Algorithmus `generate_n` erzeugt im Ausgabe-Iterator `out` eine Sequenz von `size` Elementen mit den durch den (jeweils auf's neue) aufgerufenen Generator `gen` gelieferten Werten.

Der Elementtyp des Iterators `out` und der Ergebnistyp des Generators müssen übereinstimmen.

Als Ausgabe-Iterator `out` man wiederum einen Iterator auf (einen hinreichend großen) Container oder einen Inserter verwenden.

Komplexität:

Es erfolgen genau `size` Aufrufe des Generators `gen` und Zuweisungen.

Der Algorithmus `remove`

```
template <class ForIt, class T>
ForIt remove( ForIt anf, ForIt ende, const T& wert);
```

löscht in der Sequenz `[anf, ende)` jedes Element, welches mit dem angegebenen Wert `wert` (bzgl. `==`) übereinstimmt, alle anderen Elemente bleiben in ihrer ursprünglichen relativen Reihenfolge erhalten.

Ergebnis des Algorithmus ist das (durch das Löschen ggf. neue) Ende der Sequenz. Elementtyp des Iterators und Typ `T` sollten übereinstimmen.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so werden genau n Vergleiche (mittels `==`) durchgeführt.

Der Algorithmus `remove_if`

```
template <class ForIt, class UnPred>
ForIt remove_if( ForIt anf, ForIt ende, UnPred pred);
```

löscht in der Sequenz `[anf, ende)` jedes Element, auf welches das unäre Prädikat `pred` angewendet den Wahrheitswert `true` liefert, alle anderen Elemente bleiben in ihrer ursprünglichen relativen Reihenfolge erhalten.

Ergebnis des Algorithmus ist das (durch das Löschen ggf. neue) Ende der Sequenz. Die beteiligten Typen müssen zusammenpassen, d.h. Elementtyp des Iterators, Argumenttyp des Operators müssen gleich sein.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so wird das Prädikat `pred` genau n ausgerufen.

Der Algorithmus `remove_copy`

```
template <class ForIt, class OutIt, class T>
OutIt remove_copy( ForIt anf, ForIt ende, OutIt out, const T& wert);
```

erstellt im Ausgabe-Iterator `out` eine Kopie der Sequenz `[anf, ende)`, wobei jedes Element der Sequenz `[anf, ende)`, welches mit dem Wert `wert` übereinstimmt, in der Kopie fortgelassen wird, alle anderen Elemente werden in ihrer ursprünglichen relativen Reihenfolge kopiert.

Ergebnis der Algorithmus ist der Ausgabe-Iterator nach der Erstellung der Kopie.

Die Elementtypen der Iteratoren und Typ `T` sollten übereinstimmen.

Mit der üblichen Einschränkung: `out` darf nicht innerhalb von `[anf, ende)` liegen, können sich wiederum Quellbereich und Zielbereich überlappen.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so werden genau n Vergleiche (mittels `==`) durchgeführt.

Der Algorithmus `remove_copy_if`

```
template <class ForIt, class OutIt, class UnPred>
OutIt remove_copy_if( ForIt anf, ForIt ende, OutIt out, UnPred pred);
```

erstellt im Ausgabe-Iterator `out` eine Kopie der Sequenz `[anf, ende)`, wobei jedes Element der Sequenz `[anf, ende)`, auf welches das unäre Prädikat `pred` angewendet den Wahrheitswert `true` liefert, in der Kopie fortgelassen wird, alle anderen Elemente werden in ihrer ursprünglichen relativen Reihenfolge kopiert.

Ergebnis der Algorithmus ist der Ausgabe-Iterator nach der Erstellung der Kopie.

Die beteiligten Typen müssen zusammenpassen, d.h. Elementtypen der Iteratoren und der Argumenttyp des Opertors müssen gleich sein.

Mit der üblichen Einschränkung: `out` darf nicht innerhalb von `[anf, ende)` liegen, können sich wiederum Quellbereich und Zielbereich überlappen.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so wird das Prädikat `pred` genau n ausgerufen.

Der Algorithmus `unique`

```
template <class ForIt>
ForIt unique( ForIt anf, ForIt ende);
```

entfernt aus der Sequenz `[anf, ende)` jedes Element, welches (bzgl. `==`) zu seinem Vorgänger gleich ist.

Ergebnis des Algorithmus ist das (nach dem Löschen ggf. neue) Ende der Sequenz.

Die zweite Version

```
template <class ForIt, class UnPred>
ForIt unique( ForIt anf, ForIt ende, BinPred pred);
```

entfernt aus der Sequenz `[anf, ende)` jedes Element, welches mit seinem Vorgänger in das binäre Prädikat `pred` eingesetzt (Vorgänger als erstes Argument) den Wahrheitswert `true` liefert.

Der Elementtyp des Iterortypes und die Argumenttypen des binären Prädikates müssen gleich sein.

Ergebnis des Algorithmus ist auch hier das (nach dem Löschen ggf. neue) Ende der Sequenz.

Komplexität:

Ist die Sequenz `[anf, ende)` nicht leer und n die Anzahl ihrer Elemente, so wird genau $n - 1$ mal ein Vergleich (mittels `==` bzw. `pred`) durchgeführt,

Der Algorithmus `unique_copy`

```
template <class ForIt, class OutIt>
OutIt unique_copy( ForIt anf, ForIt ende, OutIt out);
```

erstellt (im Ausgabe-Iterator `out`) eine Kopie der Sequenz `[anf, ende)`, wobei in der Kopie jedes Element der Originalsequenz, welches (bzgl. `==`) zu seinem Vorgänger der Originalsequenz gleich ist, fortgelassen ist.

Die zweite Version

```
template <class ForIt, class UnPred>
ForIt unique_copy( ForIt anf, ForIt ende, BinPred pred);
```

erstellt (im Ausgabe-Iterator `out`) eine Kopie der Sequenz `[anf, ende)`, wobei in der Kopie jedes Element der Originalsequenz, welches mit seinem Vorgänger in der Originalsequenz in das binäre Prädikat `pred` eingesetzt (Vorgänger als erstes Argument) den Wahrheitswert `true` liefert, fortgelassen ist.

Ergebnis der `unique_copy`-Algorithmen ist der Ausgabe-Iterator nach dem Kopieren. Die Elementtypen der Iterortypen müssen gleich sein und — in der zweiten Version — mit dem Typ beider Argumente des binären Prädikates übereinstimmen.

Der Ausgabe-Iterator `out` muss entweder auf einen hinreichend großen Container zeigen oder ein Inserter sein.

Quellbereich und Zielbereich dürfen hierbei nicht überlappen.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so werden genau n Vergleiche (mittels `==` bzw. `pred`) durchgeführt.

Der Algorithmus `reverse`

```
template <class BiIt>
void reverse( BiIt anf, BiIt ende);
```

dreht die Reihenfolge der durch die bidirektionalen Iteratoren gegebenen Sequenz `[anf, ende)` um (erstes Element wird zum letzten, zweites Element zum vorletzten usw.).

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so werden genau $\lfloor n/2 \rfloor$ (größte Ganzzahl kleiner gleich $n/2$) Vertauschungen durchgeführt.

Der Algorithmus `reverse_copy`

```
template <class BiIt, class OutIt>
OutIt reverse_copy( BiIt anf, BiIt ende, OutIt out);
```

erstellt (im Ausgabe-Iterator `out`) eine Kopie der durch die bidirektionalen Iteratoren gegebenen Sequenz `[anf, ende)`, wobei die Kopie die umgekehrte Elementfolge des Originals aufweist (originales erstes Element wird zum letzten der Kopie, originales zweites Element zum vorletzten der Kopie usw.).

Der Ausgabe-Iterator `out` muss wiederum entweder auf einen hinreichend großen Container zeigen oder ein Inserter sein.

Ergebnis ist die Ende-Iteratorposition der Kopie.

Quellbereich und Zielbereich dürfen hier nicht überlappen.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so werden genau n Zuweisungen durchgeführt.

Der Algorithmus `rotate`

```
template <class ForIt>
void rotate( ForIt anf, ForIt mitte, ForIt ende);
```

rotiert die Elemente der durch die Vorwärts-Iteratoren gegebenen Sequenz `[anf, ende)` um so viele Positionen nach links, dass das Element mit der vormaligen Position `mitte` das neue erste Element wird. Jedes vorne “herausfallende“ Element wird hinten wieder “eingefügt“.

Notwendig ist, dass die beiden Teilsequenzen `[anf, mitte)` und `[mitte, ende)` vernünftige Sequenzen sind (dass also `mitte` zwischen einschließlich `anf` und ausschließlich `ende` liegt).

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so werden höchstens n Vertauschungen durchgeführt.

Der Algorithmus `rotate_copy`

```
template <class ForIt, class OutIt>
OutIt rotate_copy( ForIt anf, ForIt mitte, ForIt ende, OutIt out);
```

erstellt (im Ausgabe-Iterator `out`) eine so nach links rotierte Kopie der durch die Vorwärts-Iteratoren gegebenen Sequenz `[anf, ende)`, dass das Element mit der Position `mitte` der Originalsequenz in der Kopie das erste Element ist (siehe Algorithmus `rotate`, Seite 479).

Der Ausgabe-Iterator `out` muss wiederum entweder auf einen hinreichend großen Container zeigen oder ein Inserter sein.

Ergebnis ist die Ende-Iteratorposition der Kopie.

Quellbereich und Zielbereich dürfen hier nicht überlappen.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so werden höchstens n Zuweisungen durchgeführt.

Der Algorithmus `random_shuffle`

```
template <class RanIt>
void random_shuffle( RanIt anf, RanIt ende);
```

mischt die Elemente der durch die Random-Access-Iteratoren gegebene Sequenz `[anf, ende)` “zufällig“, so dass jede Reihenfolge gleich wahrscheinlich ist (oder sein sollte!).

Aufgrund der Tatsache, dass Random-Access-Iteratoren notwendig sind, kann dieser Algorithmus nicht nur für alle Standardcontainer angewendet werden, sondern nur für `vectoren` und `deque`'s.

Die Art der Generierung der Zufallszahlen, welche der Bestimmung der Zufallsreihenfolge zugrundeliegt, kann in der zweiten Form des Algorithmus angegeben werden:

```
template <class RanIt, class RandomNumberGen>
void random_shuffle( RanIt anf, RanIt ende, RandomNumberGen gen);
```

Hierbei muss der Typ `RandomNumberGen` ein unärer Funktionsobjekttyp sein, der ein Argument und Ergebnis vom zum Iteratortyp passenden ganzzahligen Differenztyp

`iterator_traits<RanIt>::difference_type`

hat. Das entsprechende Funktionsobjekt `gen` muss bei einem Aufruf der Form `gen(n)` ein (zufälliges) Ergebnis aus dem Bereich von 0 bis $n - 1$ zurückgeben.

Diese “Funktion“ `gen` wird im Laufe des `random_shuffle`-Algorithmus mehrfach (Elementzahl der Sequenz minus 1) aufgerufen und anhand der Ergebnisse wird die neue Reihenfolge der Elemente der Sequenz konstruiert.

Beispiel: für einen selbst definierten (ganz schlechten) “Zufallsgenerator“ und dessen Verwendung mittels `random_shuffle`:

```
...
// unsigned long als Differenz-Typ:
unsigned long random(unsigned long n)
{ return 0; } // gibt immer 0 zurueck!
...
vector<double> vec(100);
...
random_shuffle(vec.begin(), vec.end(), ptr_fun(random) );
// drittes Argument: aus gewoehnlicher Funktion
// ein Funktionsobjekt machen!
...
```

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)` und die Sequenz nicht leer, so werden genau $n - 1$ Vertauschungen durchgeführt.

Der Algorithmus `partition`

```
template <class BiIt, class UnPred>  
BiIt partition( BiIt anf, BiIt ende, UnPred pred);
```

vertauscht die Elemente in der Sequenz `[anf, ende)` so, dass im Ergebnis alle Elemente, welche das unäre Prädikat `pred` erfüllen, vor allen anderen Elementen stehen. Rückgabe des Algorithmus ist die Iteratorposition auf das erste Element (der Sequenz nach der Vertauschung), welches das unäre Prädikat `pred` nicht erfüllt (alle vorher erfüllen es).

Argumenttyp des Prädikates und Elementtyp des Iterators müssen gleich sein.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)` so wird das Prädikat `pred` genau n mal ausgewertet und es werden maximal $\lfloor n/2 \rfloor$ (größte Ganzzahl kleiner gleich $n/2$) Vertauschungen von Elementen durchgeführt.

Der Algorithmus `stable_partition`

```
template <class BiIt, class UnPred>  
BiIt stable_partition( BiIt anf, BiIt ende, UnPred pred);
```

vertauscht, wie der Algorithmus `partition`, die Elemente in der Sequenz `[anf, ende)` so, dass im Ergebnis alle Elemente, welche das unäre Prädikat `pred` erfüllen, vor allen anderen Elementen stehen.

Rückgabe des Algorithmus ist die Iteratorposition auf das erste Element (der Sequenz nach der Vertauschung), welches das unäre Prädikat `pred` nicht erfüllt (alle vorher erfüllen es).

Argumenttyp des Prädikates und Elementtyp des Iterators müssen gleich sein.

Der Unterschied zum Algorithmus `partition` ist der, dass bei `stable_partition` garantiert ist, dass die relative Reihenfolge aller Elemente, welche das Prädikat erfüllen, untereinander gleich bleibt, und die relative Reihenfolge der Elemente, welche das Prädikat nicht erfüllen, ebenfalls untereinander gleich bleibt.

Komplexität:

Ist n die Anzahl der Elemente der Sequenz `[anf, ende)`, so wird das Prädikat `pred` genau n mal ausgewertet. Falls genug Speicher vorhanden ist, werden $\mathcal{O}(n)$ Vertauschungen durchgeführt, ansonsten (nicht genug Speicher) höchstens $n \cdot \ln(n)$ Vertauschungen.

11.7.5 Algorithmen und Sortierung

Bei folgenden Algorithmen liegt den Elementen der Sequenzen ein Vergleich (*kleiner*) zugrunde.

Standardmäßig wird hierbei mittels des Vergleichsoperators `<` verglichen, zu jedem Algorithmus gibt es aber zusätzlich auch eine Version, wo die Vergleichsoperation mittels eines binären Prädikates `BinPred comp` angegeben werden kann.

Sowohl der (möglicherweise selbstdefinierte) Vergleich mit `<` als auch der mittels des binären Prädikates `comp` muss zumindest eine *strikte, schwache Ordnung* auf der Menge der Elemente der Sequenz definieren.

Sei T der Elementtyp der Sequenz und (doppelter) Argumenttyp der Vergleichsoperation ($<$ oder `comp`) und a , b und c seien beliebige Werte vom Typ T , so muss also gelten:

- Für jeden Wert a (vom Typ T) muss $a < a$ bzw. `comp(a, a)` das Ergebnis *false* liefern (diese Eigenschaft heißt: *strikt*),
- Der Vergleich mit $<$ bzw. `comp` muss *transitiv* sein, d.h.
 - aus “ $a < b$ *wahr*“ und “ $b < c$ *wahr*“ muss “ $a < c$ *wahr*“ bzw.
 - aus “`comp(a, b)` *wahr*“ und “`comp(b, c)` *wahr*“ muss “`comp(a, c)` *wahr*“

folgen.

- Aus
 - “ $a < b$ *falsch*“ und “ $b < a$ *falsch*“ bzw.
 - “`comp(a, b)` *falsch*“ und “`comp(b, a)` *falsch*“

wird geschlossen, dass a und b bezüglich des Vergleichs (mit $<$ bzw. `comp`) als *gleich* angesehen werden (sie müssen nicht wirklich, etwa bzgl. `==`, gleich sein!). Diese *Gleichheit* muss wiederum transitiv sein, d.h. aus “ a *gleich* b “ und “ b *gleich* c “ muss “ a *gleich* c “ folgen.

Der Algorithmus `sort`

```
template <class RanIt>
void sort( RanIt anf, RanIt ende);

template <class RanIt, class BinPred>
void sort( RanIt anf, RanIt ende, BinPred comp);
```

sortiert die Sequenz `[anf, ende)` (aufsteigend) bzgl. $<$ bzw. `comp`. Anschließend ist für jedes Element `elem2`, welches in der Sequenz “weiter hinten“ steht, als ein Element `elem1` der Vergleich `elem2 < elem1` bzw. `comp(elem2, elem1)` *falsch*. (Das heißt nicht, das der umgekehrte Vergleich `elem1 < elem2` bzw. `comp(elem1, elem2)` *wahr* sein muss!)

Komplexität:

Sei n die Anzahl der Elemente der Sequenz, so werden im Mittel $n \cdot \ln(n)$ Vergleiche durchgeführt (d.h. es liegt besserer Sortieralgorithmus, vielleicht *Quicksort*, zugrunde).

Der Algorithmus `stable_sort`

```
template <class RanIt>
void stable_sort( RanIt anf, RanIt ende);

template <class RanIt, class BinPred>
void stable_sort( RanIt anf, RanIt ende, BinPred comp);
```

sortiert (wie `sort`) die Sequenz `[anf, ende)` (aufsteigend) bzgl. `<` bzw. `comp`. Anschließend ist für jedes Element `elem2`, welches in der Sequenz “weiter hinten” steht, als ein Element `elem1` der Vergleich `elem2 < elem1` bzw. `comp(elem2, elem1)` falsch. Elemente, die bezüglich des Vergleichs *gleich* sind, behalten ihre relative Reihenfolge untereinander.

Komplexität:

Sei n die Anzahl der Elemente der Sequenz, so werden höchstens $n \cdot (\ln(n))^2$ Vergleiche durchgeführt, falls genügend Speicher zur Verfügung steht, sind es (im Mittel) $n \cdot \ln(n)$.

Der Algorithmus `partial_sort`

```
template <class RanIt>
void partial_sort( RanIt anf, RanIt mitte, RanIt ende);

template <class RanIt, class BinPred>
void partial_sort( RanIt anf, RanIt mitte, RanIt ende, BinPred comp);
```

(der Iterator `mitte` muss dabei auf ein Element der Sequenz `[anf, ende)` zeigen.)
Sortiert teilweise die Sequenz `[anf, ende)` so, dass anschließend die Anfangssequenz `[anf, mitte)` (aufsteigend) bzgl. `<` bzw. `comp` sortiert ist und jedes Element der zweiten Teilsequenz `[mitte, ende)` nicht kleiner als jedes Element der ersten Teilsequenz `[anf, mitte)`. (Die zweite Teilsequenz selber ist nicht sortiert!)

Komplexität:

Sei n die Anzahl der Elemente der ganzen Sequenz `[anf, ende)` und m die Anzahl der Elemente der ersten Teilsequenz `[anf, mitte)`, so werden $\mathcal{O}(n \cdot \ln(m))$ Vergleiche durchgeführt.

Der Algorithmus `partial_sort_copy`

```
template <class InIt, class RanIt>
void partial_sort_copy( InIt anf, InIt ende, RanIt anf2, RanIt ende2);

template <class InIt, class RanIt, class BinPred>
void partial_sort_copy( InIt anf, InIt ende, RanIt anf2, RanIt ende2,
                       BinPred comp);
```

Sei n die Anzahl der Elemente der ersten Sequenz `[anf, ende)` (Input-Iteratoren), n_2 die Anzahl der Elemente der zweiten Sequenz `[anf2, ende2)` (Bidirektionale-Iteratoren) und m das Minimum von n und n_2 .

Kopiert die “kleinsten” m Elemente der ersten Sequenz sortiert in die ersten m Positionen der zweiten Sequenz.

Komplexität:

Sei $n = n$ die Anzahl der Elemente der ganzen Sequenz `[anf, ende)` und $m = m$ die Anzahl der kopierten Elemente, so werden $\mathcal{O}(n \cdot \ln(m))$ Vergleiche durchgeführt.

Der Algorithmus nth_element

```
template <class RanIt>
void nth_element( RanIt anf, RanIt nth, RanIt ende);

template <class RanIt, class BinPred>
void nth_element( RanIt anf, RanIt nth, RanIt ende, BinPred comp);
```

sortiert die Sequenz `[anf, ende)` so, dass jedes Element in der zweiten Teilsequenz `[nth, ende)` nicht kleiner ist als jedes Element der ersten Teilsequenz `[anf, nth)`.

Komplexität:

Sei n die Anzahl der Elemente der ganzen Sequenz `[anf, ende)`, so werden im Mittel $\mathcal{O}(n)$ Vergleiche durchgeführt.

Der Algorithmus lower_bound

```
template <class ForIt, class T>
ForIt lower_bound( ForIt anf, ForIt ende, const T& wert);

template <class ForIt, class T, class BinPred>
ForIt lower_bound( ForIt anf, ForIt ende, const T& wert, BinPred comp);
```

Die Sequenz `[anf, ende)` muss (bzgl. `<` bzw. `comp`) sortiert sein.

Gibt die erste Iteratorposition der Sequenz zurück, vor der ein Element mit dem Wert `wert` eingefügt werden könnte, ohne die Sortierung zu zerstören.

Komplexität:

Sei n die Anzahl der Elemente der ganzen Sequenz `[anf, ende)`, so werden höchstens $\ln(n) + 1$ Vergleiche durchgeführt.

Der Algorithmus upper_bound

```
template <class ForIt, class T>
ForIt upper_bound( ForIt anf, ForIt ende, const T& wert);

template <class ForIt, class T, class BinPred>
ForIt upper_bound( ForIt anf, ForIt ende, const T& wert, BinPred comp);
```

Die Sequenz `[anf, ende)` muss (bzgl. `<` bzw. `comp`) sortiert sein.

Gibt die letzte Iteratorposition der Sequenz zurück, vor der ein Element mit dem Wert `wert` eingefügt werden könnte, ohne die Sortierung zu zerstören.

Komplexität:

Sei n die Anzahl der Elemente der ganzen Sequenz `[anf, ende)`, so werden höchstens $\ln(n) + 1$ Vergleiche durchgeführt.

Der Algorithmus `equal_range`

```
template <class ForIt, class T>
pair<ForIt,ForIt> equal_range( ForIt anf, ForIt ende, const T& wert);

template <class ForIt, class T, class BinPred>
pair<ForIt,ForIt> equal_range( ForIt anf, ForIt ende, const T& wert,
                             BinPred comp);
```

Die Sequenz `[anf, ende)` muss (bzgl. `<` bzw. `comp` sortiert sein.

Gibt ein Paar `(erg_anf, erg_ende)` von Vorwärts-Iteratorposition auf Sequenz zurück, so dass ein Element mit dem angegebenen Wert `wert` vor jede Position der durch das Ergebnis gegebenen Teilsequenz `[erg_anf, erg_ende)` eingefügt werden könnte, ohne die Sortierung zu zerstören.

Komplexität:

Sei n die Anzahl der Elemente der ganzen Sequenz `[anf, ende)`, so werden höchstens $2 \cdot \ln(n) + 1$ Vergleiche durchgeführt.

Der Algorithmus `binary_search`

```
template <class ForIt, class T>
bool binary_search( ForIt anf, ForIt ende, const T& wert);

template <class ForIt, class T, class BinPred>
bool binary_search( ForIt anf, ForIt ende, const T& wert, BinPred comp);
```

Die Sequenz `[anf, ende)` muss (bzgl. `<` bzw. `comp` sortiert sein.

Gibt im Ergebnis zurück, ob ein zum Wert `wert` bzgl. des Vergleichs (`<` bzw. `comp`) “*gleiches*“ Element in der Sequenz vorhanden ist oder nicht.

Komplexität:

Sei n die Anzahl der Elemente der ganzen Sequenz `[anf, ende)`, so werden höchstens $\ln(n) + 2$ Vergleiche durchgeführt.

Der Algorithmus `merge`

```
template <class InIt, class InIt2, class OutIt>
OutIt merge( InIt anf, InIt ende, InIt2 anf2, InIt2 ende2, OutIt out);

template <class InIt, class InIt2, class OutIt, class BinPred>
OutIt merge( InIt anf, InIt ende, InIt2 anf2, InIt2 ende2, OutIt out,
             BinPred comp);
```

Die Sequenzen `[anf, ende)` und `[anf2, ende2)` müssen (bzgl. `<` bzw. `comp` sortiert sein und den gleichen Elementtyp haben.

Mischt die beiden sortierten “Eingabesequenzen“ `[anf, ende)` und `[anf2, ende2)` im Ausgabe-Iterator `out` zu einer sortierten Sequenz zusammen.

Gibt als Ergebnis die Ende-Iteratorposition der erstellten Sequenz zurück.

Die Elemente einer jeden “Eingabesequenz” behalten in der “Ausgabesequenz” ihre vormalige relative Reihenfolge.

Komplexität:

Sei n die Anzahl der Elemente der ersten Eingabesequenz $[\text{anf}, \text{ende})$ und n_2 die Anzahl der Elemente der zweiten Eingabesequenz $[\text{anf2}, \text{ende2})$, so werden höchstens $n + n_2 - 1$ Vergleiche durchgeführt.

Der Algorithmus `inplace_merge`

```
template <class BiIt>
void inplace_merge( BiIt anf, BiIt mitte, BiIt ende);

template <class BiIt, class BinPred>
void inplace_merge( BiIt anf, BiIt mitte, BiIt ende, BinPred comp);
```

Die beiden Teilsequenzen $[\text{anf}, \text{mitte})$ und $[\text{mitte}, \text{ende})$ der ganzen Sequenz $[\text{anf}, \text{ende})$ müssen (bzgl. $<$ bzw. `comp` sortiert sein.

