Skriptum

Objektorientierte Programmierung (OOP)

WS 2018/19

Stand 20.09.2018

Prof. Dr. Alfred Irber
LbA W. Tasin, M.Sc.

Inhaltsangabe

1 Grundkonzepte der Objektorientierten Programmierung (OOP)

1.1 Einführung

- **Grundgedanke der OOP:**

OOP ist eine **Softwareentwicklungsmethodik**, deren **Grundidee** aus der **Simulationstechnik** stammt :
In dieser werden die Objekte der realen Welt sowie deren Beziehungen durch entsprechende Strukturen im Rechner abgebildet.
In der OOP wird dieses Prinzip auf alle Arten von Informationen und Abläufen – auch auf solche abstrakter Natur – angewendet.
Der Aufgabenbereich eines zu lösenden Problems wird in **Objekte** und die zwischen ihnen bestehenden **Beziehungen** zerlegt. Diese werden in einem das entsprechende Problem lösenden Programm nachgebildet.

→ *Ein **OOP-Programm** besteht somit im wesentlichen aus einer Anzahl miteinander in **Beziehung stehender Objekte**.*

Diese Denk- und Vorgehensweise ermöglicht es, auch sehr **umfangreiche** und **komplexe** Aufgaben auf einem relativ **hohem Abstraktionsniveau** erfolgreich zu bearbeiten.
Sie nutzt die intellektuellen Fähigkeiten aus, die der Mensch zur Bewältigung der ihn umgebenden Komplexität entwickelt hat.
Dies sind im wesentlichen die Fähigkeiten des **Abstrahierens**, des **Klassifizierens** und des **Generalisierens**.

Auf ihnen beruhen die **Grundkonzepte** der OOP :

- Bildung von **Objekten**
- **Abstraktion** der Objekte durch **Klassen**
- **Kapselung**
- **Vererbung**
- **Polymorphie**

1.2 Grundkonzepte der OOP : Objekte und Klassen

- **Konkrete Objekte:**
  - ◇ **Objekte** sind in einem bestimmten Sinn abgeschlossene Einheiten, die durch zwei Aspekte gekennzeichnet sind :
    - ▻ Sie besitzen einen (inneren) **Zustand**
    - ▻ und sie verfügen über **Fähigkeiten**, d.h. sie können bestimmte Operationen – aktiv oder passiv – ausführen. Diese Fähigkeiten und damit das dadurch bestimmte **Verhalten** können von außen angefordert, aktiviert werden. Die Aktivierung der Fähigkeiten kann eine Zustandsänderung bewirken.

  - ◇ **Beispiel** :

Willies Uhr	
Zustand :	aktuelle Zeit
Fähigkeiten :	Uhr stellen Zeit fortschalten (ticken) Zeit darstellen

- **Objekte in der OOP**

  - ◇ In der **OOP** stehen **Objekte im Vordergrund**.
    Sie bilden die grundlegenden **Strukturierungseinheiten** eines OOP-Programmms
    Dabei kann ein Objekt **sehr konkret** aber auch **beliebig abstrakt** sein, es kann ein statisches Gebilde (z.B. ein Auto), oder einen dynamischen Ablauf (Vorgang) beschreiben (z.B. ein Tennisspiel).
    Der (innere**) Zustand** des Objekts wird durch **Datenstrukturen** (Datenkomponenten), seine **Fähigkeiten** – die von ihm ausführbaren Operationen – werden durch **Funktionen** (Prozeduren) beschrieben.
    Die **Datenkomponenten** werden auch als **Attribute**, die Funktionen (**Memberfunktionen**) als **Methoden** bezeichnet.

  - ◇ Ein Objekt verbindet also **Daten** und die zu ihrer Bearbeitung dienenden **Funktionen** (Code !) zu einer **Einhei**t.
    → Die von einem Objekt ausführbaren **Methoden** (= Funktionen) sind **Bestandteil des Objekts** und nur als solche relevant. Dies steht im **Gegensatz zur konventionellen** (prozeduralen, imperativen) Programmierung, bei der Daten und Code **getrennt** sind, wobei der Code (Prozeduren, Funktionen) eigenständig ist und im Vordergrund steht : Code wird **auf** Daten angewendet.

  - ◇ In der **OOP** wird eine **Methode für ein Objekt aufgerufen**, in dem an das Objekt – i.a. durch ein anderes Objekt – eine entsprechende **Nachricht** (**Botschaf**t) geschickt wird : Das **Objek**t interpretiert die Nachricht und **reagiert** mit der Ausführung einer zugeordneten **Operation** (Methode).
    Zwischen den Objekten bestehen also **Kommunikationsbeziehungen**.

    → *Ein OOP-Programm besteht im wesentlichen aus einer Ansammlung miteinander **kommunizierender** und dadurch interagierender **Objekte**.*

  - ◇ Der OOP-Ansatz erfordert eine **andere Vorgehensweise** bei der **Problemlösung** :
    Statt einer Top-Down-Zerlegung des Problems (→ hierarchische Modularisierung) müssen