Mischt aus den beiden sortierten “Teilsequenzen” $[\text{anf}, \text{mitte})$ und $[\text{mitte}, \text{ende})$ die ganze Sequenz (bzgl. $<$ bzw. `comp`) sortiert zusammen. Elemente der einzelnen Teilsequenzen behalten ihre ursprüngliche relative Reihenfolge.

Komplexität: Sei n die Anzahl der Elemente der ganzen Sequenz $[\text{anf}, \text{ende})$, so werden höchstens $n - 1$ Vergleiche durchgeführt.

Die Algorithmen `next_permutation` und `prev_permutation`

Ein durch $<$ bzw. `comp` auf dem Elementtyp definierte (strikte, schwache) Ordnung definiert auf Sequenzen dieses Elementtypes (Sequenzen der gleichen Länge n und mit den gleichen Elementen) eine *lexikographische Ordnung*:

Die Sequenz $[a_1, a_2, \dots, a_n]$ ist genau dann “*kleiner*“ als eine Sequenz $[b_1, b_2, \dots, b_n]$, wenn es ein k , $1 \leq k \leq n$ gibt, so dass

- für alle i , $1 \leq i < k$ gilt: a_i ist nicht *kleiner* als b_i und b_i ist nicht *kleiner* als a_i (d.h. a_i und b_i sind bezüglich des Vergleichs *gleich*).
- und a_k ist *kleiner* als b_k .

Die “*kleinste*“ Sequenz ist die aufsteigend sortierte und die “*größte*“ ist die absteigend sortierte.

Der Algorithmus

```
template <class BiIt>
bool next_permutation( BiIt anf, BiIt ende);

template <class BiIt, class BinPred>
bool next_permutation( BiIt anf, BiIt ende, BinPred comp);
```

vertauscht, falls möglich, die Elemente der Sequenz $[\text{anf}, \text{ende})$ so, dass die (bzgl. des Vergleichs mit $<$ bzw. `comp`) lexikographisch nächst größere Sequenz entsteht — in diesem Fall wird `true` als Funktionsergebnis geliefert.

Ist dies nicht möglich (d.h. die Sequenz war absteigend sortiert), so wird die Sequenz aufsteigend sortiert und das Ergebnis `false` geliefert.

```
template <class BiIt>
bool prev_permutation( BiIt anf, BiIt ende);

template <class BiIt, class BinPred>
bool prev_permutation( BiIt anf, BiIt ende, BinPred comp);
```

vertauscht, falls möglich, die Elemente der Sequenz `[anf, ende)` so, dass die (bzgl. des Vergleichs mit `<` bzw. `comp`) lexikographisch nächst kleinere Sequenz entsteht — in diesem Fall wird `true` als Funktionsergebnis geliefert.

Ist dies nicht möglich (d.h. die Sequenz war aufsteigend sortiert), so wird die Sequenz absteigend sortiert und das Ergebnis `false` geliefert.

Erläuterung zur Anzahl der Permutationen und Ermittlung der nächst größeren:

Sollte die Sequenz aus n Elementen n_1 *kleinste* (untereinander *gleiche*) Elemente, n_2 nächst *kleinere* (untereinander *gleiche* usw. und n_r größte (untereinander *gleiche*) Elemente haben ($n_1 + n_2 + \dots + n_r = n$), so gibt es genau

$$\frac{n!}{n_1! \cdot n_2! \cdot \dots \cdot n_r!}$$

verschiedene Permutationen, die von diesen Algorithmen der Reihe nach ermittelt werden.

Zur Berechnung der nächst größeren Permutation der Sequenz $[a_1, a_2, \dots, a_n]$ wird in der Sequenz ein möglichst “weit rechts“ stehendes Element a_i ermittelt, so dass irgendwo rechts von diesem Element ein noch größeres steht. Von diesen rechts von a_i stehenden Elementen, welche größer als a_i sind, wird das (oder ein) kleinste(s) Element a_j ausgewählt. Diese beiden Elemente a_i und a_j werden vertauscht und der jetzt rechts von a_j stehende Teil der Sequenz aufsteigend sortiert.

Zur Berechnung der nächst kleineren Permutation wird analog verfahren.

Trotz des scheinbar großen Aufwandes gilt für jeden Aufruf einer der Algorithmen:

Komplexität:

Sei n die Anzahl der Elemente der ganzen Sequenz `[anf, ende)`, so werden höchstens $\lfloor n/2 \rfloor$ (größte Ganzzahl kleiner gleich $n/2$) Vertauschungen durchgeführt.

11.7.6 Algorithmen für Mengen

Will man für Sequenzen Mengen-Algorithmen durchführen, so müssen alle beteiligten Sequenzen (bzgl. eines durch `<` bzw. oder durch ein binäres Prädikat `BinPred comp` im Sinne von Abschnitt 11.7.5 definierten Vergleichs) aufsteigend sortiert sein.

Da dies bei den Standardcontainern `set<T>`, `multiset<T>`, `map<Key, T>` und `multimap<Key, T>` (bei den `map`'s ist der Vergleich bezüglich des Schlüssels vom Typ `Key`) automatisch gegeben ist, sind die folgenden Algorithmen insbesondere für diese Standardcontainer geeignet.

Treten in den Sequenzen keine (bezüglich des Vergleiches) *doppelten* Elemente auf (zumindest bei `set<T>` und `map<Key, T>` der Fall), so kann der Mengenbegriff im

mathematischen Sinn verstanden werden und die Mengen-Operationen entsprechen den mathematischen.

Treten in (mindestens) einer Sequenz *gleiche* Elemente mehrfach auf, so spielen diese *Vielfachheiten* eines Elementes (Anzahl der zum Element im Sinne des Vergleiches *gleichen* Elemente in der Sequenz, das Element selbst wird mitgezählt) eine Rolle und die Sache wird etwas komplexer.

Der Algorithmus includes

```
template <class InIt, class InIt2>
bool includes( InIt anf, InIt ende, InIt2 anf2, InIt2 ende2);

template <class InIt, class InIt2, class BinPred>
bool includes( InIt anf, InIt ende, InIt2 anf2, InIt2 ende2,
              BinPred comp);
```

überprüft, ob die zweite (sortierte) Sequenz [anf2, ende2) in der ersten (sortierten) Sequenz [anf, ende) “enthalten“ ist.

Genauer, für (der Sortierreihenfolge nach) jedes Element `elem2` der zweiten Sequenz, die Vielfachheit des Elementes `elem2` in der zweiten Sequenz sei v_2 , wird folgendes überprüft:

Gibt es in der ersten Sequenz kein (bzgl. des Vergleichs) zu `elem2` *gleiches* Element, so gibt der Algorithmus `false` zurück.

Gibt es in der ersten Sequenz ein (bzgl. des Vergleichs) zu `elem2` *gleiches* Element `elem1` und ist dieses Element `elem1` habe in der ersten Sequenz die Vielfachheit v_1 und ist dann v_1 kleiner als v_2 , so gibt der Algorithmus ebenfalls `false` zurück.

Ist für alle Elemente `elem2` der zweiten Sequenz die Vielfachheit v_2 kleiner oder gleich der Vielfachheit v_1 eines *gleichen* Elementes `elem1` aus der ersten Sequenz, so gibt der Algorithmus `true` zurück.

Die Iteratortypen (und dahinter stehende Containertypen) können verschieden sein, die Elementtypen der Iteratoren (und Container) müssen gleich sein und zum Argumenttyp des Vergleichs passen.

Komplexität:

Sei n die Anzahl der Elemente der ersten Sequenz [anf, ende) und n_2 die Anzahl der Elemente der zweiten Sequenz [anf2, ende2), so werden höchstens $2 \cdot (n + n_2) - 1$ Vergleiche durchgeführt.

Der Algorithmus set_union

```
template <class InIt, class InIt2, class OutIt>
OutIt set_union( InIt anf, InIt ende, InIt2 anf2, InIt2 ende2,
                OutIt out);

template <class InIt, class InIt2, class OutIt, class BinPred>
OutIt set_union( InIt anf, InIt ende, InIt2 anf2, InIt2 ende2,
                OutIt out, BinPred comp);
```

Es wird die (sortierte) Vereinigungsmenge der beiden (sortierten), durch die Iteratoren gegebenen Sequenzen `[anf, ende)` und `[anf2, ende2)` gebildet und diese Vereinigungsmenge in den Ausgabe-Iterator `out` geschrieben (der Ausgabe-Iterator `out` muss entweder auf einen hinreichend großen Container zeigen, dessen ersten Elemente durch die Vereinigungsmenge überschrieben werden, oder ein Inserter sein).

Genauer: Es werden der Sortierreihenfolge nach alle bezüglich des Vergleichs *verschiedenen* Elemente `elem1` der ersten Sequenz betrachtet (d.h. von *gleichen* Elementen der ersten Sequenz wird nur jeweils eins in Betracht gezogen) — die Vielfachheit des Elementes `elem1` in der ersten Sequenz sei v_1 .

Gibt es in der zweiten Sequenz kein zu `elem1` *gleiches* Element oder gibt es in der zweiten Sequenz ein zu `elem1` *gleiches* Element `elem2` und die Vielfachheit von `elem2` in der zweiten Sequenz sei v_2 und ist v_2 nicht größer als die Vielfachheit v_1 , so werden nur alle zu `elem1` gleichen Elemente aus der ersten Sequenz in ihrer ursprünglichen Reihenfolge in die Vereinigungsmenge geschrieben.

Ist jedoch die Vielfachheit v_2 des Elementes `elem2` in der zweiten Sequenz größer als die Vielfachheit v_1 , so werden in die Vereinigung (Ausgabe-Iterator `out`) alle v_1 zu `elem1` *gleichen* Elemente der ersten Sequenz (in ihrer ursprünglichen Reihenfolge) geschrieben und zusätzlich (in ihrer ursprünglichen Reihenfolge) die letzten $v_2 - v_1$ zu `elem2` *gleichen* Elemente der zweiten Sequenz.

Ergebnis des Algorithmus ist die Ende-Iteratorposition des Ausgabe-Iterators `out`.

Die Iteratortypen (und dahinter stehende Containertypen) können verschieden sein, die Elementtypen der Iteratoren (und Container) müssen gleich sein und zum Argumenttyp des Vergleichs passen.

Der Zielbereich (`out` und das, was dahinter ist) darf mit keinem der Quellbereiche (erste oder zweite Sequenz) überlappen.

Komplexität:

Sei n die Anzahl der Elemente der ersten Sequenz `[anf, ende)` und n_2 die Anzahl der Elemente der zweiten Sequenz `[anf2, ende2)`, so werden höchstens $2 \cdot (n + n_2) - 1$ Vergleiche durchgeführt.

Der Algorithmus `set_intersection`

```
template <class InIt, class InIt2, class OutIt>
OutIt set_intersection( InIt anf, InIt ende, InIt2 anf2, InIt2 ende2,
                       OutIt out);

template <class InIt, class InIt2, class OutIt, class BinPred>
OutIt set_intersection( InIt anf, InIt ende, InIt2 anf2, InIt2 ende2,
                       OutIt out, BinPred comp);
```

Es wird die (sortierte) Schnittmenge der beiden (sortierten) Sequenzen `[anf, ende)` und `[anf2, ende2)` gebildet und diese Schnittmenge in den Ausgabe-Iterator `out` geschrieben (der Ausgabe-Iterator `out` muss entweder auf einen hinreichend großen Container zeigen, dessen ersten Elemente durch die Vereinigungsmenge überschrieben werden, oder ein Inserter sein).

Genauer: Es werden der Sortierreihenfolge nach alle bezüglich des Vergleichs *verschiedenen* Elemente `elem1` der ersten Sequenz betrachtet (d.h. von *gleichen* Elementen

der ersten Sequenz wird nur jeweils eins in Betracht gezogen) — die Vielfachheit des Elementes `elem1` in der ersten Sequenz sei v_1 .

Gibt es in der zweiten Sequenz kein zu `elem1` *gleiches* Element, so wird kein zu `elem1` *gleiches* Element in die Schnittmenge geschrieben.

Gibt es in der zweiten Sequenz ein zu `elem1` *gleiches* Element `elem2` und dieses Element `elem2` habe in der zweiten Sequenz die Vielfachheit v_2 , so wird dann das Minimum $v = \min\{v_1, v_2\}$ berechnet und die (in ihrer ursprünglichen Reihenfolge) ersten v zu `elem1` *gleichen* Elemente der ersten Sequenz werden in die Schnittmenge geschrieben und sonst keine zu `elem1` *gleichen* Elemente.

Ergebnis des Algorithmus ist die Ende-Iteratorposition des Ausgabe-Iterators `out`.

Die Iteratortypen (und dahinter stehende Containertypen) können verschieden sein, die Elementtypen der Iteratoren (und Container) müssen gleich sein und zum Argumenttyp des Vergleichs passen.

Der Zielbereich (`out` und das, was dahinter ist) darf mit keinem der Quellbereiche (erste oder zweite Sequenz) überlappen.

Komplexität:

Sei n die Anzahl der Elemente der ersten Sequenz `[anf, ende)` und n_2 die Anzahl der Elemente der zweiten Sequenz `[anf2, ende2)`, so werden höchstens $2 \cdot (n + n_2) - 1$ Vergleiche durchgeführt.

Der Algorithmus `set_difference`

```
template <class InIt, class InIt2, class OutIt>
OutIt set_difference( InIt anf, InIt ende, InIt2 anf2, InIt2 ende2,
                    OutIt out);
```

```
template <class InIt, class InIt2, class OutIt, class BinPred>
OutIt set_difference( InIt anf, InIt ende, InIt2 anf2, InIt2 ende2,
                    OutIt out, BinPred comp);
```

Es wird die (sortierte) Differenzmenge (erste Menge ohne Elemente der zweiten Menge) der beiden (sortierten) Sequenzen `[anf, ende)` und `[anf2, ende2)` gebildet und diese Differenzmenge in den Ausgabe-Iterator `out` geschrieben (der Ausgabe-Iterator `out` muss entweder auf einen hinreichend großen Container zeigen, dessen ersten Elemente durch die Differenzmenge überschrieben werden, oder ein Inserter sein).

Genauer: Es werden der Sortierreihenfolge nach alle bezüglich des Vergleichs *verschiedenen* Elemente `elem1` der ersten Sequenz betrachtet (d.h. von *gleichen* Elementen der ersten Sequenz wird nur jeweils eins in Betracht gezogen) — die Vielfachheit des Elementes `elem1` in der ersten Sequenz sei v_1 .

Gibt es in der zweiten Sequenz kein zu `elem1` *gleiches* Element, so werden alle v_1 zu `elem1` *gleichen* Elemente (in ihrer ursprünglichen Reihenfolge) in die Differenzmenge geschrieben.

Gibt es in der zweiten Sequenz jedoch ein zu `elem1` *gleiches* Element `elem2`, dieses Element `elem2` habe in der zweiten Sequenz die Vielfachheit v_2 , so wird die Differenz $v = v_1 - v_2$ berechnet. Ist v kleiner oder gleich 0 (d.h. v_2 ist größer oder gleich v_1), so wird in die Differenzmenge kein zu `elem1` *gleiches* Element geschrieben.

Ist jedoch v größer als 0 (v_1 größer als v_2), so werden aus der ersten Sequenz nur die (in der ursprünglichen Reihenfolge) letzten v zu `elem1` gleichen Elemente in die Differenzmenge geschrieben.

Ergebnis des Algorithmus ist die Ende-Iteratorposition des Ausgabe-Iterators `out`. Die Iteratortypen (und dahinter stehende Containertypen) können verschieden sein, die Elementtypen der Iteratoren (und Container) müssen gleich sein und zum Argumenttyp des Vergleichs passen.

Der Zielbereich (`out` und das, was dahinter ist) darf mit keinem der Quellbereiche (erste oder zweite Sequenz) überlappen.

Komplexität:

Sei n die Anzahl der Elemente der ersten Sequenz [`anf`, `ende`) und n_2 die Anzahl der Elemente der zweiten Sequenz [`anf2`, `ende2`), so werden höchstens $2 \cdot (n + n_2) - 1$ Vergleiche durchgeführt.

Der Algorithmus `set_symmetric_difference`

```
template <class InIt, class InIt2, class OutIt>
OutIt set_symmetric_difference( InIt anf, InIt ende,
                               InIt2 anf2, InIt2 ende2, OutIt out);
```

```
template <class InIt, class InIt2, class OutIt, class BinPred>
OutIt set_symmetric_difference( InIt anf, InIt ende,
                               InIt2 anf2, InIt2 ende2, OutIt out, BinPred comp);
```

Es wird die (sortierte) symmetrische Differenz (alle Elemente, die nur in einer der beiden Mengen vorkommen) der beiden (sortierten) Sequenzen [`anf`, `ende`) und [`anf2`, `ende2`) gebildet und diese symmetrische Differenz in den Ausgabe-Iterator `out` geschrieben (der Ausgabe-Iterator `out` muss entweder auf einen hinreichend großen Container zeigen, dessen ersten Elemente durch die symmetrische Differenz überschrieben werden, oder ein Insertor sein).

Genauer: Es werden der Sortierreihenfolge nach alle bezüglich des Vergleichs *verschiedenen* Elemente `elem` betrachtet, die in mindesten einer der beiden Sequenzen vorkommen (d.h. von *gleichen* Elementen wird nur jeweils eins in Betracht gezogen) — die Vielfachheit eines zu `elem` *gleichen* Elementes in der ersten Sequenz sei v_1 und die Vielfachheit eines zu `elem` *gleichen* Elementes in der zweiten Sequenz sei v_2 . (Sollte es in der ersten Sequenz kein zu `elem` *gleiches* Element geben, so ist die Vielfachheit v_1 gleich 0, sollte es in der zweiten Sequenz kein zu `elem` *gleiches* Element geben, so ist die Vielfachheit v_2 gleich 0. Nach Vorgabe muss jedoch mindestens eine der Vielfachheiten v_1 oder v_2 größer als 0 sein!)

Sind v_1 und v_2 gleich, so wird in die symmetrische Differenz kein zum Element `elem` *gleiches* Element hineingeschrieben.

Ist v_1 größer als v_2 , so werden nur die (in ihrer ursprünglichen Reihenfolge) letzten $v = v_1 - v_2$ zum Element `elem` *gleichen* Elemente der ersten Sequenz in die symmetrische Differenz geschrieben und sonst keine zu `elem` *gleichen* Elemente.

Ist v_2 größer als v_1 , so werden nur die (in ihrer ursprünglichen Reihenfolge) letzten $v = v_2 - v_1$ zum Element `elem` *gleichen* Elemente der zweiten Sequenz in die symmetrische Differenz geschrieben und sonst keine zu `elem` *gleichen* Elemente.

Ergebnis des Algorithmus ist die Ende-Iteratorposition des Ausgabe-Iterators `out`. Die Iteratortypen (und dahinter stehende Containertypen) können verschieden sein, die Elementtypen der Iteratoren (und Container) müssen gleich sein und zum Argumenttyp des Vergleichs passen.

Der Zielbereich (`out` und das, was dahinter ist) darf mit keinem der Quellbereiche (erste oder zweite Sequenz) überlappen.

Komplexität:

Sei n die Anzahl der Elemente der ersten Sequenz `[anf, ende)` und n_2 die Anzahl der Elemente der zweiten Sequenz `[anf2, ende2)`, so werden höchstens $2 \cdot (n + n_2) - 1$ Vergleiche durchgeführt.

11.7.7 Algorithmen für Heaps

Die Heap-Datenstruktur ist daraufhin optimiert, das größte Element schnell aufzufinden (und ggf. aus dem Heap zu entfernen) und ein neues Element in den Heap einzufügen.

Wie bei den Algorithmen zur Sortierung und den Mengenalgorithmen muss wiederum ein Vergleich der Elemente definiert sein, hier ist wiederum für jeden Algorithmus eine Version für den Vergleich mittels `<` und eine Version mit einer eigenen Vergleichsfunktion `BinPred comp` vorhanden.

In der STL ist eine Sequenz `[anf, ende)` mit n Elementen ist genau dann ein Heap, wenn `*anf` das (bzgl. des Vergleichs) größte ist und dieses größte Element mit (höchstens) $\mathcal{O}(n)$ Operationen aus der Sequenz entfernt werden kann oder ein neues Element ebenfalls mit höchstens $\mathcal{O}(n)$ Operationen in die Sequenz eingefügt werden kann, so dass die resultierende Sequenz jeweils wiederum ein Heap ist.

Der Algorithmus `make_heap`

```
template <class RanIt>
void make_heap( RanIt anf, RanIt ende);

template <class RanIt, class BinPred>
void make_heap( RanIt anf, RanIt ende, BinPred comp);
```

Die Elemente der Sequenz `[anf, ende)` werden so permutiert, dass aus der Sequenz (bzgl. des geg. Vergleichs) ein Heap wird.

Komplexität:

Sei n die Anzahl der Elemente der ersten Sequenz `[anf, ende)` und so werden höchstens $3 \cdot n$ Vergleiche durchgeführt.

Der Algorithmus `push_heap`

```
template <class RanIt>
void push_heap( RanIt anf, RanIt ende);

template <class RanIt, class BinPred>
void push_heap( RanIt anf, RanIt ende, BinPred comp);
```


Es wird davon ausgegangen, dass die ersten $n-1$ Elemente der n -elementigen Sequenz `[anf, ende)` (bezüglich des Vergleichs) einen Heap bilden und das verbleibende n -te Element (letztes Element der Sequenz) ein beliebiges Element ist.

Dieses letzte Element wird so in die Sequenz einsortiert, dass die resultierende Sequenz `[anf, ende)` insgesamt ein Heap ist.

Komplexität:

Sei n die Anzahl der Elemente der ersten Sequenz `[anf, ende)` und so werden höchstens $\ln(n)$ Vergleiche durchgeführt.

Der Algorithmus `pop_heap`

```
template <class RanIt>
void pop_heap( RanIt anf, RanIt ende);

template <class RanIt, class BinPred>
void pop_heap( RanIt anf, RanIt ende, BinPred comp);
```

Es wird davon ausgegangen, dass die (n -elementige) Sequenz `[anf, ende)` (bezüglich des Vergleichs) einen Heap bilden.

Die Elemente des Heaps werden so permutiert, dass das (bezüglich des Vergleichs) größte Element der Heaps an die letzte (n -te) Position der Sequenz gelangt und die verbleibenden ersten $n-1$ Elemente der Sequenz wiederum (bzgl. des Vergleichs) einen Heap bilden.

Komplexität:

Sei n die Anzahl der Elemente der ersten Sequenz `[anf, ende)` und so werden höchstens $2 \cdot \ln(n)$ Vergleiche durchgeführt.

Der Algorithmus `sort_heap`

```
template <class RanIt>
void sort_heap( RanIt anf, RanIt ende);

template <class RanIt, class BinPred>
void sort_heap( RanIt anf, RanIt ende, BinPred comp);
```

Es wird davon ausgegangen, dass die (n -elementige) Sequenz `[anf, ende)` (bezüglich des Vergleichs) einen Heap bilden.

Die Elemente des Heaps werden so permutiert, dass die Sequenz anschließend (bezüglich des Vergleichs) aufsteigend sortiert ist. (Die Sequenz ist also anschließend kein Heap mehr!)

Komplexität:

Sei n die Anzahl der Elemente der ersten Sequenz `[anf, ende)`, so werden höchstens $n \cdot \ln(n)$ Vergleiche durchgeführt.

11.7.8 Minimum, Maximum und Vergleich

Folgenden Algorithmen zur Bestimmung des Minimums bzw. Maximums zweier Elemente bzw. einer Sequenz und zum Vergleich zweier Sequenzen liegt wiederum der

Vergleich der Elemente mit `<` oder ein selbstdefinierter Vergleich mit einem binären Prädikat `BinPred comp` zugrunde.

Die Algorithmen `min` und `max`

```
template <class T>
const T& min( const T& a, const T& b);

template <class T>
const T& max( const T& a, const T& b);

template <class T, class BinPred>
const T& min( const T& a, const T& b, BinPred comp);

template <class T, class BinPred>
const T& min( const T& a, const T& b, BinPred comp);
```

Die `min`-Funktionen geben eine Referenz (auf `const`) auf das (bzgl. des Vergleichs) kleinere der beiden Elemente `a` und `b` zurück, die `max`-Funktionen eine Referenz (auf `const`) auf das größere.

Sind beide Elemente (bzgl. des Vergleichs) *gleich*, wird jeweils eine Referenz (auf `const`) auf das erste Argument zurückgegeben.

Die Algorithmen `min_element` und `max_element`

```
template <class ForIt>
ForIt min_element( ForIt anf, ForIt ende);

template <class ForIt>
ForIt max_element( ForIt anf, ForIt ende);

template <class ForIt, class BinPred>
ForIt min_element( ForIt anf, ForIt ende, BinPred comp);

template <class ForIt, class BinPred>
ForIt min_element( ForIt anf, ForIt ende, BinPred comp);
```

Diese Algorithmen geben die Iteratorposition auf das (bzgl. des Vergleichs) kleinste bzw. größte Element der Sequenz `[anf, ende)` zurück.

Sollten mehrere kleinste bzw. größte Elemente vorhanden sein, wird die Position des in der ursprünglichen Sequenz ersten kleinsten bzw. größten Elementes zurückgegeben.

Komplexität:

Sei n die Anzahl der Elemente der Sequenz `[anf, ende)` und $n > 0$, so werden genau $n - 1$ Vergleiche durchgeführt.

Der Algorithmus `lexicographical_compare`

```
template <class InIt, class InIt2>
bool lexicographical_compare( InIt anf, InIt ende, InIt2 anf2,
                             InIt2 ende2);

template <class InIt, class InIt2, class BinPred>
bool lexicographical_compare( InIt anf, InIt ende, InIt2 anf2,
                             InIt2 ende2, BinPred comp);
```

Gibt `true` zurück, falls die erste Sequenz `[anf, ende)` bezüglich des durch den Elementvergleich (`<` bzw. `comp`) auf Sequenzen induzierten *lexikographischen* Vergleich kleiner als die zweite Sequenz `[anf2, ende2)` ist.

Bei diesem *lexikographischen* Vergleich der Sequenzen wird, beginnend mit $i = 1$, das i -te Element `elem1_i` der ersten Sequenz mit dem i -ten Element `elem2_i` der zweiten Sequenz verglichen.

Ist `elem1_i < elem2_i` bzw. `comp(elem1_i, elem2_i) == true`, liefert der Algorithmus `true` zurück.

Ist `elem2_i < elem1_i` bzw. `comp(elem2_i, elem1_i) == true`, liefert der Algorithmus `false` zurück.

Gilt weder

`elem1_i < elem2_i` bzw. `comp(elem1_i, elem2_i) == true` noch

`elem2_i < elem1_i` bzw. `comp(elem2_i, elem1_i) == true`

(d.h. `elem1_i` und `elem2_i` sind bzgl. des Vergleichs *gleich*), so wird i um eins erhöht und die jeweils nächsten Elemente der Sequenzen werden miteinander verglichen.

Kommt so bis zum Ende einer der Sequenzen kein Ergebnis zustande (alle bislang verglichenen Elemente sind *gleich*), so wird, falls die erste Sequenz weniger Elemente als die zweite hat, `true` zurückgegeben und ansonsten `false`.

Komplexität:

Sei n die Anzahl der Elemente der ersten Sequenz `[anf, ende)` und $n2$ die Anzahl der Elemente der zweiten Sequenz `[anf2, ende2)` und $m = \min\{n, n2\}$ das Minimum von n und $n2$, so werden höchstens m Vergleiche durchgeführt.

11.8 Die numerische Bibliothek

11.8.1 Numerische Standardfunktionen

In der Headerdatei `<cmath>` werden für die Gleitkommatypen `float`, `double` und `long double` die “üblichen” mathematischen Funktionen deklariert.

Sei T einer dieser drei Typen, so existieren folgende Funktionen:

<code>T abs(T d);</code>	Absolutbetrag
<code>T fabs(T d);</code>	Absolutbetrag
<code>T ceil(T d);</code>	kleinster Integer nicht kleiner als d
<code>T floor(T d);</code>	größter Integer nicht größer als d

<code>T sqrt(T d);</code>	Quadratwurzel aus <code>d</code> (darf nicht negativ sein)
<code>T pow(T d, Te);</code>	<code>d</code> hoch <code>e</code> , Fehler, falls <code>d</code> gleich 0 und <code>e</code> negativ bzw. falls <code>d</code> kleiner 0 und <code>e</code> nicht ganzzahlig
<code>T pow(T d, int i);</code>	<code>d</code> hoch <code>i</code> , Fehler, falls <code>d</code> gleich 0 und <code>i</code> negativ
<code>T sin(T d);</code>	Sinus
<code>T cos(T d);</code>	Cosinus
<code>T tan(T d);</code>	Tangens
<code>T asin(T d);</code>	Arcus-Sinus
<code>T acos(T d);</code>	Arcus-Cosinus
<code>T atan(T d);</code>	Arcus-Tangens
<code>T atan2(T x, T y);</code>	entspricht <code>atan(x/y)</code>
<code>T sinh(T d);</code>	Sinus hyperbolicus
<code>T cosh(T d);</code>	Cosinus hyperbolicus
<code>T tanh(T d);</code>	Tangens hyperbolicus
<code>T exp(T d);</code>	Exponentialfunktion
<code>T log(T d);</code>	natürlicher Logarithmus (<code>d</code> muss größer 0 sein!)
<code>T log10(T d);</code>	Logarithmus zur Basis 10 (<code>d</code> muss größer 0 sein!)
<code>T modf(T d, T* p);</code>	Nachkommanteil von <code>d</code> als Ergebnis, ganzzahliger Teil nach <code>*p</code>
<code>T frexp(T d, int* p);</code>	Zahl <code>x</code> im Intervall $[0.5, 1)$ und ganzzahliges <code>y</code> finden mit <code>d = x * pow(2,y)</code> , <code>x</code> als Funktionsergebnis und <code>y</code> nach <code>*p</code>
<code>T fmod(T d, T* m);</code>	Rest der Gleitkommadivision von <code>d/m</code> , gleiches Vorzeichen wie <code>d</code> . (<code>m</code> darf nicht 0 sein!)
<code>T ldexp(T d, int i);</code>	liefert <code>d * pow(2,i)</code>

11.8.2 Komplexe Zahlen

Die Templateklasse `complex<T>` zur Realisierung komplexer Zahltypen sind in der Headerdatei `<complex>` definiert.

Der Datentyp `T` ist der Typ der Komponenten (Real- und Imaginärteil) der komplexen Zahlen — standardmäßig werden nur die drei Standardtypen `float`, `double` und `long double` unterstützt.

Erzeugung, Zuweisung komplexer Zahlen

Bei der Erzeugung einer komplexen Zahl mit Komponententyp `T` können unterschiedliche Argumente angegeben werden:

- Kein Argument:
Real- und Imaginärteil erhalten den Wert 0.
- Ein Argument vom Typ `T`:
Der Realteil erhält als Wert den Wert des Argumentes, Imaginärteil ist 0.
- Zwei Argumente vom Typ `T`:
Der Realteil wird mit dem ersten und Imaginärteil mit dem zweiten Wert vorbesetzt.

- Ein Argument vom Typ `complex<T>`:
Copy-Konstruktor, neues Element ist Kopie des Argumentes.

Die Funktion

```
complex<T> polar( T abs, T phi);
```

liefert als (temporäres) Ergebnis die aus den “Polarkoordinaten“ (`abs`, `phi`) berechnete komplexe Zahl (Absolutbetrag `abs` und Winkel `phi` im Bogenmaß).

Die Funktion

```
complex<T> conj(complex<T> c);
```

liefert als (temporäres) Ergebnis die zu `c` konjugiert komplexe Zahl (gleicher Realteil, negierter Imaginärteil).

Die Zuweisungsoperatoren

```
// T zuweisen:
```

```
complex<T>& complex<T>::operator= (const T& w);
```

```
// complex<T> zuweisen
```

```
complex<T>& complex<T>::operator= (const complex<T> & w);
```

ist ebenfalls definiert, der wie üblich zu verwenden ist:

```
...
complex<float> c1, c2;
float w;
...
c2 = w; // c2 erhaelt w als Realteil und 0 als Imaginärteil
c1 = c2; // c1 erhaelt Wert von c2, alter Wert von c1 geht verloren
...
```

`c1` und `c2` brauchen hierbei nicht einmal den gleichen Komponententyp zu haben — ggf. wird eine Typangleichung vorgenommen.

Zugriff auf komplexe Zahlen

Auf die Komponenten einer komplexen Zahl `c` kann man mit folgenden Funktionen zugreifen:

Funktion	Bedeutung
<code>c.real()</code>	liefert Wert des Realteils (Memberfunktion)
<code>real(c)</code>	liefert Wert des Realteils (globale Funktion)
<code>c.imag()</code>	liefert Wert des Imaginärteils (Memberfunktion)
<code>imag(c)</code>	liefert Wert des Imaginärteils (globale Funktion)

Diese Zugriffe sind nur lesend, d.h. hierüber kann man den Wert einer Komponente **nicht** abändern:

```
c.real() = 5.5;    // FEHLER: Funktion liefert nur lesenden Zugriff
```

Will man eine Komponente abändern, so muss man trotzdem der gesamten komplexen Zahl einen neuen Wert zuweisen:

```
c = complex( 5.5, c.imag()); // neuer Realteil, alter Imaginärteil
```

Desweiteren stehen folgende Funktionen zur Verfügung:

Funktion	Bedeutung
<code>c.norm()</code>	liefert (euklidische) Norm von <code>c</code> (Summe der Quadrate aus Realteil und Imaginärteil)
<code>c.abs()</code>	liefert Absolutbetrag von <code>c</code> (Quadratwurzel aus der Norm)
<code>c.arg()</code>	liefert Winkel (im Bogenmaß) der Polardarstellung von <code>c</code> .

Vergleiche

Für komplexe Zahlen stehen die Vergleichsoperatoren `==` und `!=` zur Verfügung — die übrigen Vergleichsoperationen `<`, `>`, `<=` und `>=` sind **nicht** definiert!

Dies liegt daran, dass ein Vergleich `<` mit der bei gewöhnlichen Zahlen üblichen Bedeutung für komplexe Zahlen aus mathematischen Gründen nicht definiert werden kann!

Aufgrund des fehlenden Vergleichsoperators `<` können komplexe Zahlen nicht ohne weiteres in Mengen (`set` oder `multiset`) verwaltet oder als Schlüssel in Maps (`map` oder `multimap`) verwendet werden, da hierzu der Vergleichsoperator `<` notwendig ist. Wegen des entsprechenden Konstruktors kann eine komplexe Zahl auch mit einem Skalar verglichen werden:

```
...
complex<double> c1,c2;
double x;
...
// Vergleich zwischen
if ( c1 == c2) ... // komplex und komplex
if ( c1 != c2) ... // komplex und komplex
if ( c1 == x ) ... // komplex und skalar
if ( c1 != x ) ... // komplex und skalar
if ( x == c1) ... // skalar und komplex
if ( x != c1) ... // skalar und komplex
...
```

Arithmetische Operationen

Beide Vorzeichenoperatoren `+` und `-` sind für komplexe Zahlen verfügbar, der Ausdruck `-c` hat eine komplexe Zahl als Wert, deren Real- und Imaginärteil jeweils dem von `c` mit anderem Vorzeichen entsprechen.

Die für Zahlen üblichen Grundrechenarten sind für komplexe Zahlen natürlich ebenfalls definiert:

Addition	+	Subtraktion	-
Multiplikation	*	Division	/

Diese Operatoren funktionieren auch, wenn einer der beteiligten Operanden ein Skalar ist!

Bei den für komplexe Zahlen ebenfalls vorhandenen arithmetischen Zuweisungen `+=`, `-=`, `*=` und `/=` muss der linke Operand natürlich komplex sein.

Ein-/Ausgabe komplexer Zahlen

Für komplexe Zahlen sind Ausgabeoperator `<<` und Eingabeoperator `>>` definiert. Der Ausgabeoperator `<<` gibt eine komplexe Zahl in der Form:

```
( realteil, imaginaerteil)
```

aus.

Der Eingabeoperator `>>` akzeptiert folgende Eingaben:

```
( realteil, imaginaerteil)
```

```
( realteil )
```

```
realteil
```

Ist kein Imaginärteil angegeben, so wird dieser als 0 angenommen.

Transzendente Funktionen für komplexe Zahlen

Für komplexe Zahlen (mit `float`, `double` oder `long double` als Komponententyp) sind die Funktionen:

Exponentialfunktion	<code>exp(c)</code>
natürlicher Logarithmus (Hauptzweig)	<code>log(c)</code>
10-er Logarithmus	<code>log10(c)</code>
Potenzfunktion entspricht	<code>pow(c1, c2)</code> <code>exp(log(c2)* c1)</code>
Quadratwurzel (welche?)	<code>sqrt(c)</code>
Sinusfunktion	<code>sin(c)</code>
Cosinusfunktion	<code>cos(c)</code>
Tangensfunktion	<code>tan(c)</code>
Sinus hyperbolicus	<code>sinh(c)</code>
Cosinus hyperbolicus	<code>cosh(c)</code>
Tangens hyperbolicus	<code>tanh(c)</code>

(Hierbei ist `c` eine komplexe Zahl und mindestens eine der Zahlen `c1` oder `c2` ist ebenfalls komplex — die andere kann auch komplex oder aber vom zugrundeliegenden Skalartyp sein.)

11.8.3 Mathematische Vektoren

Die Templateklasse `valarray<T>` stellt Vektoren zur Verfügung, deren Komponenten von einem gewissen "Zahltyp" `T` sind und bei denen umfangreiche arithmetische Operationen definiert sind, die auf die entsprechenden Operationen der Komponenten zurückgeführt werden.

Der zugrundeliegende Komponententyp `T` ist üblicherweise ganzzahlig oder `float`, `double` oder `long_double` — aber es sollte auch (weitgehend) mit komplexen Zahlen (siehe Abschnitt 11.8.2) funktionieren!

(Diese Templateklasse wird von unserer C++-Umgebung noch nicht zur Verfügung gestellt!)

Zur Verwendung von `valarray`'s muss die Headerdatei `<valarray>` includet werden.

Erzeugung und Größe eines `valarray`'s

Die Erzeugung eines `valarray<T>` kann wie die eines `vektor<T>` erfolgen (Elementtyp ist jeweils `T`):

- `valarray<T> a;`
Erzeugt einen zunächst leeren mathematischen Vektor (Länge 0).
- `valarray<T> a(n);`
Erzeugt einen mathematischen Vektor der Länge `n`. Alle `n` Komponenten des Vektors vom Typ `T` werden mit dem `T`-Standardkonstruktor erzeugt.
- `valarray<T> a(elem, n);`
Erzeugt einen mathematischen Vektor der Länge `n`, wobei alle `n` Komponenten Kopien des angegebenen Objektes `elem` vom Typ `T` sind. (Man beachte die Reihenfolge der Argumente!)

Neben dem üblichen Copy-Konstruktor kann man ein `valarray<T>` mit einem (eingebauten) Feld vom Typ `T` als Argument erzeugen:

```
...
T T_Feld[100];      // eingebautes Feld vom Typ T
...                // Komponenten von T_Feld irgendwie belegen
...
valarray<T> a(T_Feld, 100);
...
```

hierbei wird `a` als `valarray<T>` mit 100 Elementen erzeugt, wobei diese 100 Komponenten von `a` Kopien der (ersten) 100 Elemente des Feldes `T_Feld` sind. (Die Länge des eingebauten Feldes darf größer als die als zweites Argument angegebene Länge sein.)

Daneben gibt es eine Reihe von anderen Möglichkeiten, einen neuen `valarray<T>` als Teil eines vorhandenen `valarray<T>` zu erzeugen. Hierauf wird in den Abschnitten 11.8.4, 11.8.5, 11.8.6 und 11.8.7 näher eingegangen.

Wie bei einem "gewöhnlichen" `vektor<T>` kann man die Größe (Anzahl der Elemente) eines `valarray<T>` `a(...)` mittels der Elementfunktion `a.size()` ermitteln. (Eine Elementfunktion `empty()` zur Abfrage, ob ein `valarray<T>` leer ist oder nicht, **existiert nicht!**)

Die Funktion zum Leeren eines `valarray<T>` `a(...)` heißt `a.free()`, die Anzahl der Elemente ist anschließend 0. (Bei Vektoren und anderen Containerklassen heißt diese Funktion `clear()`!)

Wie bei `vektor<T>` kann man die Größe eines `valarray<T>` `a(...)` ändern:

- `a.resize(n);` Ändert die Größe von `a` auf `n`. Ist `n` kleiner als die bisherige Größe, werden am Ende von `a` entsprechend viele Elemente gelöscht. Ist `n` größer als die bisherige Größe, werden am Ende entsprechend viele neue Elemente vom Typ `T` (mit dem zugehörigen Standardkonstruktor) erzeugt.
- `a.resize(n,elem);`
Ändert die Größe von `a` auf `n`. Ist `n` kleiner als die bisherige Größe, werden am Ende entsprechend viele Elemente gelöscht. Ist `n` größer als die bisherige Größe, werden am Ende entsprechend viele neue Elemente vom Typ `T` als Kopien des angegebenen Objektes `elem` (vom Typ `T`) erzeugt.

Die bei `vector<T>` vorhandenen Funktionen `push_front(...)` und `pop_front()` zum Einfügen bzw. Entfernen eines Elementes am Ende des Vektors **existieren** bei `valarray`'s **nicht**!

Zuweisungen und Indizierung

Zwei `valarray`'s vom gleichen Elementtyp **und gleicher Länge** können einander zugewiesen werden:

```
valarray<T> a(100), b(100), c(200);    // drei Valarrays
...
a = b;                                // ok, jedes Element von b wird dem entsprechenden
                                     // Element von a zugewiesen
b = c;                                // FEHLER: unterschiedliche Laenge!
...
```

Darüberhinaus kann einem `valarray<T>` ein Objekt vom Typ `T` zugewiesen werden, wobei jedes Element des Valarrays den rechts stehenden Wert erhält:

```
double elem = 3.14;
valarray<double> a(100);
...
a = elem;    // alle 100 Elemente von a erhalten den Wert 3.14
...
```

Wie bei der Erzeugung von `valarray`'s ist es ebenfalls möglich, einem `valarray<T>` einen Teil eines anderen `valarray<T>` zuzuweisen! Hierauf wird in den Abschnitten 11.8.4, 11.8.5, 11.8.6 und 11.8.7 näher eingegangen.

Der Indexzugriff mittels des `[]`-Operators ist wie bei Vektoren ungeprüft:

```
valarray<T> a(100);
...
...a[0]...;           // Zugriff auf's erste Element
...a[99]...;          // Zugriff auf's letzte Element
...a[200]...;         // FEHLER, vom Compiler nicht erkannt
...a[-23]...;         // FEHLER, vom Compiler nicht erkannt
...
```

In den Abschnitten 11.8.4, 11.8.5, 11.8.6 und 11.8.7 wird beschrieben, wie man mittels Indizierung, indem man etwas anderes als ganzzahlige Indizes verwendet, einen größeren Teil (nicht nur ein Element) aus einem `valarray<T>` extrahieren kann!

Operatoren für `valarray`'s

Für mathematische Vektoren (`valarray<T>`) sind eine ganze Reihe von Operationen definiert, die auf die entsprechenden Operationen des zugrundeliegenden Komponententyps `T` (sollte ein Standardtyp sein!) zurückgeführt werden — insbesondere müssen diese Operationen für den Typ `T` definiert sein, ansonsten meldet der Compiler bei der Spezialisierung der entsprechenden Templates (meist mehrere) Fehler.

Unäre Operationen für mathematische Vektoren

Ist *unary-op* einer der folgenden unären Operatoren:

+	unäres Plus	-	unäres Minus
~	Bit-Komplement	!	Negation

und `a` ein `valarray<T>` und der Operator *unary-op* für diesen Standardtyp `T` (mit Ergebnis ebenfalls vom Typ `T` bzw. Typ `bool` beim Operator `!`) definiert, so ist dieser Operator *unary-op* auch auf `a` anwendbar:

```
...unary-op a...
```

und das Ergebnis dieser Operation ist ein temporäres `valarray<T>` mit der gleichen Länge wie `a` und die einzelnen Elemente dieses temporären `valarray<T>` sind die Ergebnisse der Anwendung des Operators auf die korrespondierenden Elemente von `a`!

Beispiel:

```
...
valarray<double> a(100), b(100), c(200);
...
a = -b; // ok: Werte von a werden die negierten Werte von b
a = ~b; // FEHLER: Operator ~ fuer double nicht definiert!
a = -c; // FEHLER: bei Zuweisung muessen Laengen uebereinstimmen!
...
valarray<int> d(10), e(10);
...
d = ~e; // ok: Operator ~ fuer int definiert,
        // Werte von d sind Komplemente der Werte von e
...
```

Binäre Operationen für mathematische Vektoren

binary-op sei einer der binären Operatoren:

Operator	Operation	Operator	Operation
+	Additions	-	Subtraktion
*	Multiplikation	/	Division
%	Modulo	^	Bit-Exklusives Oder
&	Bit-Und		Bit-Inklusives Oder
<<	Links-Shift	>>	Rechts-Shift
&&	Logisches Und		logisches Oder

und sei T ein Datentyp, für den der Operator *binary-op* definiert ist. Weiterhin seien $a1$ und $a2$ zwei `valarray<T>`'s **der gleichen Länge** und $elem$ ein Objekt vom Typ T .

Dann sind die Operationen:

```
a1 binary-op a2
a1 binary-op elem
elem binary-op a1
```

definiert.

Die Operation $a1 \text{ *binary-op* } a2$ liefert ein (temporäres) `valarray<T>`, in dem jedes Element das Ergebnis der Operation der entsprechenden Elemente von $a1$ und $a2$ ist. Bei den Operationen $a1 \text{ *binary-op* } elem$ bzw. $elem \text{ *binary-op* } a1$ wird ein (temporäres) `valarray<T>` erzeugt, in dem jedes Element die Verknüpfung des Wertes $elem$ mit dem entsprechenden Element von $a1$ ist.

Bei den Operationen `&&` und `||` ist das Ergebnis ein `valarray<bool>`.

Entsprechend sind die Zuweisungsoperatoren

```
+= , -= , *= , /= , %= , ^= , &= , |= , <=<= , >>=
```

wie üblich definiert, d.h. ist *op* einer der Operatoren `+`, `-`, `*`, `/`, `%`, `^`, `&`, `|`, `<<` oder `>>`, so ist

```
a1 op= a2
```

eine abkürzende Schreibweise für

```
a1 = a1 op a2
```

Hierbei ist $a1$ ein `valarray<T>` und $a2$ ebenfalls ein gleich langes `valarray<T>` oder ein Objekt vom Typ T .

Vergleichsoperatoren für mathematische Vektoren

Die Vergleichsoperatoren

<code>==</code>	Test auf Gleichheit
<code>!=</code>	Test auf Ungleichheit
<code><</code>	Test auf kleiner
<code><=</code>	Test auf kleiner oder gleich
<code>></code>	Test auf größer
<code>>=</code>	Test auf größer oder gleich

sind ebenfalls für `valarray<T>`'s definiert (wenn der entsprechende Vergleich für den Elementtyp T definiert ist — ist für alle Standardtypen, nicht jedoch für komplexe Zahlen der Fall!).

Ist nun *vgl-op* einer dieser Vergleichsoperatoren und sind $a1$ und $a2$ zwei `valarray<T>`'s **der gleichen Länge** und ist $elem$ ein Objekt vom Typ T , so sind die Vergleiche

```
a1 vgl-op a2
a1 vgl-op elem
elem vgl-op a1
```

definiert und das Ergebnis ist ein `valarray<bool>` der gleichen Länge wie $a1$, dessen Komponenten die Ergebnisse des entsprechenden komponentenweisen Vergleiches der Komponenten von $a1$ mit $a2$ bzw. mit $elem$ sind.

Weitere Elementfunktionen zur Klasse `valarray<T>`

Folgende drei Funktionen liefern ein Ergebnis vom Typ `T`:

- `sum()`
ermittelt die Summe aller Elemente des Vektors.
- `min()`
ermittelt das Minimum aller Elemente des Vektors.
- `max()`
ermittelt das Maximum aller Elemente des Vektors.

(Da es sich um Elementfunktionen handelt, müssen sie für ein `valarray<T> a(...)`; in der Form `a.sum()`, `a.min()` bzw. `a.max()` aufgerufen werden! Natürlich muss für `sum` der Additionsoperator `+` und für `min` bzw. `max` der Vergleichsoperator `<` für Elemente vom Typ `T` definiert sein — ist für alle Standardtypen, nicht jedoch für komplexe Zahlen der Fall!)

Die Funktion

```
a.shift(n);
```

liefert als Ergebnis einen (temporären) `valarray<T>` der gleichen Länge wie `valarray<T> a`, in dem die Elemente von `a` um `n` Positionen nach “vorne“, falls `n` positiv ist (d.h. das erste Element des neuen `valarray`’s erhält den Wert von `a[n]`, das zweite den Wert von `a[n+1]` usw.) bzw. nach “hinten“ (Gegenrichtung), wenn `n` negativ ist.

Beim Shift nach “vorne“ (`n` positiv) wird “hinten“ mit `T`-Standardwerten aufgefüllt, beim Shift nach “hinten“ wird “vorne“ mit `T`-Standardwerten aufgefüllt.

Die Funktion

```
a.cshift(n);
```

liefert als Ergebnis einen (temporären) `valarray<T>` der gleichen Länge wie `valarray<T> a`, bei dem die Elemente aus `a` um `n` Positionen zyklisch vertauscht werden, d.h. beim Shift nach “vorne“ (`n` positiv) werden die “vorne herausfallenden“ Elemente “hinten“ wieder eingefügt und beim Shift nach “hinten“ entsprechend die “hinten herausfallenden“ Elemente “vorne“ wieder eingefügt.

Ist `fkt` eine Funktion (oder ein Funktionsobjekt), welches für Objekte `elem` vom Typ `T` “aufgerufen“ werden kann und ein Ergebnis vom Typ `T` liefert (d.h. der Ausdruck `fkt(elem)` wird vom Compiler verstanden und der Ergebnistyp dieses Ausdrucks ist wiederum `T`), so wird bei

```
a.apply( fkt)
```

`fkt` auf jedes Element des `valarray<T> a(...)`; “angewendet“ und die jeweiligen Ergebnisse in einem (temporären) `valarray<T>` der gleichen Länge wie `a` zurückgegeben!

Mathematische Funktionen

Folgende (einargumentige) Mathematische Funktionen können auf ein `valarray<T> a` angewendet werden, wenn diese mathematische Funktion für den Elementtyp `T` definiert ist:

Funktion	Bedeutung	Funktion	Bedeutung
<code>abs(a)</code>	Absolutbetrag	<code>exp(a)</code>	Exponentialfunktion
<code>sqrt(a)</code>	Quadratwurzel	<code>log(a)</code>	natürlicher Logarithmus
<code>log10(a)</code>	Logarithmus zur Basis 10	<code>sin(a)</code>	Sinus-Funktion
<code>cos(a)</code>	Cosinus-Funktion	<code>tan(a)</code>	Tangens-Funktion
<code>sinh(a)</code>	Sinus hyperbolicus	<code>cosh(a)</code>	Cosinus hyperbolicus
<code>tanh(a)</code>	Tangens hyperbolicus	<code>asin(a)</code>	Arcus Sinus
<code>acos(a)</code>	Arcus Cosinus	<code>atan(a)</code>	Arcus Tangens

Funktionsergebnis ist jeweils ein `valarray<T>` der gleichen Länge wie `a`, dessen Komponenten die Funktionsergebnisse der jeweiligen Anwendung der Funktion auf die Komponenten von `a` sind.

Auch die zweiargumentigen mathematischen Funktionen `pow` und `atan2` werden auf komponentenweises Rechnen zurückgeführt. Ergebnis ist jeweils wieder ein temporärer mathematischer Vektor.

In folgender Tabelle seien `a` und `b` zwei (gleichlange) `valarray<T>` und `elem` ein Objekt vom Typ `T`:

<code>pow(a,b)</code>	<code>k</code> -te Komponente des Ergebnisses ist <code>pow(a[k] , b[k])</code>
<code>pow(a, elem)</code>	<code>k</code> -te Komponente des Ergebnisses ist <code>pow(a[k] , elem)</code>
<code>pow(elem, a)</code>	<code>k</code> -te Komponente des Ergebnisses ist <code>pow(elem, a[k])</code>
<code>atan2(a,b)</code>	<code>k</code> -te Komponente des Ergebnisses ist <code>atan2(a[k] , b[k])</code>
<code>atan2(a,elem)</code>	<code>k</code> -te Komponente des Ergebnisses ist <code>atan2(a[k] , elem)</code>
<code>atan2(elem,a)</code>	<code>k</code> -te Komponente des Ergebnisses ist <code>atan2(elem, a[k])</code>

11.8.4 Slices zu einem `valarray`

Im Zusammenhang mit `valarray`'s ist der Hilfs-Datentyp `slice` definiert. Ein Objekt dieses Types besteht (im Wesentlichen) aus drei vorzeichenlosen ganzzahligen Werten, die mittels des Konstruktors:

```
slice scheibe (start, size, stride);
```

gesetzt werden und mittels der Elementfunktionen:

```
scheibe.start();      // gibt anfang zurueck
scheibe.size();       // gibt anzahl zurueck
scheibe.stride();     // gibt abstand zurueck
```

abgefragt werden können.

Die Hauptanwendung eines `slice` erfolgt in Zusammenhang mit einem `valarray<T>` und Indizierung mit `[]`, hierbei wird im Allgemeinen nur ein temporäres `slice`-Objekt verwendet.

Ist `v` ein (genügend großes) `valarray<T>` so liefert die Operation

```
v[ slice(anf, anz, abst) ]
```

Zugriff auf die Elemente:

```
v[anf], v[anf+abst], v[anf+2*abst], ..., v[anf+(anz-1)*abst]
```

von `v`, konkret:

Es sei der Valarray `v` wie folgt definiert:

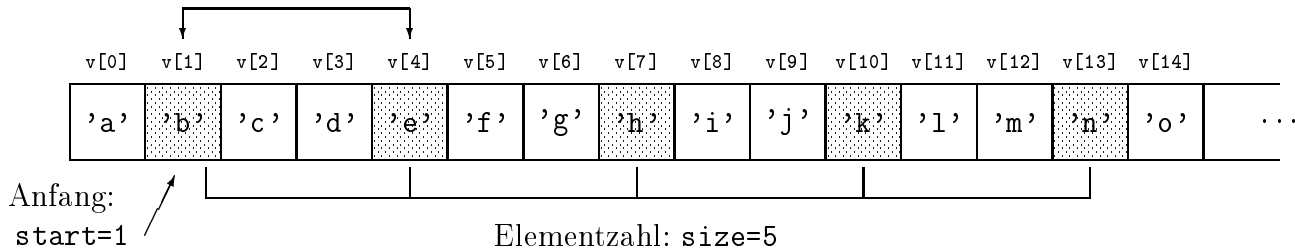
```
valarray<char> v("abcdefghijklmnopqrstuvwxyz",26);
```

durch

```
v[ slice(1, 5, 3) ]
```

erhält man Zugriff auf die in folgendem Bild schraffierten Elemente von `v`:

jeweiliger Abstand: `stride=3`



Der genaue Typ des Ergebnisses der Operation `v[slice(anf, anz, abst)]` ist hierbei ein sogenanntes `slice_array<T>` — ein ebenfalls zu `valarray<T>` gehörender Hilfstyp.

Die Indizierung eines `valarray<T>`'s mit einem `slice` ist die einzige Möglichkeit, ein `slice_array<T>` zu erzeugen (es sind keine Konstruktoren verfügbar) und ein `slice_array<T>` ist immer in Zusammenhang mit dem `valarray<T>` zu sehen, aus dem es mittels Indizierung erzeugt wurde: das `slice_array` bietet Zugriff auf die (durch das `slice` ausgewählten) Elemente des zugrundeliegenden `valarray<T>`. Natürlich kann man über diesen `slice_array`-Zugriff die Werte des `valarray`'s nur dann ändern, wenn der `valarray` **nicht konstant** ist. (Formal ist das Ergebnis der Indizierung einen **konstanten** `valarray`'s mit einem `slice` ein temporäres `valarray` und kein `slice_array`!)

Zu einem `slice_array<T>` existiert beispielsweise eine Elementfunktion `void fill(T&);`, die allen Elementen des `slice_array<T>` das angegebene Argument zuweist. Hierdurch erhalten somit einige (die durch das `slice` ausgewählten) Elemente des zugrundeliegenden `valarray<T>` einen neuen Wert!

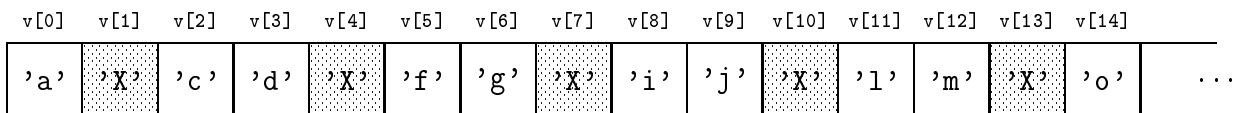
Man könnte z.B. für unser oben definiertes

```
valarray<char> v("abcdefghijklmnopqrstuvwxyz",26);
```

diese Funktion wie folgt aufrufen:

```
v.[ slice(1,5,3) ].fill('X');
```

und hätte in `v` die durch den `slice` bestimmten Elemente geändert, die neue Belegung von `v` sieht dann so aus:



Operationen für `slice_array`'s

Neben der oben schon beschriebenen Elementfunktion `fill` zum Besetzen aller Elemente des `slice_array`'s mit einem neuen Wert existieren für ein `slice_array<T>`

noch die Zuweisungsoperatoren `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `&=`, `|=`, `<<=` und `>>=`, wobei auf der rechten Seite des Operators ein `valarray<T>` der richtigen Größe stehen muss.

Die Bedeutung dieser Operatoren entspricht der bei `valarray`'s: die entsprechende Operation wird komponentenweise für die durch das `slice_array` bestimmten Elemente des zugrundeliegenden `valarray`'s durchgeführt.

Beispiel:

```
...
valarray<double> a( 5.0, 10);    // Laenge 10, alle Wert 5.0
valarray<double> b( 3.0, 5);    // Laenge 5, alle Werte 3.0
...
a[ slice(0,5,2) ] += b;        // in a wird zu jeder Komponente mit
                                // geradem Index 3.0 addiert
...
a[ slice(1,5,2) ] *= b;        // in a wird jede Komponente mit
                                // ungeradem Index mit 3.0 multipliziert
...
```

Operationen mit `slice_array`'s

Bei der Erzeugung (Konstruktor) eines neuen `valarray<T>` kann ein `slice_array<T>` als Argument angegeben werden. Das `valarray` übernimmt die Anzahl der Elemente aus dem `slice_array` und die Elemente des `valarray` sind Kopien der Komponenten des `slice_array`'s, etwa:

```
...
valarray<char> v("abcdef",6);    // v hat 6 Elemente mit Werten:
                                // 'a', 'b', 'c', 'd', 'e' und 'f'
valarray<char> w( v[slice(0,3,2)] ); // w hat 3 Elemente mit Werten:
                                // 'a', 'c' und 'e'
...
```

Ebenso kann einem `valarray<T>` ein (gleichlanges) `slice_array<T>` zugewiesen werden:

```
...
valarray<char> v("abcdef",6);    // v hat 6 Elemente mit Werten:
                                // 'a', 'b', 'c', 'd', 'e' und 'f'
valarray<char> w(3);              // w hat 3 Elemente mit Standardwerten
...
w = v[ slice(0,3,2) ];           // Werte von w sind jetzt: 'a', 'c' und 'e'
...
```

Matrizen und `valarray`'s

In mathematischen Anwendungen muss man häufig mit Matrizen rechnen.

Matrizen lassen sich einfach als `valarray`'s realisieren, indem man die einzelnen Zeilen (oder Spalten) der Matrix der Reihe nach im `valarray` ablegt.

Eine 4×3 -Matrix, deren Aussehen man sich meist wie folgt veranschaulicht (die Zahlentupel entsprechen den in der Mathematik üblichen Indizes):

00	01	02
10	11	12
20	21	22
30	31	32

könnte in einem `valarray<...> v` (spaltenweise) wie folgt abgelegt sein (entsprechende Matrix-Indizexpaare sind in den Quadraten — zugehörnde `valarray`-Indizes sind unter die Quadrate eingetragen):

1. Spalte				2. Spalte				3. Spalte			
00	10	20	30	01	11	21	31	02	12	22	32
0	1	2	3	4	5	6	7	8	9	10	11

Durch `slice`'s und `slice_array`'s hat man nun die Möglichkeit, auf Zeilen oder Spalten dieser Matrix zuzugreifen!

Beispiel:

1. Zugriff auf die dritte Zeile:

- (a) anschaulich als Matrix:

	00	01	02
	10	11	12
→ dritte Zeile	20	21	22
	30	31	32

- (b) anschaulich im `valarray<T>`:

1. Spalte				2. Spalte				3. Spalte			
00	10	20	30	01	11	21	31	02	12	22	32
0	1	2	3	4	5	6	7	8	9	10	11

- (c) als `slice_array`:

```
v[ slice(2,3,4) ];
```


(Startindex ist 2, 3 Elemente im Abstand jeweils 4!)

2. Zugriff auf zweite Spalte:

(a) anschaulich als Matrix:

zweite Zeile

00	01	02
10	11	12
20	21	22
30	31	32

(b) anschaulich im `valarray<T>`:

1. Spalte				2. Spalte				3. Spalte			
00	10	20	30	01	11	21	31	02	12	22	32
0	1	2	3	4	5	6	7	8	9	10	11

(c) als `slice_array`:

```
v[ slice(4,4,1) ];
```

(Startindex ist 4, 4 Elemente im Abstand jeweils 1!)

Mit dieser Technik kann man auch auf Teile einer Zeile bzw. Spalte einer Matrix zugreifen, indem man beispielsweise den Startindex (erstes `slice`-Argument) vergrößert oder die Länge des `slice` (zweites Argument) verkleinert.

Natürlich kann man mit `slice`'s auf einige andere (nicht alle) Komponentenmuster (etwa Diagonale, Nebendiagonalen, ...) einer Matrix zugreifen!

11.8.5 Verallgemeinerte Slices

Mit einem `slice` könnte man beispielsweise nicht auf die in folgendem Bild schraffierte Teilmatrix der Matrix aus dem letzten Abschnitt zugreifen:

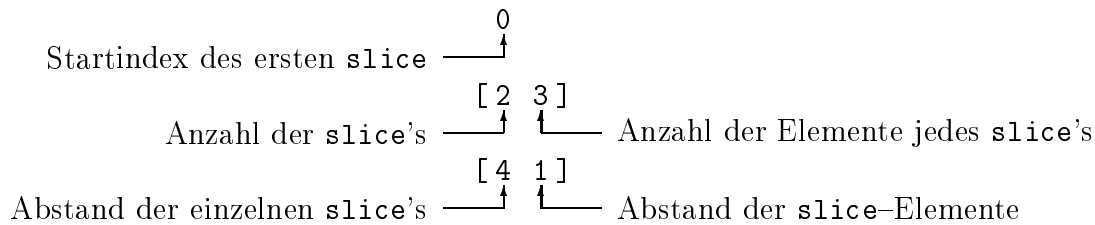
00	01	02
10	11	12
20	21	22
30	31	32

Eine Analyse dieses Teilbereichs der Matrix ergibt, dass dieser etwa wie folgt beschrieben werden kann:

die ersten drei Elemente der ersten Spalte **und** die ersten drei Elemente der zweiten Spalte.

Auf die ersten drei Elemente der ersten Spalte kann man über `slice(0,3,1)` und auf die ersten drei Elemente der zweiten Spalte über (den "gleichartigen") `slice(4,3,1)` zugreifen ("gleichartig" bedeutet: gleiche Länge und gleicher Abstand, unterschiedlicher Startindex)!

Wir haben es also mit zwei ("eindimensionalen") `slices` zu tun, wobei der Anfangsindex des zweiten `slice` um vier Positionen weiter nach hinten liegt, als die des ersten. Die Teilmatrix kann also durch folgende Zahlen beschrieben werden:



Ein solches "zweidimensionales `slice`" wird also durch drei Dinge beschrieben:

1. den Startindex (hier 0) des ersten (eindimensionalen) `slice`,
2. einem Vektor von Anzahlen (hier [2 3], die letzte Zahl 3 ist die Anzahl der Elemente eines `slice` und die Zahl 2 davor ist die Anzahl solcher `slice`'s),
3. einem Vektor von Abständen (hier [4 1], die letzte Zahl 1 ist der Abstand der Elemente eines `slice` und die Zahl davor ist der Abstand der Anfangspositionen der einzelnen `slice`'s).

Zur Realisierung eines solchen "zweidimensionalen Slice" gibt es in der Standardbibliothek den Hilfs-Datentyp `gslice` (*generalised slice*). Ein `gslice` wird konstruiert aus Startindex (vom Typ `size_t`), Anzahl-Vektor und Abstands-Vektor (beide vom Typ `valarray<size_t>`), d.h. bei der Konstruktion eines `gslice` muss man neben dem ganzzahligen (nicht negativen) Startindex zwei (gleichlange) `valarray<size_t>` angeben.

Für unser Beispiel (3×2 -Teilmatrix einer 4×3 -Matrix) könnte die Erzeugung des `gslice` wie folgt aussehen:

```
...
size_t start = 0;           // Startindex
size_t anz[] = { 2, 3 };    // gewoehnliches Feld der Anzahlen
size_t abst[] = { 4, 1 };   // gewoehnliches Feld der Abstaende
...
valarray<size_t> anzahlen(anz,2); // valarray<size_t> der Anzahlen
valarray<size_t> abstaende(abst,2); // valarray<size_t> der Abstaende
...
...gslice( start, anzahlen, abstaende)... // "zweidimensionales" slice
...
```

Ist nun wieder `v` unser `valarray<>` der Länge 12, welches wir als (spaltenweise) abgespeicherte 4×3 -Matrix auffassen, so kann mittels der Indizierung

```
v[ gslice(start, anzahlen, abstaende)]
```

auf die 3×2 -Teilmatrix oben links zugegriffen werden.

D.h. für den Indexoperator `[]` eines `valarray<T>` kann als Index ein `gslice` angegeben werden und man erhält hierüber wiederum den Zugriff auf die durch den `gslice` beschriebenen Komponenten des zugrundeliegenden `valarray<T>`.

Formal ist das Ergebnis der Indizierung eines `valarray<T>` mit einem `gslice` ein `gslice_array<T>`, falls das `valarray<T>` nicht konstant war, ansonsten wieder ein temporäres `valarray<T>` mit den durch das `gslice` spezifizierten Werten!

Ein `gslice_array<T>` hat die gleichen Eigenschaften wie ein `slice_array<T>`, d.h. es verfügt über die Elementfunktion `fill(T&)` zum Neubesetzen aller durch das `gslice` beschriebenen Elemente des zugrundeliegenden `valarray<T>` sowie die Zuweisungsoperatoren `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `&=`, `|=`, `<<=` und `>>=`, wobei auf der rechten Seite des Operators wiederum ein `valarray<T>` der richtigen Größe stehen muss.

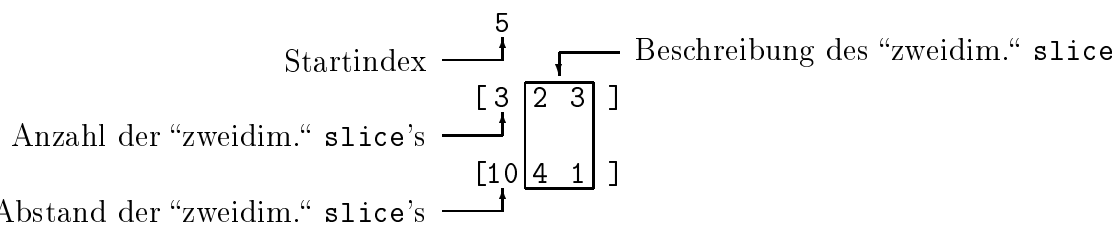
Ein `gslice_array<T>` kann wiederum bei der Erzeugung eines `valarray<T>` angegeben werden und einem `valarray<T>` kann ein `gslice_array<T>` zugewiesen werden.

Höherdimensionale `gslice`'s

Es gibt keine Beschränkung der Dimension von `gslice`'s — man muss nur bei der Erzeugung eines `gslice` längere Anzahl- und Abstands-Vektoren angeben, etwa:

```
...
size_t start  = 5;           // Startindex
size_t anz[]  = { 3, 2, 3};  // gewöhnliches Feld der Anzahlen
size_t abst[] = {10, 4, 1};  // gewöhnliches Feld der Abstaende
...
valarray<size_t> anzahlen(anz,2); // valarray<size_t> der Anzahlen
valarray<size_t> abstaende(abst,2); // valarray<size_t> der Abstaende
...
...gslice( start, anzahlen, abstaende)... // "dreidimensionales" slice
...
```

Die Bedeutung dieser Zahlen und bei Indizierung der dahinterliegenden Komponenten eines `valarray<T>` ist:



Entsprechend können auch "vierdimensionale", "fünfdimensionale", ... `slice`'s definiert werden!

11.8.6 Masken für valarray's

Ist `v` ein `valarray<T>` einer gewissen Länge und `mask` ein `valarray<bool>` (Komponententyp: `bool`!) der gleichen Länge, so kann man `mask` als "Index" zum `v` angeben:

```
...v[ mask ]...;
```

Ergebnis ist ein sogenannten `mask_array<T>`, mit dem man Zugriff auf genau diejenigen Elemente von `v` hat, zu denen die korrespondierende (gleicher Index) Komponente in `mask` den Wahrheitswert `true` hat. Die Elemente von `v`, zu denen die korrespondierenden Komponenten in `mask` den Wahrheitswert `false` haben, werden "ausgeblendet".

Ansonsten hat ein `mask_array<T>` genau die entsprechenden Eigenschaften wie ein `slice_array<T>` oder ein `gslice_array<T>` (Elementfunktion `fill(T&)`, Operatoren `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `&=`, `|=`, `<=<=` und `>=>=`, Erzeugung von und Zuweisung an `valarray<>`'s).

`mask_array`'s müssen dann verwendet werden, wenn die aus dem `valarray` zu "extrahierenden" Elemente nicht durch ein so wie für `slice`'s bzw. `gslice`'s notwendiges, einfaches Muster beschrieben werden können!

11.8.7 Indirekte valarray's

Beliebige Teilmuster mit Umnummerierung eines `valarray<T>` können mit einem `indirect_array<T>` "extrahiert" werden.

Ein `indirect_array<T>` ist das Ergebnis der Indizierung eines `valarray<T>` `v(...)` einer gewissen Länge `n` mit einem (höchstens gleichlangen)

```
valarray<size_t> indirekt(...)
```

(Komponententyp: `size_t`!), also das Ergebnis eines Ausdrucks der Form:

```
...v[ indirekt ]...;
```

wobei alle Komponenten von `indirekt` verschieden und aus dem Bereich von 0 bis `n-1` sein müssen.

Das resultierende `indirect_array<T>` hat ebensoviele Elemente wie `indirekt` und die `k`-te Komponente des `indirect_array<T>` korrespondiert zu dem Element von `v` mit Index `indirekt[k]`.

Beispiel:

```
...
size_t ind[] = { 3,2,1,0};           // gewoehnliches Feld
valarray<size_t> indirekt(ind,4);    // valarray<size_t>
...
valarray<...> v(...);               // irgendein valarray der Laenge mind. 4
...
... v [ indirekt ] ...;              // ersten vier Elemente von v in
                                     // umgekehrter Reihenfolge
...
```

Ein `indirect_array<T>` bietet die gleiche Funktionalität wie ein `slice_array<T>`, ein

`gslice_array<T>` oder ein `mask_array<T>` (Elementfunktion `fill(T&)` , Operatoren `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `&=`, `|=`, `<=<=` und `>>=` , Erzeugung von und Zuweisung an `valarray<T>`'s).

11.8.8 Numerische Algorithmen für Sequenzen

Die C++-Standardbibliothek stellt in Headerdatei `<numeric>` einige numerische Algorithmen für allgemeine, durch Iteratoren gegebene Sequenzen zur Verfügung, die in Ihrer Form den Algorithmen aus `<algorithm>` (vgl. Abschnitt 11.7) entsprechen.

Für `valarray`'s sind diese Algorithmen nicht geeignet, denn `valarray`'s haben keine Iteratoren. Desweiteren stehen für `valarray`'s spezielle effiziente Funktionen zur Verfügung.

Für die numerischen Algorithmen für Sequenzen werden wiederum folgende Bezeichnungen zugrundegelegt:

- **InIt:**
Bezeichnung für Input-Iterator-Typ.
- **OutIt:**
Bezeichnung für Output-Iterator-Typ.
- **BinOp**
binärer Funktionsobjekt-Typ (zwei Argumenttypen, ein Ergebnistyp).

Der numerische Algorithmus `accumulate`

```
template <class InIt, class T>
T accumulate( InIt anf, InIt ende, T wert);
```

addiert (mittels des Operators `+`) auf den angegebenen Wert `wert` der Reihe nach die Werte aller Elemente der Sequenz `[anf, ende)` und gibt den so berechneten Summenwert zurück.

Genauer: der zurückgegebene Wert `acc` wird wie folgt berechnet:

- `acc` wird mit `wert` vorbesetzt,
- für der Reihe nach jedes Element `elem` der Sequenz wird
`acc = acc + elem;`
berechnet.
- nach Bearbeitung der ganzen Sequenz wird `acc` zurückgegeben.

Der Elementtyp der Sequenz muss mit dem Typ `T` übereinstimmen und für diesen Typ muss der Additionsoperator `+` definiert sein.

Bei der zweiten Version:

```
template <class InIt, class T, class BinOp>
T accumulate( InIt anf, InIt ende, T wert, BinOp op);
```

übernimmt die binäre Operation `op` die Rolle der Addition mittels `+`. D.h. der zurückgegebene Wert `acc` wird wie folgt berechnet:

- `acc` wird mit `wert` vorbesetzt,
- für der Reihe nach jedes Element `elem` der Sequenz wird
`acc = op(acc , elem);`
 berechnet.
- nach Bearbeitung der ganzen Sequenz wird `acc` zurückgegeben.

Der numerische Algorithmus `inner_product`

```
template <class InIt, class InIt2, class T>
T inner_product( InIt anf, InIt ende, InIt2 anf2, T wert);
```

multipliziert jedes Element der ersten Sequenz [`anf`, `ende`) mit dem korrespondierenden Element der zweiten Sequenz, auf deren Anfang der Iterator `anf2` zeigt (es wird davon ausgegangen, dass die zweite Sequenz mindestens soviele Elemente hat wie die erste) und addiert das Produkt auf die mit dem angegebenen Wert `wert` vorbesetzte Summe. (Es wird also so eine Art *Skalarprodukt* berechnet.)

Genauer: der zurückgegebene Wert `acc` wird wie folgt berechnet:

- `acc` wird mit `wert` vorbesetzt,
- für der Reihe nach jedes Element `elem1` der ersten Sequenz sei `elem2` das korrespondierende Element der zweiten Sequenz und es wird
`acc = acc + elem1 * elem2;`
 berechnet.
- nach Bearbeitung der ganzen ersten Sequenz wird `acc` zurückgegeben.

Die Elementtypen der Sequenzen müssen mit dem Typ `T` übereinstimmen und für diesen Typ muss der Additions- und Multiplikationsoperator (`+` bzw. `*`) definiert sein.

Bei der zweiten Version:

```
template <class InIt, class InIt2, class T, class BinOp, class BinOp2>
T inner_product( InIt anf, InIt ende, InIt2 anf2, T wert, BinOp op,
                BinOp2 op2);
```

übernimmt die binäre Operation `op` die Rolle der Addition und die zweite binäre Operation `op2` die Rolle der Multiplikation. D.h. der zurückgegebene Wert `acc` wird wie folgt berechnet:

- `acc` wird mit `wert` vorbesetzt,

- für der Reihe nach jedes Element `elem1` der ersten Sequenz sei `elem2` das korrespondierende Element der zweiten Sequenz und es wird

```
acc = op( acc , op2( elem1, elem2) );
```

berechnet.

- nach Bearbeitung der ganzen ersten Sequenz wird `acc` zurückgegeben.

Der numerische Algorithmus `partial_sum`

```
template <class InIt, class OutIt>
OutIt partial_sum( InIt anf, InIt ende, OutIt out);
```

Die Elementtypen der Iteratoren müssen gleich sein und für diesen Typ muss der Additions- (+) definiert sein.

Berechnet im Ausgabe-Iterator eine Sequenz, welche aus den Partialsummen der Eingabesequenz [`anf`, `ende`) besteht. (Additionsoperator muss für den Elementtyp definiert sein!)

Genauer: Seien a_1, a_2, \dots, a_n die Elemente der ersten Sequenz [`anf`, `ende`), so werden in die Ausgabesequenz wie folgt berechnete Werte b_1, b_2, \dots, b_n geschrieben:

- $b_1 = a_1$
- für i von 2 bis n wird zu $b_i = b_{i-1} + a_i$ berechnet.

Mathematisch

$$b_i = \sum_{k=1}^i a_k$$

Bei der zweiten Version:

```
template <class InIt, class OutIt, class BinOp>
OutIt partial_sum( InIt anf, InIt ende, OutIt out, BinOp op);
```

übernimmt die binäre Operation `op` die Rolle der Addition, d.h. die Werte b_i der Ausgabesequenz werden wie folgt berechnet:

- $b_1 = a_1$
- für i von 2 bis n wird zu $b_i = \text{op}(b_{i-1}, a_i)$ berechnet.

Der Ausgabe-Iterator `out` muss in beiden Algorithmen auf einen hinreichend großen Container zeigen, dessen (erste) Elemente durch die Ausgabesequenz überschrieben werden, oder ein Inserter sein und darf mit der Anfangs-Iteratorposition `anf` übereinstimmen.

Ergebnis der Algorithmen ist der Ausgabe-Iterator auf das Ende der Ausgabesequenz.

Der numerische Algorithmus adjacent_difference

```
template <class InIt, class OutIt>
OutIt adjacent_difference( InIt anf, InIt ende, InIt2 anf2, T wert);
```

Die Elementtypen der Sequenzen müssen mit dem Typ T übereinstimmen und für diesen Typ muss der Subtraktionsoperator $(-)$ definiert sein.

Berechnet im Ausgabe-Iterator eine Sequenz, welche aus den Differenzen benachbarter Elemente der Eingabesequenz `[anf, ende)` besteht.

Genauer: Seien a_1, a_2, \dots, a_n die Elemente der ersten Sequenz `[anf, ende)`, so werden in die Ausgabesequenz wie folgt berechnete Werte b_1, b_2, \dots, b_n geschrieben:

- für i von 2 bis n wird $b_i = a_i + a_{i-1}$ gesetzt.

(über das erste Element b_1 ist im Standard nichts gesagt, üblicherweise dürfte es gleich a_1 sein!)

Bei der zweiten Version:

```
template <class InIt, class OutIt, class BinOp>
OutIt adjacent_difference( InIt anf, InIt ende, OutIt out, BinOp op);
```

übernimmt die binäre Operation `op` die Rolle der Subtraktion, d.h. die Werte b_i der Ausgabesequenz werden wie folgt berechnet:

- für i von 2 bis n wird $b_i = \text{op}(a_i, a_{i-1})$ gesetzt.

Der Ausgabe-Iterator `out` muss in beiden Algorithmen auf einen hinreichend großen Container zeigen, dessen (erste) Elemente durch die Ausgabesequenz überschrieben werden, oder ein Inserter sein und darf mit der Anfangs-Iteratorposition `anf` übereinstimmen.

Ergebnis der Algorithmen ist der Ausgabe-Iterator auf das Ende der Ausgabesequenz.

Teil IV

Anhang

Literaturverzeichnis

- [Boo 94] G. Booch, *Object-Oriented Analysis and Design*, Benjamin/Cummings Publishing Company, 1994
- [Han 00] RRZN Hannover, *Die Programmiersprache C++ für C-Programmierer*, 11-te Auflage, Regionales Rechenzentrum für Niedersachsen/ Universität Hannover, 2000
- [ISO 98] ISO/IEC JTC1/SC22 Sekretariat, CD 14882: *Programming Language C++*, Veröffentlichung der International-Standards-Organisation, 1998
- [Jos 94] N. Josuttis, *Objektorientiertes Programmieren in C++*, Addison-Wesley, 1994
- [Jos 96] N. Josuttis, *Die C++-Standardbibliothek*, Addison-Wesley, 1996
- [Ker 90] B. W. Kernighan und D. Ritchie, *Programmieren in C*, 2-te Auflage, Carl Hanser, 1990
- [Kuh 95] St. Kuhlins, *Manipulierte Ströme*, c't 1995, Heft 10, Seite 354–356
- [Mey 92] S. Meyers, *Effective C++*, Addison-Wesley 1992
- [Mey 96] S. Meyers, *More Effective C++*, Addison-Wesley 1996
- [Pri 98] P. Prinz, U. Kirch-Prinz, *Objektorientiertes Programmieren in ANSI-C++*, Prentice Hall, 1998
- [Str 94] B. Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994
- [Str 98] B. Stroustrup, *Die C++ Programmiersprache*, 3-te Auflage, Addison-Wesley, 1998

Index

- ! Negationsoperator, 135, **137**
 - Überladung, 166
 - für `valarray<>`, 502
- != Vergleichsoperator, 135, **138**
 - als Template, 345
 - für `bitset<>`, 444
 - für `complex<>`, 498
 - für Container, 365
 - für `deque<>`, 387
 - für Iteratoren, 348, 361
 - für `list<>`, 400
 - für `map<>`, 427
 - für `multimap<>`, 427
 - für `multiset<>`, 413
 - für `queue<>`, 430
 - für `set<>`, 413
 - für `stack<>`, 438
 - für `string`, 73, 331
 - Überladung, 166
 - für `valarray<>`, 503
 - für `vector<>`, 378
- () Funktionsaufruf, 135, **136**
 - Überladung, **159**, 166
- (type) Cast-Operator, 135, **137**
 - Überladung, 166
- * Stern-Operator
 - Multiplikation, 135, **138**
 - für `complex<>`, 498
 - Überladung, 166
 - für `valarray<>`, 502
 - Verweis, 135, **137**
 - für Iteratoren, 348
 - Überladung, 166
- *= Zuweisungsoperator, 135, **139**
 - für `complex<>`, 499
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
- Überladung, 166
 - für `valarray<>`, 503
- + Plus-Operator
 - Addition, 135, **138**
 - `complex<>`, 498
 - für Iteratoren, 349
 - Überladung, 166
 - für `valarray<>`, 502
 - Verkettung von `strings`, 73, 334, 335
 - Vorzeichen, 135, **137**
 - `complex<>`, 498
 - Überladung, 166
 - für `valarray<>`, 502
- ++ Inkrement-Operator
 - für Iteratoren, 348
 - Postfix, 135, **136**
 - Überladung, **163**, 166
 - Präfix, 135, **136**, **137**
 - Überladung, **163**, 166
 - Überladung, **163**
- += Zuweisungsoperator, 135, **139**
 - Anhängen an `string`, 74, 334
 - für `complex<>`, 499
 - für Iteratoren, 349
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - Überladung, 166
 - für `valarray<>`, 503
- , Kommaoperator, 135, **139**
 - für Klassen, 141
 - Überladung, 166
- Minus-Operator
 - Addition
 - für Iteratoren, 349
 - Subtraktion, 135, **138**
 - für `complex<>`, 498
 - Überladung, 166

- für `valarray<>`, 502
- Vorzeichen, 135, **137**
 - `complex<>`, 498
 - Überladung, 166
 - für `valarray<>`, 502
- Dekrement-Operator
 - für Iteratoren, 349
 - Postfix, 135, **136**
 - Überladung, **163**, 166
 - Präfix, 135, **136**, **137**
 - Überladung, **163**, 166
 - Überladung, **163**
- = Zuweisungsoperator, 135, **139**
 - für `complex<>`, 499
 - für Iteratoren, 349
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - Überladung, 166
 - für `valarray<>`, 503
- > Komponenten-Zugriff, 135, **136**
 - für Iteratoren, 348
 - Überladung, **161**, 166
- >* für Komponentenzeiger, 135, 137
 - Überladung, 166
- . Komponenten-Zugriff, 135, **136**
 - Überladung, 166
- .* für Komponentenzeiger, 135, 137
 - Überladung, 166
- / Divisionsoperator, 135, **138**
 - für `complex<>`, 498
 - Überladung, 166
 - für `valarray<>`, 502
- /= Zuweisungsoperator, 135, **139**
 - für `complex<>`, 499
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - Überladung, 166
 - für `valarray<>`, 503
- :: Bereichs-Operator, 135, **136**
 - Bereichsauflösung, 14
 - explizite Qualifikation, 243, 244
 - Überladung, 166
- < Vergleichsoperator, 135, **138**
 - für Container, 365
 - für `deque<>`, 387
 - für Iteratoren, 349
 - für `pair<>`, 346
 - für `list<>`, 400
 - für `map<>`, 427
 - für `multimap<>`, 427
 - für `multiset<>`, 413
 - für `queue<>`, 430
 - für `set<>`, 413
 - für `stack<>`, 438
 - für `string`, 73, 331
 - Überladung, 166
- << Shift-Operator, 135, **138**
 - Ausgabeoperator, 64, 275
 - für `bitset<>`, 445
 - für `complex<>`, 499
 - für `string`, 341
 - Überladung für Manipulator, 293
 - für `bitset<>`, 444
 - Links-Shift, 65
 - Überladung, 166
 - für `valarray<>`, 502
- <= Zuweisungsoperator, 135, **139**
 - für `bitset<>`, 443
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - Überladung, 166
 - für `valarray<>`, 503
- <= Vergleichsoperator, 135, **138**
 - als Template, 345
 - für Container, 365
 - für `deque<>`, 387
 - für Iteratoren, 349
 - für `list<>`, 400
 - für `map<>`, 427
 - für `multimap<>`, 427
 - für `multiset<>`, 413
 - für `queue<>`, 430
 - für `set<>`, 413
 - für `stack<>`, 438
 - für `string`, 73, 331
 - Überladung, 166

- für `valarray<>`, 503
- für `vector<>`, 378
- = Zuweisungsoperator, 135, **139**
 - für `complex<>`, 497
 - für Container, 366
 - für `deque<>`, 381
 - für Iteratoren, 349
 - für Klassen, 140
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für Iteratoren, 361
 - für `list<>`, 392
 - für `map<>`, 417
 - für `mask_array<>`, 512
 - für `multimap<>`, 417
 - für `multiset<>`, 404
 - für `priority_queue<>`, 436
 - für `queue<>`, 431
 - für `set<>`, 404
 - für `slice_array<>`, 507
 - für `stack<>`, 439
 - für `string`, 74, 326
 - Überladung, 152, **153**, 166
 - für `valarray<>`, 501
 - für `vector<>`, 369
 - virtuell, 218
- == Vergleichsoperator, 135, **138**
 - für `bitset<>`, 444
 - für `complex<>`, 498
 - für Container, 365
 - für `deque<>`, 387
 - für `pair<>`, 346
 - für Iteratoren, 348, 361
 - für `list<>`, 400
 - für `map<>`, 427
 - für `multimap<>`, 427
 - für `multiset<>`, 413
 - für `queue<>`, 430
 - für `set<>`, 413
 - für `stack<>`, 438
 - für `string`, 73, 330
 - Überladung, 166
 - für `valarray<>`, 503
 - für `vector<>`, 378
- > Vergleichsoperator, 135, **138**
 - als Template, 345
 - für Container, 365
- für `deque<>`, 387
- für Iteratoren, 349
- für `list<>`, 400
- für `map<>`, 427
- für `multimap<>`, 427
- für `multiset<>`, 413
- für `queue<>`, 430
- für `set<>`, 413
- für `stack<>`, 438
- für `string`, 73, 331
- Überladung, 166
- für `valarray<>`, 503
- für `vector<>`, 378
- >= Vergleichsoperator, 135, **138**
 - als Template, 345
 - für Container, 365
 - für `deque<>`, 387
 - für Iteratoren, 349
 - für `list<>`, 400
 - für `map<>`, 427
 - für `multimap<>`, 427
 - für `multiset<>`, 413
 - für `queue<>`, 430
 - für `set<>`, 413
 - für `stack<>`, 438
 - für `string`, 73, 331
 - Überladung, 166
 - für `valarray<>`, 503
 - für `vector<>`, 378
- >> Shift-Operator, 135, **138**
 - für `bitset<>`, 444
 - Eingabeoperator, 65, 68, 285
 - für `bitset<>`, 445
 - für `complex<>`, 499
 - für `string`, 341
 - Überladung für Manipulator, 294
 - Überladung, 166
 - für `valarray<>`, 502
- >= Zuweisungsoperator, 135, **139**
 - für `bitset<>`, 443
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - Überladung, 166
 - für `valarray<>`, 503
- ?: Bedingter Ausdruck, 135, **138**

- Überladung, 166
- [] Indexoperator, 135, **136**
 - für `bitset<>`, 446
 - für `deque<>`, 385
 - für Iteratoren, 349
 - für `string`, 325
 - Überladung, **157**, 166
 - für `valarray<>`, 501, 505, 511, 512
 - für `vector<>`, 375
- % Modulo-Operator, 135, **138**
 - Überladung, 166
 - für `valarray<>`, 502
- %= Zuweisungsoperator, 135, **139**
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - Überladung, 166
 - für `valarray<>`, 503
- & Kaufmann-Und
 - Adressoperator, 135, **137**
 - für Klassen, 141
 - Überladung, 166
 - für `bitset<>`, 445
 - Bit-Und, 135, **138**
 - Überladung, 166
 - für `valarray<>`, 502
- &= Zuweisungsoperator, 135, **139**
 - für `bitset<>`, 442
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - Überladung, 166
 - für `valarray<>`, 503
- && Logisches Und, 135, **138**
 - Überladung, 166
 - für `valarray<>`, 502
- ^ Exklusives Bit-Oder, 135, **138**
 - für `bitset<>`, 445
 - Überladung, 166
 - für `valarray<>`, 502
- ^= Zuweisungsoperator, 135, **139**
 - für `bitset<>`, 443
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - Überladung, 166
 - für `valarray<>`, 503
- | Bit-Oder, 135, **138**
 - für `bitset<>`, 445
 - Überladung, 166
 - für `valarray<>`, 502
- |= Zuweisungsoperator, 135, **139**
 - für `bitset<>`, 442
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - Überladung, 166
 - für `valarray<>`, 503
- || Logisches Oder, 135, **138**
 - Überladung, 166
 - für `valarray<>`, 502
- ~ Komplementoperator, 135, **137**
 - Überladung, 166
 - für `valarray<>`, 502
- Komplementoperator
 - für `bitset<>`, 444
- abgeleitete Klasse, 189
- Ableitung, *siehe* Vererbung
- abs()
 - `complex<>`, 498
 - mathematische Funktion, 495
 - `valarray<>`, 505
- abstrakte Basisklasse, 237
- Abstrakte Datentypen, 82
- accumulate(), **513**
- acos()
 - mathematische Funktion, 496
 - `valarray<>`, 505
- adjacent_difference(), **516**
- adjacent_find()
 - Algorithmus, 458, **466**
- Adresse
 - eines Konstruktors, 120
 - Funktion, 293
- advance()
 - für Iteratoren, 353
- <algorithm>
 - Headerdatei, 458
- Algorithmen, 456–495

- accumulate(), 513
- adjacent_difference(), 516
- adjacent_find(), 458, 466
- binary_search(), 460, 485
- copy(), 459, 470
- copy_backward(), 459, 471
- count(), 457, 458, 467
- count_if(), 458, 467
- equal(), 458, 467
- equal_range(), 460, 485
- fill(), 459, 475
- fill_n(), 459, 475
- find(), 457, 458, 464
- find_end(), 458, 469
- find_first_of(), 458, 465
- find_if(), 458, 465
- for_each(), 458, 462
- generate(), 459, 475
- generate_n(), 459, 476
- includes(), 460, 488
- inner_product(), 514
- inplace_merge(), 460, 486
- iter_swap(), 459, 473
- lexicographical_compare(), 461, 495
- lower_bound(), 460, 484
- make_heap(), 461, 492
- max(), 461, 494
- max_element(), 461, 494
- merge(), 460, 485
- min(), 461, 494
- min_element(), 461, 494
- mismatch(), 458, 468
- next_permutation(), 460, 486
- nth_element(), 460, 484
- numerische, 513–516
- partial_sort(), 460, 483
- partial_sort_copy(), 460, 483
- partial_sum(), 515
- partition(), 459, 481
- pop_heap(), 461, 493
- prev_permutation(), 460, 486
- push_heap(), 461, 492
- random_shuffle(), 459, 480
- remove(), 459, 476
- remove_copy(), 459, 477
- remove_copy_if(), 459, 477
- remove_if(), 459, 476
- replace(), 459, 474
- replace_copy(), 459, 474
- replace_copy_if(), 459, 474
- replace_if(), 459, 474
- reverse(), 459, 478
- reverse_copy(), 459, 479
- rotate(), 459, 479
- rotate_copy(), 459, 479
- search(), 458, 468
- search_n(), 458, 469
- set_difference(), 460, 490
- set_intersection(), 460, 489
- set_symmetric_difference(), 460, 491
- set_union(), 460, 488
- sort(), 460, 482
- sort_heap(), 461, 493
- stable_partition(), 459, 481
- stable_sort(), 460, 482
- swap(), 459, 473
- swap_ranges(), 459, 473
- transform(), 459, 472
- Übersicht, 458–461
- unique(), 459, 477
- unique_copy(), 459, 478
- upper_bound(), 460, 484
- Allokator, 367
- Anwenderschnittstelle, 197
- any()
 - für bitset<>, 444
- append()
 - für string, 333
- apply()
 - valarray<>, 504
- arg()
 - complex<>, 498
- asin()
 - mathematische Funktion, 496
 - valarray<>, 505
- assign()
 - für deque<>, 382
 - für list<>, 392
 - für string, 327
 - für vector<>, 369, 370
- assoziatives Feld, 414
- Assoziativität, 135, 142

- at()
 - für deque<>, 385
 - für string, 326
 - für vector<>, 375
- atan()
 - mathematische Funktion, 496
 - valarray<>, 505
- atan2()
 - mathematische Funktion, 496
 - für valarray<>, 505
- Ausgabe, 275–285
 - dezimal, 280
 - Formatierung, 277–285
 - ganzzahliger Werte
 - Basis, 280
 - führendes Zeichen, 282
 - Gleitkommazahlen, 281
 - Dezimalpunkt, 282
 - Festpunktschreibweise, 281
 - Voreinstellung, 282
 - wissenschaftlich, 281
 - hexadezimal, 281
 - oktal, 280
 - Pufferung, 276, 283
 - von bitset<>, 445
 - von strings, 341
 - Zahlwerte
 - Großbuchstaben, 283
 - Vorzeichen, 282
- Ausgabeoperator <<, **64**, 275
 - für bitset<>, 445
 - für complex<>, 499
 - für string, 341
 - Überladung für Manipulator, 293
- Ausgabestrom, 64, 69
- Ausnahme, 48
 - abfangen, 48
 - Ausnahmeschnittstelle, 53
 - auswerfen, 48
 - bad_alloc, 57, 312
 - bad_cast, 226
 - bad_exception, 54, 310
 - bad_typeid, 227, 313
 - Behandlung, 47–54
 - bei Strömen, 307
 - domain_error, 312
 - exception, 309
 - in der Standardbibliothek, 309–315
 - Übersicht, 315
 - in Destruktoren, 122
 - in Konstruktoren, 113
 - invalid_argument, 312, 441
 - ios_base::failure, 308, 314
 - length_error, 74, 75, 312, 322–325, 327, 333, 338
 - logic_error, 311
 - nicht abgefangene, 52
 - out_of_range, 312, 324, 326, 327, 329, 330, 332, 333, 335, 336, 375, 385, 441, 443, 444
 - overflow_error, 312
 - range_error, 312
 - runtime_error, 312
 - und Vererbung, 234
 - underflow_error, 312
 - unerwartete, 54
 - Unterscheidung von Ausnahmen, 51
 - weiterreichen, 50
- Ausrichtung, 279
- Auswertungsreihenfolge, 136
- back()
 - für Container, 366
 - für deque<>, 380
 - list<>, 389
 - für list<>, 395
 - für queue<>, 431
- Back-Insertter, 355
- back_inserter(), 356
- back_insert_iterator, 355
- bad()
 - ios, 288
- bad_alloc, 57, 312
- bad_cast, 226
- bad_exception, 54, 310
- bad_typeid, 227, 313
- base(), 355
- basic_filebuf<>, 271
- basic_fstream<>, 272
- basic_ifstream<>, 272
- basic_ios<>, 272
- basic_iostream<>, 272
- basic_istream<>, 272
- basic_istreamstream<>, 272

- basic_ofstream<>, 272
- basic_ostream<>, 272
- basic_ostringstream<>, 272
- basic_streambuf<>, 271
- basic_string<>, 317
- basic_stringbuf<>, 271
- basic_stringstream<>, 272
- Basisklasse, 189
 - abstrakte, 237
 - direkte, 190
 - indirekte, 190
 - mehrfache, 243
 - virtuelle, 247
- begin()
 - Container, 361
 - für deque<>, 384
 - für list<>, 394
 - für map<>, 420
 - für multimap<>, 420
 - für multiset<>, 407
 - für set<>, 407
 - für string, 319, 321
 - für vector<>, 373
- Bereichsauflösung, 14
- Bibliotheksfunktion
 - abs(), 495
 - complex<>, 498
 - für valarray<>, 505
 - acos(), 496
 - für valarray<>, 505
 - advance()
 - für Iteratoren, 353
 - any()
 - für bitset<>, 444
 - append()
 - für string, 333
 - apply()
 - valarray<>, 504
 - arg()
 - complex<>, 498
 - asin(), 496
 - für valarray<>, 505
 - assign()
 - für deque<>, 382
 - für list<>, 392
 - für string, 327
 - für vector<>, 369, 370
 - at()
 - für deque<>, 385
 - für string, 326
 - für vector<>, 375
 - atan(), 496
 - für valarray<>, 505
 - atan2(), 496
 - für valarray<>, 505
 - back()
 - für Container, 366
 - für deque<>, 380
 - für list<>, 389, 395
 - für queue<>, 431
 - bad()
 - ios, 288
 - base(), 355
 - begin()
 - für deque<>, 384
 - für list<>, 394
 - für map<>, 420
 - für multimap<>, 420
 - für multiset<>, 407
 - für set<>, 407
 - für string, 319, 321
 - für vector<>, 373
 - capacity()
 - für string, 322
 - für vector<>, 370
 - ceil(), 495
 - clear()
 - für Container, 366
 - für deque<>, 386
 - ios, 289, 306
 - für list<>, 396
 - für map<>, 421
 - für multimap<>, 421
 - für multiset<>, 408
 - für set<>, 408
 - für string, 335
 - für vector<>, 378
 - close()
 - für Streams, 303
 - compare()
 - für string, 329
 - conj()
 - complex<>, 497
 - copy()

- für string, 329
- cos(), 496
 - complex<>, 499
 - für valarray<>, 505
- cosh(), 496
 - complex<>, 499
 - für valarray<>, 505
- count()
 - für bitset<>, 444
 - für map<>, 424
 - für multimap<>, 424
 - für multiset<>, 411
 - für set<>, 411
- cshift()
 - valarray<>, 504
- c_str()
 - für string, 328
- data()
 - für string, 328
- distance()
 - für Iteratoren, 354
- empty()
 - für Container, 364
 - für deque<>, 383
 - für list<>, 393
 - für map<>, 418
 - für multimap<>, 418
 - für multiset<>, 405
 - für priority_queue<>, 435
 - für queue<>, 431
 - für set<>, 405
 - für stack<>, 439
 - für string, 322
 - für vector<>, 370
- end()
 - für deque<>, 384
 - für list<>, 394
 - für map<>, 420
 - für multimap<>, 420
 - für multiset<>, 407
 - für set<>, 407
 - für string, 319, 321
 - für vector<>, 373
- eof()
 - ios, 288
- equal_range()
 - für map<>, 425
 - für multimap<>, 425
 - für multiset<>, 411
 - für set<>, 411
- erase()
 - für Container, 367
 - für deque<>, 386
 - für list<>, 390, 396
 - für map<>, 421, 422
 - für multimap<>, 421, 422
 - für multiset<>, 408, 409
 - für set<>, 408, 409
 - für string, 335
 - für vector<>, 377
- exceptions()
 - ios, 308
- exp(), 496
 - complex<>, 499
 - für valarray<>, 505
- fabs(), 495
- fail()
 - ios, 288
- fill()
 - für gslice_array<>, 511
 - für indirect_array<>, 513
 - für mask_array<>, 512
 - ostream, 279
 - für slice_array<>, 506
- find()
 - für map<>, 424
 - für multimap<>, 424
 - für multiset<>, 411
 - für set<>, 411
 - für string, 338
- find_first_not_of()
 - für string, 340
- find_first_of()
 - für string, 339
- find_last_not_of()
 - für string, 340
- find_last_of()
 - für string, 340
- flags()
 - ostream, 283
- flip()
 - für bitset<>, 443, 444
 - vector<bool>, 379
- floor(), 495

- flush()
 - ostream, 276
- fmod(), 496
- free()
 - valarray<>, 500
- frexp(), 496
- front()
 - für Container, 366
 - für deque<>, 380
 - für list<>, 388, 395
 - für queue<>, 431
- get()
 - istream, 285
- getline(), 72
 - istream, 286
 - für string, 341
- good()
 - ios, 288
- ignore()
 - istream, 286
- imag()
 - complex<>, 497
- insert()
 - für deque<>, 386
 - für list<>, 396
 - für map<>, 421, 422
 - für multimap<>, 421, 422
 - für multiset<>, 408, 409
 - für set<>, 408, 409
 - für string, 332
 - für vector<>, 377
- is_open()
 - für Streams, 304
- iter_swap(), 354
- key_comp()
 - für map<>, 424
 - für multimap<>, 424
 - für multiset<>, 410
 - für set<>, 410
- ldexp(), 496
- length()
 - für string, 322
- log(), 496
 - complex<>, 499
 - für valarray<>, 505
- log10(), 496
 - complex<>, 499
 - für valarray<>, 505
- lower_bound()
 - für map<>, 425
 - für multimap<>, 425
 - für multiset<>, 411
 - für set<>, 411
- max()
 - valarray<>, 504
- max_size()
 - für Container, 365
 - für deque<>, 383
 - für list<>, 393
 - für map<>, 418
 - für multimap<>, 418
 - für multiset<>, 405
 - für set<>, 405
 - für string, 322
 - für vector<>, 370
- merge()
 - für list<>, 399
- min()
 - valarray<>, 504
- modf(), 496
- none()
 - für bitset<>, 444
- norm()
 - complex<>, 498
- open()
 - für Streams, 304
- peek()
 - istream, 286
- polar()
 - complex<>, 497
- pop()
 - für queue<>, 431
 - für stack<>, 439
- pop_back()
 - für deque<>, 380, 386
 - für list<>, 388, 396
 - für vector<>, 377
- pop_front()
 - für deque<>, 379, 380, 387
 - für list<>, 388, 396
- poph()
 - für priority_queue<>, 436
- pow(), 496
 - complex<>, 499

- für valarray<>, 505
- precision()
 - ostream, 278
- push()
 - für priority_queue<>, 435
 - für queue<>, 431
 - für stack<>, 439
- push_back()
 - für deque<>, 380, 386
 - für list<>, 388, 395
 - für string, 334
 - für vector<>, 376
- push_front()
 - für deque<>, 379, 380, 386
 - für list<>, 388, 396
- put()
 - ostream, 275
- putback()
 - istream, 286
- rbegin()
 - für deque<>, 384
 - für list<>, 394
 - für map<>, 420
 - für multimap<>, 420
 - für multiset<>, 407
 - für set<>, 407
 - für string, 320, 321
 - für vector<>, 373
- rdstate()
 - ios, 289
- read()
 - istream, 286
- real()
 - complex<>, 497
- remove()
 - für list<>, 398
- remove_if()
 - für list<>, 398
- rend()
 - für deque<>, 384
 - für list<>, 394
 - für map<>, 420
 - für multimap<>, 420
 - für multiset<>, 407
 - für set<>, 407
 - für string, 320, 321
 - für vector<>, 373
- replace()
 - für string, 336
- reserve()
 - für string, 323
 - für vector<>, 371
- reset()
 - für bitset<>, 443
- resize()
 - für deque<>, 383
 - für list<>, 393
 - für string, 322
 - valarray<>, 501
 - für vector<>, 371
- rfind()
 - für string, 339
- seekg()
 - istream, 306
- seekp()
 - ostream, 305
- set()
 - für bitset<>, 443
- setf()
 - ios_base, 279, 287
- set_new_handler(), 58
- setstate()
 - ios, 290
- set_terminate(), 311
- set_unexpected(), 54, 310
- shift()
 - valarray<>, 504
- sin(), 496
 - complex<>, 499
 - für valarray<>, 505
- sinh(), 496
 - complex<>, 499
 - für valarray<>, 505
- size()
 - für bitset<>, 442
 - für Container, 364
 - für deque<>, 383
 - für list<>, 393
 - für map<>, 418
 - für multimap<>, 418
 - für multiset<>, 405
 - für priority_queue<>, 435
 - für queue<>, 431
 - für set<>, 405

- für slice, 505
- für stack<>, 439
- für string, 322
- valarray<>, 500
- für vector<>, 370
- sort()
 - für list<>, 398, 399
- splice()
 - für list<>, 397, 398
- sqrt(), 496
 - complex<>, 499
 - für valarray<>, 505
- start()
 - für slice, 505
- str()
 - istringstream, 343
 - ostringstream, 343
 - stringstream, 343
- stride()
 - für slice, 505
- substr()
 - für string, 335
- sum()
 - valarray<>, 504
- swap()
 - für Container, 366
 - für deque<>, 386, 387
 - für list<>, 396, 400
 - für map<>, 421, 427
 - für multimap<>, 421, 427
 - für multiset<>, 408, 413
 - für set<>, 408, 413
 - für string, 342
 - für vector<>, 378
- sync()
 - istream, 286
- tan(), 496
 - complex<>, 499
 - für valarray<>, 505
- tanh(), 496
 - complex<>, 499
 - für valarray<>, 505
- tellg()
 - istream, 306
- tellp()
 - ostream, 305
- terminate(), 311
- test()
 - für bitset<>, 444
- tie()
 - istream, 307
- top()
 - für priority_queue<>, 435
 - für stack<>, 439
- to_string()
 - für bitset<>, 445
- to_ulong()
 - für bitset<>, 445
- uncaught_exception(), 124, 311
- unexpected(), 310
- unget()
 - istream, 286
- unique()
 - für list<>, 398
- unsetf()
 - ios_base, 279, 287
- upper_bound()
 - für map<>, 425
 - für multimap<>, 425
 - für multiset<>, 411
 - für set<>, 411
- value_comp()
 - für map<>, 424
 - für multimap<>, 424
 - für multiset<>, 410
 - für set<>, 410
- width()
 - istream, 286
 - ostream, 278
- write()
 - ostream, 275
- bidirectional_iterator_tag, 350
- Bidirektional-Iterator, 349
 - bidirectional_iterator_tag, 350
- binary_function
 - Basisklasse zu Funktionsobjekten, 450
- binary_search()
 - Algorithmus, 460, **485**
- bind1st()
 - Binder, 452
- bind2nd()
 - Binder, 452
- Binder, 452

- `bind1st()`, 452
- `bind2nd()`, 452
- `<bitset>`
 - Headerdatei, 440
- `bitset<>`, 440–448
 - `!=`, 444
 - `<<`, 444
 - `<=<`, 443
 - `==`, 444
 - `>>`, 444
 - `>>=`, 443
 - `[]`, 446
 - `&`, 445
 - `&=`, 442
 - `^`, 445
 - `^=`, 443
 - `|`, 445
 - `|=`, 442
 - `,`, 444
 - `any()`, 444
 - Ausgabe, 445
 - `count()`, 444
 - Eingabe, 445
 - `flip()`, 443, 444
 - Konstruktoren, 440
 - `none()`, 444
 - reference, 446–448
 - `reset()`, 443
 - `set()`, 443
 - `size()`, 442
 - `test()`, 444
 - `to_string()`, 445
 - `to_ulong()`, 445
 - Typumwnadlung, 445
- C++-String, 70, 318
- `capacity()`
 - für `string`, 322
 - für `vector<>`, 370
- `cast`
 - `const`, 12, 37
 - Crosscast*, 256
 - Downcast*, 256
 - `dynamic`, 13
 - `reinterpret`, 13
 - `static`, 12
 - Upcast*, 255
- `catch`, 48
 - `catch(...)`, 50
- C-Code, 9
- `ceil()`
 - mathematische Funktion, 495
- `cerr`, 64, 273
- `char`, 272
- `char_traits<>`, 317
- `char_traits<>`, 267–271
- `cin`, 64, 273
- `class`, 87
 - implizit `private`, 87
- `clear()`
 - für Container, 366
 - für `deque<>`, 386
 - `ios`, 289, 306
 - für `list<>`, 396
 - für `map<>`, 421
 - für `multimap<>`, 421
 - für `multiset<>`, 408
 - für `set<>`, 408
 - für `string`, 335
 - für `vector<>`, 378
- `clog`, 273
- `close()`
 - für Streams, 303
- `<cmath>`
 - Headerdatei, 495
- `compare()`
 - für `string`, 329
- `<complex>`
 - Headerdatei, 496
- `complex<>`, 496–499
 - `!=`, 498
 - `*`, 498
 - `*=`, 499
 - `+`, 498
 - `+=`, 499
 - `-`, 498
 - `-=`, 499
 - `/`, 498
 - `/=`, 499
 - `<<`, 499
 - `=`, 497
 - `==`, 498
 - `>>`, 499
 - `abs()`, 498

- arg(), 498
- Ausgabe, 499
- conj(), 497
- cos(), 499
- cosh(), 499
- Eingabe, 499
- exp(), 499
- imag(), 497
- konjugiert komplex, 497
- log(), 499
- log10(), 499
- norm(), 498
- polar(), 497
- Polarkoordinaten, 497, 498
- pow(), 499
- real(), 497
- sin(), 499
- sinh(), 499
- sqrt(), 499
- tan(), 499
- tanh(), 499
- conj()
 - complex<>, 497
- const
 - Funktionsergebnis, 34
 - Funktionsparameter, 30
 - Referenz, 30
 - Zeiger, 28
- const_cast<>(), 12, **37**, 135, 137
 - Überladung, 166
- const_pointer
 - Container, 360
- const_reference
 - Container, 360
- Container, 359–427
 - !=, 365
 - <, 365
 - <=, 365
 - ==, 365
 - >, 365
 - >=, 365
 - Adapter, 427–439
 - priority_queue<>, 432–436
 - queue<>, 427–432
 - stack<>, 436–439
 - Allokator, 367
 - back(), 366
 - begin(), 361
 - clear(), 366
 - const_iterator, 360
 - const_pointer, 360
 - const_reference, 360
 - const_reverse_iterator, 360
 - deque<>, 379–387
 - Destruktor, 364
 - difference_type, 360
 - empty(), 364
 - end(), 361
 - erase(), 367
 - front(), 366
 - Gemeinsamkeiten, 360–367
 - Hilfstypen, 360
 - iterator, 360
 - Iteratoren, 361
 - Konstruktor, 364
 - lexikalischer Vergleich, 365
 - list<>, 387–400
 - map<>, 413–427
 - max_size(), 365
 - multimap<>, 413–427
 - multiset<>, 400–413
 - pointer, 360
 - rbegin(), 361
 - reference, 360
 - rend(), 361
 - reverse_iterator, 360
 - set<>, 400–413
 - size(), 364
 - size_type, 360
 - swap(), 366
 - value_type, 360
 - vector<>, 368–378
- Containerklasse, *siehe* Container
- container_type
 - priorityqueue<>, 434
 - queue<>, 430
 - stack<>, 438
- copy()
 - Algorithmus, 459, **470**
 - für string, 329
- copy_backward()
 - Algorithmus, 459, **471**
- Copy-Konstruktor, 115
 - selbst definieren, 117

- `cos()`
 - `complex<>`, 499
 - mathematische Funktion, 496
 - `valarray<>`, 505
- `cosh()`
 - `complex<>`, 499
 - mathematische Funktion, 496
 - `valarray<>`, 505
- `count()`
 - Algorithmus, 457, 458, **467**
 - für `bitset<>`, 444
 - für `map<>`, 424
 - für `multimap<>`, 424
 - für `multiset<>`, 411
 - für `set<>`, 411
- `count_if()`
 - Algorithmus, 458, **467**
- `cout`, 64, 273
- Crosscast*, 256
- `cshift()`
 - `valarray<>`, 504
- `<cstdlib>`
 - Headerdatei, 55
- `c_str()`
 - für `string`, 328
- C-String, 70, 318
- `<cstring>`
 - Headerdatei, 318
- `data()`
 - für `string`, 328
- Datei
 - Behandlung, 69, 301–307
 - Öffnen, 69, 301, 303
 - Positionierung, 305–307
 - Schließen, 70, 301, 303
- Dateibehandlung, 304
- Datenkapselung, 80
- `dec` Manipulator
 - `istream`, 287
 - `ostream`, 280
- Deklaration
 - von Variablen, 13
- `delete`, 56, 135, 137
 - Überladung, 166
- `delete[]`, 56, 135, 137
 - Überladung, 166
- `<deque>`
 - Headerdatei, 379
- `deque<>`, 379–387
 - `!=`, 387
 - `<`, 387
 - `<=`, 387
 - `=`, 381
 - `==`, 387
 - `>`, 387
 - `>=`, 387
 - `[]`, 385
 - `assign()`, 382
 - `at()`, 385
 - `back()`, 380
 - `begin()`, 384
 - `clear()`, 386
 - Destruktoren, 381
 - `empty()`, 383
 - `end()`, 384
 - `erase()`, 386
 - `front()`, 380
 - Größe, 382
 - `insert()`, 386
 - Iteratoren, 383
 - Konstruktoren, 381
 - `max_size()`, 383
 - `pop_back()`, 380, 386
 - `pop_front()`, 379, 380, 387
 - `push_back()`, 380, 386
 - `push_front()`, 379, 380, 386
 - `rbegin()`, 384
 - `rend()`, 384
 - `resize()`, 383
 - `size()`, 383
 - `swap()`, 386, 387
 - Typen, 380
 - Vergleiche, 387
 - wesentliche Funktionalität, 379
- Destruktor, 94, 95, 120–124
 - Aufruf, 96
 - für Container, 364
 - für `deque<>`, 381
 - für `list<>`, 391
 - für `map<>`, 416
 - für `multimap<>`, 416
 - für `multiset<>`, 403
 - für `priority_queue<>`, 436

- für `queue<>`, 432
- selbst schützen, 122
- für `set<>`, 403
- für `stack<>`, 439
- Standard, 95, 122
- für `string`, 325
- und Ausnahmen, 121, 122
- und Ressourcenmanagement, 124
- und Vererbung, **233**
- für `vector<>`, 368
- virtuell, 122, 216
- dezimale
 - Ausgabe, 280
 - Eingabe, 287
- `difference_type`
 - Container, 360
- `distance()`
 - für Iteratoren, 354
- `divides<>`
 - Funktionsobjekt, 450
- `domain_error`, 312
- Downcast*, 256
- `dynamic_cast<>()`, 13, 135, 137, **224**, 255
 - Überladung, 166
- dynamische Komponente, *siehe* Komponente dynamische
- Ein- Ausgabe, 267–308
 - Hintergrund, 270
- einfache Vererbung, 189
- Einfügen
 - in `strings`, 331
- Eingabe, 285–287
 - dezimale, 287
 - Formatierung, 286–287
 - ganzzahliger Werte
 - Basis, 287
 - hexadezimale, 287
 - oktale, 287
 - von `bitset<>`, 445
 - von `strings`, 341
- Eingabeoperator `>>`
 - für `bitset<>`, 445
- Eingabeoperator `>>`, **65**, 68, 285
 - `complex<>`, 499
 - für `string`, 341
 - Überladung für Manipulator, 294
- Eingabestrom, 64, 69
- Elementfunktion, *siehe* Member Funktion
- Elementfunktionsadapter, 453
- `empty()`
 - für Container, 364
 - für `deque<>`, 383
 - für `list<>`, 393
 - für `map<>`, 418
 - für `multimap<>`, 418
 - für `multiset<>`, 405
 - für `priority_queue<>`, 435
 - für `queue<>`, 431
 - für `set<>`, 405
 - für `stack<>`, 439
 - für `string`, 322
 - für `vector<>`, 370
- `end()`
 - Container, 361
 - für `deque<>`, 384
 - für `list<>`, 394
 - für `map<>`, 420
 - für `multimap<>`, 420
 - für `multiset<>`, 407
 - für `set<>`, 407
 - für `string`, 319, 321
 - für `vector<>`, 373
- `endl` Manipulator
 - `ostream`, 67, 276
- `ends` Manipulator
 - `ostream`, 67, 277
- `eof()`
 - `ios`, 288
- EOF*-Istream-Iterator, 358
- `equal()`
 - Algorithmus, 458, **467**
- `equal_range()`
 - Algorithmus, 460, **485**
 - für `map<>`, 425
 - für `multimap<>`, 425
 - für `multiset<>`, 411
 - für `set<>`, 411
- `equal_to<>`
 - Prädikat, 451
- `erase()`
 - für Container, 367

- für deque<>, 386
- list<>, 390
- für list<>, 396
- für map<>, 421, 422
- für multimap<>, 421, 422
- für multiset<>, 408, 409
- für set<>, 408, 409
- für string, 335
- für vector<>, 377
- <exception>, 124
 - Headerdatei, 53, 54
- exception, *siehe* Ausnahme
 - Basis-Fehlerklasse, 309
 - Headerdatei, 309
- exceptions()
 - ios, 308
- exp()
 - complex<>, 499
 - mathematische Funktion, 496
 - valarray<>, 505
- explizite Qualifikation, 243, 244, 275
- export, 188
- Fabrik*, 264
- fabs()
 - mathematische Funktion, 495
- Factory*, 264
- fail()
 - ios, 288
- Fehler
 - in Strömen, 68, 287–290
- Fehlermeldung
 - what(), 309
- Fehlerobjekt, *siehe* Ausnahme
- Feld
 - assoziatives, 414
- Feldbreite, 278, 286
- Feldweite, *siehe* Feldbreite
- filebuf, 272
- File-Stream, 272
- fill()
 - Algorithmus, 459, **475**
 - für gslice_array<>, 511
 - für indirect_array<>, 513
 - für mask_array<>, 512
 - ostream, 279
 - für slice_array<>, 506
- fill_n()
 - Algorithmus, 459, **475**
- find()
 - Algorithmus, 457, 458, **464**
 - für map<>, 424
 - für multimap<>, 424
 - für multiset<>, 411
 - für set<>, 411
 - für string, 338
- find_end()
 - Algorithmus, 458, **469**
- find_first_not_of()
 - für string, 340
- find_first_of()
 - Algorithmus, 458, **465**
 - für string, 339
- find_if()
 - Algorithmus, 458, **465**
- find_last_not_of()
 - für string, 340
- find_last_of()
 - für string, 340
- fixed Manipulator
 - ostream, 281
- flags()
 - ostream, 283
- flip()
 - für bitset<>, 443, 444
 - vector<bool>, 379
- floor()
 - mathematische Funktion, 495
- flush()
 - ostream, 276
- flush Manipulator
 - ostream, 67, 276
- fmod()
 - mathematische Funktion, 496
- fmtflags, *siehe* ios_base::fmtflags
- for_each()
 - Algorithmus, 458, **462**
- Forward-Iterator, 349
 - forward_iterator_tag, 350
- forward_iterator_tag, 350
- free()
 - valarray<>, 500
- Freispeicherverwaltung, 55–61
- frexp()

- mathematische Funktion, 496
- friend, 133, 150
- front()
 - für Container, 366
 - für deque<>, 380
 - list<>, 388
 - für list<>, 395
 - für queue<>, 431
- Front-Insertter, 356
- front_inserter(), 356
- front_insert_iterator, 356
- fstream, 272, 301
 - close(), 303
 - is_open(), 304
 - open(), 304
- Füllzeichen, 279
- <functional>
 - Headerdatei, 450
- Funktion
 - Adapter, 453–455
 - für Elementfunktionen, 453
 - für Funktionszeiger, 453
 - mem_fun(), 453
 - mem_fun_ref(), 453
 - ptr_fun(), 453
 - befreundete, 132
 - Ergebnis
 - const, 34
 - Objekt, 448–456
 - divides<>, 450
 - minus<>, 450
 - modulus<>, 450
 - multiplies<>, 450
 - negate<>, 450
 - plus<>, 450
 - Objekttyp, 448
 - Parameter
 - const, 30
 - rein virtuelle, 237
 - Überladung, 43
 - virtuelle, 208, 212–220, 255
 - Aufruf, 215
 - Defaultargumente, 216
 - Ergebnistyp, 209
- generate()
 - Algorithmus, 459, **475**
- generate_n()
 - Algorithmus, 459, **476**
- generische Programmierung, 171
- get()
 - istream, 285
- getline(), 72
 - istream, 286
 - für string, 341
- good()
 - ios, 288
- greater<>
 - Prädikat, 451
- greater_equal<>
 - Prädikat, 451
- gslice, 509–511
 - gslice_array<>
 - *, 511
 - +=, 511
 - =, 511
 - /=, 511
 - <<=, 511
 - =, 511
 - >>=, 511
 - %=, 511
 - &=, 511
 - ^=, 511
 - |=, 511
 - Operationen, 511
- Headerdatei
 - <algorithm>, 458
 - <bitset>, 440
 - <cmath>, 495
 - <complex>, 496
 - <cstdlib>, 55
 - <cstring>, 318
 - <deque>, 379
 - <exception>, 53, 54, 124, **309**
 - <functional>, 450
 - <iomanip>, 299
 - <iostream>, 63, 274, 314
 - <iterator>, 347
 - list<>, 388
 - map<><map>, 413
 - <new>, 57, 312
 - <numeric>, 513
 - <queue>, 427, 432

- `<set>`, 400
 - `<sstream>`, 274, 342
 - `<stack>`, 436
 - `<stdexcept>`, 226, 227, **311**
 - `<string>`, 70, 317
 - `<stringstream>`, 342
 - `<typeinfo>`, 226, 313
 - `<valarray>`, 500
 - `<vector>`, 368
- Heap, 432, 461
- hex Manipulator
 - `istream`, 287
 - `ostream`, 281
- hexadezimal
 - Ausgabe, 281
- hexadezimale
 - Eingabe, 287
- Hilfsklasse, 105
- `ifstream`, 69, 272, 301
 - `close()`, 303
 - `is_open()`, 304
 - `open()`, 304
- `ignore()`
 - `istream`, 286
- `imag()`
 - `complex<>`, 497
- Implementierungsdetail, 86
- `includes()`
 - Algorithmus, 460, **488**
- Information-Hiding, 80, 82, 83, 86
- Inheritance, *siehe* Vererbung
- Initialisierungsliste, 111, 128
- `inline`, 40
 - explizit, 89
 - implizit, 89
- `inner_product()`, **514**
- `inplace_merge()`
 - Algorithmus, 460, **486**
- Input-Iterator, 348
 - `input_iterator_tag`, 350
- `input_iterator_tag`, 350
- `insert()`
 - für `deque<>`, 386
 - für `list<>`, 396
 - für `map<>`, 421, 422
 - für `multimap<>`, 421, 422
 - für `multiset<>`, 408, 409
 - für `set<>`, 408, 409
 - für `string`, 332
 - für `vector<>`, 377
- Insert-Iterator, 355
 - Operatoren für, 355
- Insertter, 356
- `insertter()`, 356
- `insert_iterator`, 356
- internal Manipulator
 - `ostream`, 280
- `invalid_argument`, 312
- `<iomanip>`
 - Headerdatei, 299
- `ios`, 272
 - `bad()`, 288
 - `clear()`, 306
 - `clear()`, 289
 - `eof()`, 288
 - `exceptions()`, 308
 - `fail()`, 288
 - `good()`, 288
 - `rdstate()`, 289
 - `setstate()`, 290
- `ios_base`, 271
 - `ios_base::adjustfield`, 280, 285
 - `ios_base::app`, 302
 - `ios_base::ate`, 302
 - `ios_base::badbit`, 288, 303, 308
 - `ios_base::basefield`, 280, 285, 287
 - `ios_base::beg`, 305
 - `ios_base::binary`, 302
 - `ios_base::cur`, 305
 - `ios_base::dec`, 280, 287
 - `ios_base::end`, 305
 - `ios_base::eofbit`, 288, 308
 - `ios_base::failbit`, 288, 303, 308
 - `ios_base::failure`, 308, 314
 - `ios_base::fixed`, 281
 - `ios_base::floatfield`, 281, 285
 - `ios_base::fmtflags`, 279, 283
 - `ios_base::goodbit`, 288
 - `ios_base::hex`, 287
 - `ios_base::in`, 302, 342
 - `ios_base::internal`, 280
 - `ios_base::iostate`, 288
 - `ios_base::left`, 280

- `ios_base::hex`, 281
- `ios_base::oct`, 280, 287
- `ios_base::out`, 302, 342
- `ios_base::right`, 280
- `ios_base::scientific`, 281
- `ios_base::seekdir`, 305
- `ios_base::showbase`, 282
- `ios_base::showpoint`, 282
- `ios_base::showpos`, 282
- `ios_base::skipws`, 287
- `ios_base::trunc`, 302
- `ios_base::unitbuf`, 283
- `ios_base::uppercase`, 283
- `setf()`, 279, 287
- `unsetf()`, 279, 287
- `<iostream>`
 - Headerdatei, 63, 274, 314
- `iostream`, 272
- `is_open()`
 - für Streams, 304
- Ist-Ein-Beziehung*, 193
- `istream`, 64, 272
 - `dec` Manipulator, 287
 - `get()`, 285
 - `getline()`, 286
 - `hex` Manipulator, 287
 - `ignore()`, 286
 - Iterator, 358
 - `noskipws` Manipulator, 287
 - `oct` Manipulator, 287
 - `peek()`, 286
 - `putback()`, 286
 - `read()`, 286
 - `seekg()`, 306
 - `setw` Manipulator, 286
 - `skipws` Manipulator, 287
 - `sync()`, 286
 - `tellp()`, 306
 - `tie()`, 307
 - `unget()`, 286
 - `width()`, 286
 - `ws` Manipulator, 287
- `istringstream`, 272, 342
 - `str()`, 343
- `istrstream`, 342
- Iterator, 347–359
 - `!=`, 348, 361
 - `*`, 348
 - `+`, 349
 - `++`, 348
 - `+=`, 349
 - `-`, 349
 - `-`, 349
 - `-=`, 349
 - `->`, 348
 - `<`, 349
 - `<=`, 349
 - `=`, 349, 361
 - `==`, 348, 361
 - `>`, 349
 - `>=`, 349
 - `[]`, 349
 - Adapter, 354–356
 - `advance()`, 353
 - Back-Insertter, 355
 - Bidirektional, 349
 - `const_iterator`
 - Container, 360
 - `const_reverse_iterator`
 - Container, 360
 - Container, 361
 - für `deque<>`, 383
 - `distance()`, 354
 - Forward, 349
 - Front-Insertter, 356
 - für Streams, 357–359
 - Input, 348
 - Insertter, 355
 - Insertter, 356
 - `iterator`
 - Container, 360
 - `iter_swap()`, 354
 - Kategorie, 348–350, 352
 - für `list<>`, 393
 - für `map<>`, 418
 - für `multimap<>`, 418
 - für `multiset<>`, 405
 - Operatoren für, 348–350
 - Output, 348
 - Random-Access, 349
 - Reverse, 354
 - `reverse_iterator`
 - Iterator, 360
 - Sequenz, 347

- für `set<>`, 405
- für `string`, 319, 321
- Traits, 350–352
- für `vector<>`, 372
- Zeiger als, 351
- `<iterator>`
 - Headerdatei, 347
- `iterator_traits<>`, 350
- `iter_swap()`, 354
 - Algorithmus, 459, **473**
- `key_comp()`
 - für `map<>`, 424
 - für `multimap<>`, 424
 - für `multiset<>`, 410
 - für `set<>`, 410
- `key_compare`
 - `map<>`, 416
 - `multimap<>`, 416
 - `multiset<>`, 403
 - `set<>`, 403
- `key_type`
 - `map<>`, 416
 - `multimap<>`, 416
 - `multiset<>`, 403
 - `set<>`, 403
- Klasse, 86
 - als Softwarebaustein, 91
 - `basic_filebuf<>`, 271
 - `basic_fstream<>`, 272
 - `basic_ifstream<>`, 272
 - `basic_ios<>`, 272
 - `basic_iostream<>`, 272
 - `basic_istream<>`, 272
 - `basic_istreamstream<>`, 272
 - `basic_ofstream<>`, 272
 - `basic_ostream<>`, 272
 - `basic_ostreamstream<>`, 272
 - `basic_streambuf<>`, 271
 - `basic_string<>`, 317
 - `basic_stringbuf<>`, 271
 - `basic_stringstream<>`, 272
 - befreundete, 132
 - Definition, 86
 - Deklaration, 86
 - eingebettete Hilfsklasse, 105
 - `filebuf`, 272
 - `fstream`, 272, 301
 - `ifstream`, 272, 301
 - Implementierungsmöglichkeiten, 89
 - `ios`, 272
 - `ios_base`, 271
 - `iostream`, 272
 - `istream`, 272
 - `istreamstream`, 272, 342
 - `istrstream`, 342
 - Klassenhierarchie, 190, 255
 - Klassenkonstante, 128
 - Klassenrumpf, 86
 - Komponente
 - statische, 125
 - `ifstream`, 301
 - `ofstream`, 272
 - `ostream`, 272
 - `ostreamstream`, 272, 342
 - `ostrstream`, 342
 - polymorphe, 224, 237
 - `streambuf`, 272
 - `string`, 317
 - `stringbuf`, 272
 - `stringstream`, 272, 342
 - `strstream`, 342
 - Verwendung von, **103**
 - `wfilebuf`, 273
 - `wfstream`, 273
 - `wifstream`, 273
 - `wios`, 273
 - `wiostream`, 273
 - `wistream`, 273
 - `wistreamstream`, 273
 - `wofstream`, 273
 - `wostream`, 273
 - `wostreamstream`, 273
 - `wstreambuf`, 273
 - `wstring`, 317
 - `wstringbuf`, 273
 - `wstringstream`, 273
- Kommentar, 10
- komplexe Zahlen, 496–499
 - arithmetische Operationen, 498
 - Ein-/Ausgabe, 499
 - Imaginärteil, 497
 - mathematische Funktionen, 499
 - Realteil, 497

- Vergleichsoperatoren, 498
- Zuweisung, 496
- Komponente
 - dynamische, 95, 115, 233, 317
 - Komponentenzeiger, **129**
 - konstante, 128
 - Referenz, 128
- Konstante
 - als Komponente, 128
- konstantes Datenobjekt, 26–38
- Konstruktor, 83, **88**, 95, 106–120
 - Adresse von, 120
 - Aufruf, 83
 - für `bitset<>`, 440
 - für Container, 364
 - Copy-Konstruktor, 108, 115, 232
 - Defaultargumente, 110
 - für `deque<>`, 381
 - Initialisierungsliste, 111, 128, 232
 - für `list<>`, 391
 - für `map<>`, 416
 - für `multimap<>`, 416
 - für `multiset<>`, 403
 - parameterloser, 107, 110
 - für `queue<>`, 430, 431, 435
 - selbst schützen, 113
 - selbstgeschriebener, 109
 - für `set<>`, 403
 - für `stack<>`, 438
 - Standard, 88, **107**
 - für `string`, 323
 - Typumwandlung, 118, 256
 - und Ausnahmen, 113
 - und Ressourcenmanagement, 124
 - und Vererbung, **231**
 - für `valarray<>`, 500
 - für `vector<>`, 368
 - verbieten, 116
- Konversionsoperator, 166, 256
 - in der Standardbibliothek, 168
- Laufzeittypinformation, 224
- `ldexp()`
 - mathematische Funktion, 496
- left Manipulator
 - `ostream`, 280
- `length()`
 - für `string`, 322
- `length_error`, 74, 75, 312, 322–325, 327, 333, 338
- `less<>`
 - Prädikat, 451
- `less_equal<>`
 - Prädikat, 451
- `lexicographical_compare()`
 - Algorithmus, 461, **495**
- lexikalischer Vergleich, 365
- `<list>`
 - Headerdatei, 388
- `list<>`, 387–400
 - `!=`, 400
 - `<`, 400
 - `<=`, 400
 - `=`, 392
 - `==`, 400
 - `>`, 400
 - `>=`, 400
 - `assign()`, 392
 - `back()`, 389, 395
 - `begin()`, 394
 - `clear()`, 396
 - Destruktoren, 391
 - `empty()`, 393
 - `end()`, 394
 - `erase()`, 390, 396
 - `front()`, 388, 395
 - Größe, 392
 - `insert()`, 396
 - Iteratoren, 393
 - Konstrukturen, 391
 - `max_size()`, 393
 - `merge()`, 399
 - `pop_back()`, 388, 396
 - `pop_front()`, 388, 396
 - `push_back()`, 388, 395
 - `push_front()`, 388, 396
 - `rbegin()`, 394
 - `remove()`, 398
 - `remove_if()`, 398
 - `rend()`, 394
 - `resize()`, 393
 - `size()`, 393
 - `sort()`, 398, 399
 - `splice()`, 397, 398

- `swap()`, 396, 400
- Typen, 391
- `unique()`, 398
- Vergleiche, 399, 400
- `log()`
 - `complex<>`, 499
 - mathematische Funktion, 496
 - `valarray<>`, 505
- `log10()`
 - `complex<>`, 499
 - mathematische Funktion, 496
 - `valarray<>`, 505
- `logical_and<>`
 - Prädikat, 451
- `logical_not<>`
 - Prädikat, 451
- `logical_or<>`
 - Prädikat, 451
- `logic_error`, 311
- `lower_bound()`
 - Algorithmus, 460, **484**
 - für `map<>`, 425
 - für `multimap<>`, 425
 - für `multiset<>`, 411
 - für `set<>`, 411
- `make_heap()`
 - Algorithmus, 461, **492**
- `makepair()` Erzeugung eines `pair<>`, 347
- Manipulator, 67, 293–301
 - `dec`
 - `istream`, 287
 - `ostream`, 280
 - `endl`, 67, 276
 - `ends`, 67, 277
 - `fixed`
 - `ostream`, 281
 - `flush`, 67, 276
 - `hex`
 - `istream`, 287
 - `ostream`, 281
 - `internal`
 - `ostream`, 280
 - `left`
 - `ostream`, 280
 - mit Argument
 - selbst definiert, 294
 - mit mehreren Argumenten
 - selbst definiert, 300
- `noshowbase`
 - `ostream`, 282
- `noshowpoint`
 - `ostream`, 282
- `noshowpos`
 - `ostream`, 283
- `noskipws`
 - `istream`, 287
- `nouppercase`
 - `ostream`, 283
- `oct`
 - `istream`, 287
 - `ostream`, 280
- ohne Argument
 - selbst definiert, 293
- `resetiosflags`
 - `ostream`, 285
- `right`
 - `ostream`, 280
- `scientific`
 - `ostream`, 281
- `setbase()`
 - `ostream`, 281
- `setfill()`
 - `ostream`, 279
- `setiosflags`
 - `ostream`, 285
- `setprecision()`
 - `ostream`, 279
- `setw()`
 - `istream`, 286
 - `ostream`, 278
- `showbase`
 - `ostream`, 282
- `showpoint`
 - `ostream`, 282
- `showpos`
 - `ostream`, 283
- `skipws`
 - `istream`, 287
- `uppercase`
 - `ostream`, 283
- `ws`, 68
 - `istream`, 287

- <map>
 - Headerdatei, 413
- map<>, 413–427
 - !=, 427
 - <, 427
 - <=, 427
 - =, 417
 - ==, 427
 - >, 427
 - >=, 427
 - begin(), 420
 - clear(), 421
 - count(), 424
 - Destruktoren, 416
 - empty(), 418
 - end(), 420
 - equal_range(), 425
 - erase(), 421, 422
 - find(), 424
 - Größe, 417
 - insert(), 421, 422
 - Iteratoren, 418
 - key_comp(), 424
 - key_compare, 416
 - key_type, 416
 - Konstruktor, 416
 - lower_bound(), 425
 - mapped_type, 416
 - max_size(), 418
 - rbegin(), 420
 - rend(), 420
 - size(), 418
 - swap(), 421, 427
 - upper_bound(), 425
 - value_comp(), 424
 - value_compare, 416
 - Vergleiche, 426
- mapped_type
 - map<>, 416
 - multimap<>, 416
- mathematischer Vektor, 499–513
- max()
 - Algorithmus, 461, **494**
 - valarray<>, 504
- max_element()
 - Algorithmus, 461, **494**
- max_size()
 - für Container, 365
 - für deque<>, 383
 - für list<>, 393
 - für map<>, 418
 - für multimap<>, 418
 - für multiset<>, 405
 - für set<>, 405
 - für string, 322
 - für vector<>, 370
- mehrfache Basisklasse, 243
- Mehrfachvererbung, 190, 240–255
 - Namenskonflikte, 242
- Member
 - Daten, 83
 - statische, 125
 - Funktion, 83, 97
 - Aufruf, 85, 97
 - Definition, 83
 - Definition im Klassenrumpf, 89
 - Deklaration, 83
 - explizit inline, 89
 - implizit inline, 89
 - konstante, 99
 - statische, 127
- mem_fun()
 - Elementfunktionsadapter, 453
- mem_fun_ref()
 - Elementfunktionsadapter, 453
- merge()
 - Algorithmus, 460, **485**
 - für list<>, 399
- Methode, 83
- min()
 - Algorithmus, 461, **494**
 - valarray<>, 504
- min_element()
 - Algorithmus, 461, **494**
- minus<>
 - Funktionsobjekt, 450
- mismatch()
 - Algorithmus, 458, **468**
- modf()
 - mathematische Funktion, 496
- Modulare Programmierung, 80
- modulus<>
 - Funktionsobjekt, 450
- multimap<>, 413–427

- !=, 427
- <, 427
- <=, 427
- =, 417
- ==, 427
- >, 427
- >=, 427
- begin(), 420
- clear(), 421
- count(), 424
- Destruktoren, 416
- empty(), 418
- end(), 420
- equal_range(), 425
- erase(), 421, 422
- find(), 424
- Größe, 417
- insert(), 421, 422
- Iteratoren, 418
- key_comp(), 424
- key_compare, 416
- key_type, 416
- Konstruktoren, 416
- lower_bound(), 425
- mapped_type, 416
- max_size(), 418
- rbegin(), 420
- rend(), 420
- size(), 418
- swap(), 421, 427
- upper_bound(), 425
- value_comp(), 424
- value_compare, 416
- Vergleiche, 426
- multiset<>
 - Typen, 415
- multiple Inheritance, *siehe* Mehrfach-
vererbung
- multiplies<>
 - Funktionsobjekt, 450
- multiset<>, 400–413
 - !=, 413
 - <, 413
 - <=, 413
 - =, 404
 - ==, 413
 - >, 413
 - >=, 413
 - begin(), 407
 - clear(), 408
 - count(), 411
 - Destruktoren, 403
 - empty(), 405
 - end(), 407
 - equal_range(), 411
 - erase(), 408, 409
 - find(), 411
 - Größe, 404
 - insert(), 408, 409
 - Iteratoren, 405
 - key_comp(), 410
 - key_compare, 403
 - key_type, 403
 - Konstruktoren, 403
 - lower_bound(), 411
 - max_size(), 405
 - rbegin(), 407
 - rend(), 407
 - size(), 405
 - swap(), 408, 413
 - Typen, 402
 - upper_bound(), 411
 - value_comp(), 410
 - value_compare, 403
 - Vergleiche, 412
- mutable, 101
- Namensbereich, 15–22
 - Alias-Name, 21
 - unbenannter, 21
- namespace, *siehe* Namensbereich
- negate<>
 - Funktionsobjekt, 450
- Negierer, 456
 - not1, 456
 - not2, 456
- <new>
 - Headerdatei, 57, 312
- new, 55, 135, 137
 - Handler, 58
 - nothrow, 58
 - Überladung, 166
 - und Konstruktoren, 88, 119
- new[], 56, 135, 137

- nothrow, 58
 - Überladung, 166
 - und Konstruktoren, 88, 119
- new_handler, 59
- next_permutation()
 - Algorithmus, 460, **486**
- none()
 - für bitset<>, 444
- norm()
 - complex<>, 498
- noshowbase Manipulator
 - ostream, 282
- noshowpoint Manipulator
 - ostream, 282
- noshowpos Manipulator
 - ostream, 283
- noskipws Manipulator
 - istream, 287
- not1
 - Negierer, 456
- not2
 - Negierer, 456
- not_equal_to<>
 - Prädikat, 451
- nothrow, 58
- nouppercase Manipulator
 - ostream, 283
- npos
 - für string, 322
- nth_element()
 - Algorithmus, 460, **484**
- <numeric>
 - Headerdatei, 513
- Objekt, 86
 - aktuelles, 97, 98, 100
 - Freigabe, 94
- oct Manipulator
 - istream, 287
 - ostream, 280
- Öffnen
 - einer Datei, 301, 303
- off_type, 305
- ofstream, 69, 272, 301
 - close(), 303
 - is_open(), 304
 - open(), 304
- oktale
 - Ausgabe, 280
 - Eingabe, 287
- OO
 - Objektorientierung, 1
- OOA
 - objektorientierte Analyse, 1
- OOD
 - objektorientiertes Design, 1
- OOP
 - objektorientierte Programmierung, 1
- open()
 - für Streams, 304
- Operationen
 - für bitset<>, 442–446
 - für complex<>, 498–499
 - für valarray<>, 502–505
- Operator, 135–170
 - !, 135, **137**
 - Überladung, 166
 - für valarray<>, 502
 - !=, 73, 135, **138**
 - als Template, 345
 - für bitset<>, 444
 - für complex<>, 498
 - für Container, 365
 - für deque<>, 387
 - für Iteratoren, 348, 361
 - für list<>, 400
 - für map<>, 427
 - für multimap<>, 427
 - für multiset<>, 413
 - für queue<>, 430
 - für set<>, 413
 - für stack<>, 438
 - für string, 331
 - Überladung, 166
 - für valarray<>, 503
 - für vector<>, 378
 - (), 135, **136**
 - Überladung, **159**, 166
 - (type), 135, **137**
 - Überladung, 166
 - *, 135, **137**, **138**
 - für complex<>, 498
 - für Iteratoren, 348

- Überladung, 166
 - für `valarray<>`, 502
- *, 135, 139**
 - für `complex<>`, 499
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - Überladung, 166
 - für `valarray<>`, 503
- +, 73, 135, 137, 138**
 - für `complex<>`, 498
 - für Iteratoren, 349
 - für `string`, 334, 335
 - Überladung, 166
 - für `valarray<>`, 502
- ++, 135, 136, 137**
 - für Iteratoren, 348
 - Überladung, **163**, 166
- +=, 74, 135, 139**
 - für `complex<>`, 499
 - für Iteratoren, 349
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - für `string`, 334
 - Überladung, 166
 - für `valarray<>`, 503
- ,, 135, 139**
 - für Klassen, 141
 - Überladung, 166
- , 135, 137, 138**
 - für `complex<>`, 498
 - für Iteratoren, 349
 - Überladung, 166
 - für `valarray<>`, 502
- , 135, 136, 137**
 - für Iteratoren, 349
 - Überladung, **163**, 166
- =, 135, 139**
 - für `complex<>`, 499
 - für Iteratoren, 349
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - Überladung, 166
 - für `valarray<>`, 503
- >, 135, 136**
 - für Iteratoren, 348
 - Überladung, **161**, 166
- >*, 135, 137**
 - Überladung, 166
- ., 135, 136**
 - Überladung, 166
- .*, 135, 137**
 - Überladung, 166
- /, 135, 138**
 - für `complex<>`, 498
 - Überladung, 166
 - für `valarray<>`, 502
- /=, 135, 139**
 - für `complex<>`, 499
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - Überladung, 166
 - für `valarray<>`, 503
- ::, 14, 135, 136**
 - Überladung, 166
- <, 73, 135, 138**
 - für Container, 365
 - für `deque<>`, 387
 - für Iteratoren, 349
 - für `pair<>`, 346
 - für `list<>`, 400
 - für `map<>`, 427
 - für `multimap<>`, 427
 - für `multiset<>`, 413
 - für `queue<>`, 430
 - für `set<>`, 413
 - für `stack<>`, 438
 - für `string`, 331
 - Überladung, 166
 - für `valarray<>`, 503
 - für `vector<>`, 378
- <<, 64, 135, 138**
 - für `bitset<>`, 444, 445
 - für `complex<>`, 499
 - für `string`, 341
 - Überladung, 166
 - für `valarray<>`, 502

- <<=**, 135, **139**
 - für `bitset<>`, 443
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - Überladung, 166
 - für `valarray<>`, 503
- >>=**
 - für `bitset<>`, 443
- <=**, 73, 135, **138**
 - als Template, 345
 - für Container, 365
 - für `deque<>`, 387
 - für Iteratoren, 349
 - für `list<>`, 400
 - für `map<>`, 427
 - für `multimap<>`, 427
 - für `multiset<>`, 413
 - für `queue<>`, 430
 - für `set<>`, 413
 - für `stack<>`, 438
 - für `string`, 331
 - Überladung, 166
 - für `valarray<>`, 503
 - für `vector<>`, 378
- =**, 74, 135, **139**
 - für `complex<>`, 497
 - für Container, 366
 - für `deque<>`, 381
 - für Iteratoren, 349
 - für Klassen, 140
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für Iteratoren, 361
 - für `list<>`, 392
 - für `map<>`, 417
 - für `mask_array<>`, 512
 - für `multimap<>`, 417
 - für `multiset<>`, 404
 - für `set<>`, 404
 - für `slice_array<>`, 507
 - für `string`, 326
 - Überladung, **153**, 166
 - für `valarray<>`, 501
 - für `vector<>`, 369
 - virtuell, 218
- ==**, 73, 135, **138**
 - für `bitset<>`, 444
 - für `complex<>`, 498
 - für Container, 365
 - für `deque<>`, 387
 - für `pair<>`, 346
 - für Iteratoren, 348, 361
 - für `list<>`, 400
 - für `map<>`, 427
 - für `multimap<>`, 427
 - für `multiset<>`, 413
 - für `priority_queue<>`, 436
 - für `queue<>`, 430, 431
 - für `set<>`, 413
 - für `stack<>`, 438, 439
 - für `string`, 330
 - Überladung, 166
 - für `valarray<>`, 503
 - für `vector<>`, 378
- >**, 73, 135, **138**
 - als Template, 345
 - für Container, 365
 - für `deque<>`, 387
 - für Iteratoren, 349
 - für `list<>`, 400
 - für `map<>`, 427
 - für `multimap<>`, 427
 - für `multiset<>`, 413
 - für `queue<>`, 430
 - für `set<>`, 413
 - für `stack<>`, 438
 - für `string`, 331
 - Überladung, 166
 - für `valarray<>`, 503
 - für `vector<>`, 378
- >=**, 73, 135, **138**
 - als Template, 345
 - für Container, 365
 - für `deque<>`, 387
 - für Iteratoren, 349
 - für `list<>`, 400
 - für `map<>`, 427
 - für `multimap<>`, 427
 - für `multiset<>`, 413
 - für `queue<>`, 430
 - für `set<>`, 413
 - für `stack<>`, 438
 - für `string`, 331
 - Überladung, 166
 - für `valarray<>`, 503
 - für `vector<>`, 378

- für `string`, 331
- Überladung, 166
- für `valarray<>`, 503
- für `vector<>`, 378
- `>>`, 65, 68, 135, **138**
 - für `bitset<>`, 444, 445
 - für `complex<>`, 499
 - für `string`, 341
 - Überladung, 166
 - für `valarray<>`, 502
- `>>=`, 135, **139**
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - Überladung, 166
 - für `valarray<>`, 503
- `?:`, 135, **138**
 - Überladung, 166
- `[]`, 135, **136**
 - für `bitset<>`, 446
 - für `deque<>`, 385
 - für Iteratoren, 349
 - für `string`, 325
 - Überladung, **157**, 166
 - für `valarray<>`, 501, 505, 511, 512
 - für `vector<>`, 375
- `%`, 135, **138**
 - Überladung, 166
 - für `valarray<>`, 502
- `%=`, 135, **139**
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - Überladung, 166
 - für `valarray<>`, 503
- `&`, 135, **137**, **138**
 - für `bitset<>`, 445
 - für Klassen, 141
 - Überladung, 166
 - für `valarray<>`, 502
- `&=`, 135, **139**
 - für `bitset<>`, 442
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - Überladung, 166
 - für `valarray<>`, 512
- für `slice_array<>`, 507
- Überladung, 166
- für `valarray<>`, 503
- `&&`, 135, **138**
 - Überladung, 166
 - für `valarray<>`, 502
- `^`, 135, **138**
 - für `bitset<>`, 445
 - Überladung, 166
 - für `valarray<>`, 502
- `^=`, 135, **139**
 - für `bitset<>`, 443
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - Überladung, 166
 - für `valarray<>`, 503
- `|`, 135, **138**
 - für `bitset<>`, 445
 - Überladung, 166
 - für `valarray<>`, 502
- `|=`, 135, **139**
 - für `bitset<>`, 442
 - für `gslice_array<>`, 511
 - für `indirect_array<>`, 513
 - für `mask_array<>`, 512
 - für `slice_array<>`, 507
 - Überladung, 166
 - für `valarray<>`, 503
- `||`, 135, **138**
 - Überladung, 166
 - für `valarray<>`, 502
- `~`, 135, **137**
 - Überladung, 166
 - für `bitset<>`, 444
 - für `valarray<>`, 502
- Assoziativität, 135, 142
- Auswertungsreihenfolge, 136
- `const_cast<>()`, 135, 137
 - Überladung, 166
- `delete`, 56, 135, 137
 - Überladung, 166
- `delete[]`, 56, 135, 137
 - Überladung, 166
- `dynamic_cast<>()`, 135, 137, 255

- Überladung, 166
- für Insert-Iteratoren, 355
- für Istream-Iteratoren, 359
- für Iteratoren, 348–350
- für Ostream-Iteratoren, 357
- Konversion, 166–168, 256
- neu definieren, 142
- new**, 55, 88, 119, 135, 137
 - Überladung, 166
- new[]**, 56, 88, 119, 135, 137
 - Überladung, 166
- nicht überladbarer, 142
- Operatorfunktion, 143
- Priorität, 135, 142
- reinterpret_cast<>()**, 135, 137
 - Überladung, 166
- sizeof**, 135, **137**
 - Überladung, 166
- Standard für Klassen, 140
- static_cast<>()**, 135, 137, 255
 - Überladung, 166
- typeid()**, 135, 137
 - Überladung, 166
- überladbarer, 142
- Überladung, 139–166
 - () Funktionsaufruf, **159**
 - ++** Inkrement-Operator, **163**
 - Dekrement-Operator, **163**
 - >** Komponenten-Zugriff, **161**
 - =** Zuweisungsoperator, 152, **153**
 - []** Indexoperator, **157**
 - als Member, 147–150
 - binär als Member, **147**
 - binär global, **143**
 - Ergebnistyp, 152
 - für **enums**, **146**
 - für Standardtypen, 143
 - global, 143–147, 150
 - große Typen, 151
 - Grundlagen, 142–143
 - in Standardbibliothek, 168
 - Kombinationen, 143
 - nur als Member, 153
 - Semantik, 143
 - Signatur, 150
 - spezielle Operatoren, 153–164
 - Standardparameter, 151
 - Syntax, 143
 - Typumwandlung, 150
 - Übersicht, 164
 - unär als Member, **149**
 - unär global, **145**
 - Übersicht, 135
 - Vorrang, 135
- ostream**, 64, 272
 - dec** Manipulator, 280
 - endl** Manipulator, 276
 - ends** Manipulator, 277
 - fill()**, 279
 - fixed** Manipulator, 281
 - flags()**, 283
 - flush()**, 276
 - flush** Manipulator, 276
 - hex** Manipulator, 281
 - internal** Manipulator, 280
 - Iterator, 357
 - left** Manipulator, 280
 - noshowbase** Manipulator, 282
 - noshowpoint** Manipulator, 282
 - noshowpos** Manipulator, 283
 - nouppercase** Manipulator, 283
 - oct** Manipulator, 280
 - precision()**, 278
 - put()**, 275
 - resetiosflags()** Manipulator, 285
 - right** Manipulator, 280
 - scientific** Manipulator, 281
 - seekp()**, 305
 - setbase()** Manipulator, 281
 - setfill()** Manipulator, 279
 - setiosflags()** Manipulator, 285
 - setprecision()** Manipulator, 279
 - setw()** Manipulator, 278
 - showbase** Manipulator, 282
 - showpoint** Manipulator, 282
 - showpos** Manipulator, 283
 - streamsize**, 276
 - tellp()**, 305
 - uppercase** Manipulator, 283
 - width()**, 278
 - write()**, 275
- ostreamstream**, 272, 342
 - str()**, 343
- ostrstream**, 342

- out_of_range, 312, 324, 326, 327, 329,
330, 332, 333, 335, 336, 375, 385,
441, 443, 444
- Output-Iterator, 348
 - output_iterator_tag, 350
- output_iterator_tag, 350
- overflow_error, 312

- pair <> Template-Klasse, 346
 - < Vergleichsoperator, 346
 - == Vergleichsoperator, 346
 - makepair(), 347
- Parameter
 - Standard, 38
- partial_sort()
 - Algorithmus, 460, **483**
- partial_sort_copy()
 - Algorithmus, 460, **483**
- partial_sum(), **515**
- partition()
 - Algorithmus, 459, **481**
- peek()
 - istream, 286
- Platzieren von Objekten, 60
- plus<>
 - Funktionsobjekt, 450
- pointer
 - Container, 360
- polar()
 - complex<>, 497
- Polarkoordinaten, 497, 498
- polymorphe Klasse, 224
- Polymorphie, **220**, 255
- pop()
 - für queue<>, 431
 - für stack<>, 439
- pop_back()
 - für deque<>, 380, 386
 - list<>, 388
 - für list<>, 396
 - für vector<>, 377
- pop_front()
 - für deque<>, 379, 380, 387
 - list<>, 388
 - für list<>, 396
- poph()
 - für priority_queue<>, 436
- pop_heap()
 - Algorithmus, 461, **493**
- pos_type, 305
- pow()
 - complex<>, 499
 - mathematische Funktion, 496
 - für valarray<>, 505
- Prädikat, 451
 - equal_to<>, 451
 - greater<>, 451
 - greater_equal<>, 451
 - less<>, 451
 - less_equal<>, 451
 - logical_and<>, 451
 - logical_not<>, 451
 - logical_or<>, 451
 - not_equal_to<>, 451
- Präzision, 278
- precision()
 - ostream, 278
- prev_permutation()
 - Algorithmus, 460, **486**
- Priorität, 135, 142
- priority_queue<>
 - ==, 436
 - Containeradapter, 432–436
 - container_type, 434
 - Destruktor, 436
 - empty(), 435
 - poph(), 436
 - push(), 435
 - size(), 435
 - size_type, 434
 - top(), 435
 - value_type, 434
- private, 83, **86**
 - Vererbung, 196
- Programmierparadigmen, 79
 - akstrakte Datentypen, 82
 - generische Programmierung, 171
 - modulare Programmierung, 80
 - prozedurale Programmierung, 79
- protected, 83, **86**
 - Vererbung, 195
- Prozedurale Programmierung, 79
- ptrdiff_t, 351
- ptr_fun()

- Funktionszeigeradapter, 453
- public, 83, **86**
- Vererbung, 193
- push()
 - für priority_queue<>, 435
 - für queue<>, 431
 - für stack<>, 439
- push_back()
 - für deque<>, 380, 386
 - list<>, 388
 - für list<>, 395
 - für string, 334
 - für vector<>, 376
- push_front()
 - für deque<>, 379, 380, 386
 - list<>, 388
 - für list<>, 396
- push_heap()
 - Algorithmus, 461, **492**
- put()
 - ostream, 275
- putback()
 - istream, 286
- <queue>
 - Headerdatei, 427, 432
- queue<>
 - !=, 430
 - <, 430
 - <=, 430
 - ==, 430, 431
 - >, 430
 - >=, 430
 - back(), 431
 - Containeradapter, 427–432
 - container_type, 430
 - Destruktor, 432
 - empty(), 431
 - front(), 431
 - Konstruktor, 431, 435
 - Konstruktoren, 430
 - pop(), 431
 - push(), 431
 - size(), 431
 - size_type, 430
 - value_type, 430
- Random-Access-Iterator, 349
- random_access_iterator_tag, 350
- random_access_iterator_tag, 350
- random_shuffle()
 - Algorithmus, 459, **480**
- range_error, 312
- rbegin()
 - Container, 361
 - für deque<>, 384
 - für list<>, 394
 - für map<>, 420
 - für multimap<>, 420
 - für multiset<>, 407
 - für set<>, 407
 - für string, 320, 321
 - für vector<>, 373
- rdstate()
 - ios, 289
- read()
 - istream, 286
- real()
 - complex<>, 497
- reference
 - Container, 360
- Referenz, 22–26, 66
 - als Funktionsergebnis, 24
 - als Funktionsparameter, 23
 - als Komponente, 128
 - auf const, 30
- reinterpret_cast<>(), 13, 135, 137
 - Überladung, 166
- remove()
 - Algorithmus, 459, **476**
 - für list<>, 398
- remove_copy()
 - Algorithmus, 459, **477**
- remove_copy_if()
 - Algorithmus, 459, **477**
- remove_if()
 - Algorithmus, 459, **476**
 - für list<>, 398
- rend()
 - Container, 361
 - für deque<>, 384
 - für list<>, 394
 - für map<>, 420
 - für multimap<>, 420
 - für multiset<>, 407

- für `set<>`, 407
 - für `string`, 320, 321
 - für `vector<>`, 373
- `replace()`
 - Algorithmus, 459, **474**
 - für `string`, 336
- `replace_copy()`
 - Algorithmus, 459, **474**
- `replace_copy_if()`
 - Algorithmus, 459, **474**
- `replace_if()`
 - Algorithmus, 459, **474**
- replizierte Basisklasse, *siehe* *mehrfache Basisklasse*
- `reserve()`
 - für `string`, 323
 - für `vector<>`, 371
- Reservespeicher, 59
- `reset()`
 - für `bitset<>`, 443
- `resetiosflags` Manipulator
 - `ostream`, 285
- `resize()`
 - für `deque<>`, 383
 - für `list<>`, 393
 - für `string`, 322
 - `valarray<>`, 501
 - für `vector<>`, 371
- Ressourcenmanagement, 124
- `reverse()`
 - Algorithmus, 459, **478**
- Reverse-Iterator, 354
 - `base()`, 355
- `reverse_copy()`
 - Algorithmus, 459, **479**
- `reverse_iterator`
 - für `string`, 320
- `rfind()`
 - für `string`, 339
- `right` Manipulator
 - `ostream`, 280
- `rotate()`
 - Algorithmus, 459, **479**
- `rotate_copy()`
 - Algorithmus, 459, **479**
- RTTI*, *siehe* *Laufzeittypinformation*
- `runtime_error`, 312
- Schablone, *siehe* *Template*
- Schließen
 - einer Datei, 301, 303
- Schlüsselwort
 - `catch`, 48
 - `class`, 87
 - `export`, 188
 - `friend`, 133
 - `mutable`, 101
 - `private`, 83, **87**, 196
 - `protected`, 83, **87**, 195
 - `public`, 83, **87**, 193
 - `static`, 125
 - `struct`, 83
 - `this`, 98
 - `throw`, 48
 - `try`, 48
 - `typedef`, 131, 182
 - `typename`, 352
 - `using`, 17, 198
 - `virtual`, 208
- Schnittstelle, 195–197
- Schnittstellenklasse, 239
- `scientific` Manipulator
 - `ostream`, 281
- `search()`
 - Algorithmus, 458, **468**
- `search_n()`
 - Algorithmus, 458, **469**
- `seekg()`
 - `istream`, 306
- `seekp()`
 - `ostream`, 305
- Sequenz, 347
- `<set>`
 - Headerdatei, 400
- `map<>`
 - Typen, 415
- `set()`
 - für `bitset<>`, 443
- `set<>`, 400–413
 - `!=`, 413
 - `<`, 413
 - `<=`, 413
 - `=`, 404
 - `==`, 413
 - `>`, 413

- `>=`, 413
- `begin()`, 407
- `clear()`, 408
- `count()`, 411
- Destruktoren, 403
- `empty()`, 405
- `end()`, 407
- `equal_range()`, 411
- `erase()`, 408, 409
- `find()`, 411
- Größe, 404
- `insert()`, 408, 409
- Iteratoren, 405
- `key_comp()`, 410
- `key_compare`, 403
- `key_type`, 403
- Konstruktoren, 403
- `lower_bound()`, 411
- `max_size()`, 405
- `rbegin()`, 407
- `rend()`, 407
- `size()`, 405
- `swap()`, 408, 413
- Typen, 402
- `upper_bound()`, 411
- `value_comp()`, 410
- `value_compare`, 403
- Vergleiche, 412
- `setbase()` Manipulator
 - `ostream`, 281
- `set_difference()`
 - Algorithmus, 460, **490**
- `setf()`
 - `ios_base`, 279, 287
- `setfill()` Manipulator
 - `ostream`, 279
- `set_intersection()`
 - Algorithmus, 460, **489**
- `setiosflags` Manipulator
 - `ostream`, 285
- `set_new_handler()`, 58
- `setprecision()` Manipulator
 - `ostream`, 279
- `setstate()`
 - `ios`, 290
- `set_symmetric_difference()`
 - Algorithmus, 460, **491**
- `set_terminate()`, 311
- `set_unexpected()`, 54, 310
- `set_union()`
 - Algorithmus, 460, **488**
- `setw()` Manipulator
 - `istream`, 286
 - `ostream`, 278
- `shift()`
 - `valarray<>`, 504
- `showbase` Manipulator
 - `ostream`, 282
- `showpoint` Manipulator
 - `ostream`, 282
- `showpos` Manipulator
 - `ostream`, 283
- Signatur, 8, 209
- simple Inheritance, *siehe* einfache Vererbung
- `sin()`
 - `complex<>`, 499
 - mathematische Funktion, 496
 - `valarray<>`, 505
- `sinh()`
 - `complex<>`, 499
 - mathematische Funktion, 496
 - `valarray<>`, 505
- `size()`
 - für `bitset<>`, 442
 - für Container, 364
 - für `deque<>`, 383
 - für `list<>`, 393
 - für `map<>`, 418
 - für `multimap<>`, 418
 - für `multiset<>`, 405
 - für `priority_queue<>`, 435
 - für `queue<>`, 431
 - für `set<>`, 405
 - für `slice`, 505
 - für `stack<>`, 439
 - für `string`, 322
 - `valarray<>`, 500
 - für `vector<>`, 370
- `sizeof`, 135, **137**
 - Überladung, 166
- `size_t`, 276
- `size_type`
 - Container, 360

- priority_queue<>, 434
- queue<>, 430
- stack<>, 438
 - für string, 322
- skipws Manipulator
 - istream, 287
- slice, 505–509
 - size(), 505
 - slice_array<>
 - *, 507
 - +=, 507
 - =, 507
 - /=, 507
 - <<=, 507
 - =, 507
 - >>=, 507
 - %=, 507
 - &=, 507
 - ^=, 507
 - |=, 507
 - fill(), 506
 - Operationen, 506–509
 - start(), 505
 - stride(), 505
- slice_array<>, *siehe* unter slice
- Softwarebaustein, 91
- sort()
 - Algorithmus, 460, **482**
 - für list<>, 398, 399
- sort_heap()
 - Algorithmus, 461, **493**
- Speichermangel, 57
 - Reservespeicher, 59
- splice()
 - für list<>, 397, 398
- sqrt()
 - complex<>, 499
 - mathematische Funktion, 496
 - valarray<>, 505
- <sstream>
 - Headerdatei, 274, 342
- stable_partition()
 - Algorithmus, 459, **481**
- stable_sort()
 - Algorithmus, 460, **482**
- <stack>
 - Headerdatei, 436
- stack<>
 - !=, 438
 - <, 438
 - <=, 438
 - ==, 438, 439
 - >, 438
 - >=, 438
 - Containeradapter, 436–439
 - container_type, 438
 - Destruktor, 439
 - empty(), 439
 - Konstruktor, 438
 - pop(), 439
 - push(), 439
 - size(), 439
 - size_type, 438
 - top(), 439
 - value_type, 438
- Standard
 - Ausgabe, 64, 273
 - Eingabe, 64, 273
 - Fehlerausgabe, 64, 273
 - Protokollkanal, 273
- Standardbibliothek, 267–516
 - Einblick, 63–75
- Standardcontainer, *siehe* Container
- Standarddestruktor, 95, 122
- Standardkonstruktor, 88, 107
 - für Istream-Iteratoren, 358
- Standardparameter, 38
- start()
 - für slice, 505
- static, 125
- static_cast<>(), 12, 135, 137, 255
 - Überladung, 166
- stderr, 64
- <stdexcept>
 - Headerdatei, 226, 227, 311
- stdin, 64
- stdout, 64
- str()
 - istreamstream, 343
 - ostreamstream, 343
 - stringstream, 343
- streambuf, 272
- streamsize, 276, 285, 287
- stride()

- für slice, 505
- <string>
 - Headerdatei, 70, 317
- string, 70–75, 317–344
 - !=, 331
 - +, 334, 335
 - +=, 334
 - <, 331
 - <=, 331
 - =, 326
 - ==, 330
 - >, 331
 - >=, 331
 - [], 325
 - Anhängen, 74
 - append(), 333
 - assign(), 327
 - at(), 326
 - begin(), 319, 321
 - capacity(), 322
 - clear(), 335
 - compare(), 329
 - const_iterator, 321
 - const_reverse_iterator, 321
 - copy(), 329
 - c_str(), 328
 - data(), 328
 - Destruktor, 325
 - Ein-/Ausgabe, 71
 - empty(), 322
 - end(), 319, 321
 - erase(), 335
 - Erzeugung, 71
 - find(), 338
 - find_first_not_of(), 340
 - find_first_of(), 339
 - find_last_not_of(), 340
 - find_last_of(), 340
 - getline(), 341
 - insert(), 332
 - iterator, 319, 321
 - Iteratoren, 318–321
 - Konstruktor, 323
 - length(), 322
 - max_size(), 322
 - npos, 322
 - push_back(), 334
 - rbegin(), 320, 321
 - rend(), 320, 321
 - replace(), 336
 - reserve(), 323
 - resize(), 322
 - reverse_iterator, 320, 321
 - rfind(), 339
 - size(), 322
 - size_type, 322
 - substr(), 335
 - swap(), 342
 - Teilstring, 335
 - Vergleich, 72
 - Verketteten, 73
 - Zugriff auf einzelne Zeichen, 72, 325
 - Zuweisung, 74, 326
- stringbuf, 272
- String-Stream, 272, 307, 342
- stringstream, 272, 342
 - str(), 343
- <strstream>
 - Headerdatei, 342
- strstream, 342
- struct, 83
 - implizit public, 87
- substr()
 - für string, 335
- sum()
 - valarray<>, 504
- swap()
 - Algorithmus, 459, **473**
 - für Container, 366
 - für deque<>, 386, 387
 - für list<>, 396, 400
 - für map<>, 421, 427
 - für multimap<>, 421, 427
 - für multiset<>, 408, 413
 - für set<>, 408, 413
 - für string, 342
 - für vector<>, 378
- swap_ranges()
 - Algorithmus, 459, **473**
- sync()
 - istream, 286
- tan()
 - complex<>, 499

- mathematische Funktion, 496
- valarray<>, 505
- tanh()
 - complex<>, 499
 - mathematische Funktion, 496
 - valarray<>, 505
- tellg()
 - istream, 306
- tellp()
 - ostream, 305
- Template, 171–188
 - Default-Argument, 177
 - Element-Template, 185
 - export, 188
 - Funktion, 171–180
 - Grundlagen, 171
 - Typumwandlung, 173
 - gewöhnliche Parameter, 176, 183
 - Implementierungsmöglichkeiten, 187
 - Instanziierung, 173, 174, 182
 - Klasse, 180–187
 - Grundlagen, 180
 - Parameter, 175, 182
 - partielle Spezialisierung, 179
 - Spezialisierung, 178, 184
 - Template als Parameter, 177, 183
 - Typparameter, 175, 183
 - Überladung, 178
 - und Vererbung, 187, **227**
- terminate(), 311
- terminate_handler, 310
- test()
 - für bitset<>, 444
- this, 98
- this-Zeiger, 98
 - const, 101
- throw, 48
- tie()
 - istream, 307
- top()
 - für priority_queue<>, 435
 - für stack<>, 439
- to_string()
 - für bitset<>, 445
- to_ulong()
 - für bitset<>, 445
- transform()
 - Algorithmus, 459, **472**
- try, 48
- try-Block, 48
- typedef, 131, 182
- typeid(), 135, 137, **226**
 - Überladung, 166
- <typeinfo>
 - Headerdatei, 226, 313
- Typen
 - Standard, 10
- typename, 352
- Typumwandlung, 12, 255
 - bei Templates, 173
 - bitset<>, 445
 - Konversionoperator, 166, 169
 - mit Konstruktor, 118, 166, 169, 256
 - selbstdefinierte, 166
 - von C-String nach C++String, 323
- Überdecken, 213
- Überladen, 213
- Überladung
 - von Funktionen, **43**, 169
 - von Operatoren, 139–166, 169
- unary_function
 - Basisklasse zu Funktionsobjekten, 450
- uncaught_exception(), 124, 311
- underflow_error, 312
- unexpected(), 310
- unexpected_handler, 54, 310
- unget()
 - istream, 286
- unique()
 - Algorithmus, 459, **477**
 - für list<>, 398
- unique_copy()
 - Algorithmus, 459, **478**
- unsetf()
 - ios_base, 279, 287
- Upcast, 255
- upper_bound()
 - Algorithmus, 460, **484**
 - für map<>, 425
 - für multimap<>, 425
 - für multiset<>, 411
 - für set<>, 411

- uppercase Manipulator
 - ostream, 283
- using
 - Deklaration, 17
 - bei Vererbung, 198
 - Direktive, 17
- <valarray>
 - Headerdatei, 500
- valarray<>, 499–513
 - !, 502
 - !=, 503
 - , 502
 - *, 503
 - +, 502
 - +=, 503
 - , 502
 - =, 503
 - /, 502
 - /=, 503
 - <, 503
 - <<, 502
 - <<=, 503
 - <=, 503
 - =, 501
 - ==, 503
 - >, 503
 - >=, 503
 - >>, 502
 - >>=, 503
 - [], 501
 - %, 502
 - %=, 503
 - &, 502
 - &=, 503
 - &&, 502
 - ^, 502
 - ^=, 503
 - |, 502
 - |=, 503
 - ||, 502
 - , 502
 - abs(), 505
 - acos(), 505
 - als Matrix, 507
 - apply(), 504
 - asin(), 505
 - atan(), 505
 - atan2(), 505
 - cos(), 505
 - cosh(), 505
 - cshift(), 504
 - Erzeugung, 500, 507, 511–513
 - exp(), 505
 - free(), 500
 - gslice, 509–511
 - gslice_array<>, 509–511
 - indirect_array<>, 512
 - *=, 513
 - +=, 513
 - =, 513
 - /=, 513
 - <<=, 513
 - =, 513
 - >>=, 513
 - %=, 513
 - &=, 513
 - =^=, 513
 - |=, 513
 - fill(), 513
 - Operationen, 513
- indirekte, 512
- Indizierung, **501**, 505, 511, 512
- Konstruktoren, 500
- log(), 505
- log10(), 505
- mask_array<>, 512
 - *=, 512
 - +=, 512
 - =, 512
 - /=, 512
 - <<=, 512
 - =, 512
 - >>=, 512
 - %=, 512
 - &=, 512
 - ^=, 512
 - |=, 512
- fill(), 512
- Operationen, 512
- Masken, 512
- mathematische Funktionen, 504
- max(), 504
- min(), 504

- Operatoren, 502–505
 - binäre, 502
 - unäre, 502
 - Vergleiche, 503
- pow(), 505
- resize(), 501
- shift(), 504
- sin(), 505
- sinh(), 505
- size(), 500
- slice, 505–509
- slice_array<>, 505–509
- sqrt(), 505
- sum(), 504
- tan(), 505
- tanh(), 505
- Zuweisung, **501**, 507, 512, 513
- value_comp()
 - für map<>, 424
 - für multimap<>, 424
 - für multiset<>, 410
 - für set<>, 410
- value_compare
 - map<>, 416
 - multimap<>, 416
 - multiset<>, 403
 - set<>, 403
- value_type
 - Container, 360
 - priority_queue<>, 434
 - queue<>, 430
 - stack<>, 438
- Variable
 - Deklaration, 13
- <vector>
 - Headerdatei, 368
- vector<>, 368–378
 - !=, 378
 - <, 378
 - <=, 378
 - =, 369
 - ==, 378
 - >, 378
 - >=, 378
 - [], 375
 - assign(), 369, 370
 - at(), 375
 - begin(), 373
 - capacity(), 370
 - clear(), 378
 - Destruktoren, 368
 - empty(), 370
 - end(), 373
 - erase(), 377
 - Größe, 370
 - insert(), 377
 - Iteratoren, 372
 - Kapazität, 370
 - Konstruktoren, 368
 - max_size(), 370
 - pop_back(), 377
 - push_back(), 376
 - rbegin(), 373
 - rend(), 373
 - reserve(), 371
 - resize(), 371
 - size(), 370
 - swap(), 378
 - Typen, 368
 - Vergleiche, 378
- vector<bool>, 378
 - flip(), 379
- Vektor, *siehe* vector<>
 - mathematischer, 499–513
- Vererbung, 189–264
 - Anwenderschnittstelle, 197
 - Ausnahmen, 234
 - einfache, 189–191
 - Grundlagen, 189–191
 - Mehrfachvererbung, 190, 240–255
 - Memberfunktion neudefinieren, 203
 - Probleme, 207
- private, 196
- protected, 195
- public, 193
- Schnittstelle, 195–197
- und Destruktoren, **233**
- und dynamische Komponenten, 233
- und Konstruktoren, **231**
- und Templates, 187, **227**
- using-Deklaration, 198
- Vererbungsschnittstelle, 197
- virtuelle Funktion, 208
- Zugriffsschutz, 191

Vergleich

- für `bitset<>`, 444
- für `Container`, 365
- für `deque<>`, 387
- für Iteratoren, 348, 349, 361
- für Iteratoren, 365
- für `list<>`, 400
- für `map<>`, 427
- für `multimap<>`, 427
- für `multiset<>`, 413
- für `priority_queue<>`, 436
- für `queue<>`, 430
- für `set<>`, 413
- für `stack<>`, 438
- für `string`, 72, 329
- für `vector<>`, 378

Verketteten

- von `strings`, 334
- von `strings`, 73

`virtual`, 208

virtuelle Basisklasse, 247

Vorrang, 135

`wcerr`, 274

`wchar_t`, 272

`wcin`, 274

`wclog`, 274

`wcout`, 274

`wfilebuf`, 273

`wfstream`, 273

`what()`

- Fehlerstring liefern, 309

`width()`

- `istream`, 286

- `ostream`, 278

`wifstream`, 273

`wios`, 273

`wiostream`, 273

`wistream`, 273

`wistringstream`, 273

`wofstream`, 273

`wostream`, 273

`wostringstream`, 273

`write()`

- `ostream`, 275

`ws` Manipulator

- `istream`, 68, 287

`wstreambuf`, 273

`wstring`, 317

`wstringbuf`, 273

`wstringstream`, 273

Zeiger

- als Iterator, 351

- auf `const`, 28

- auf Komponente, 129

- `this`, 98

Zugriffsabschnitt, 86, 133, 192

- mehrfache Nennung, 87

- `private`, 83, **86**

- `protected`, 83, **86**

- `public`, 83, **86**

Zugriffsschutz, 83, **86**

- Vererbung, 191

Zuweisung

- `complex<>`, 497

- für Klassen, 140

- für `string`, 326

- virtuell, 218

Zwischenraumzeichen, 66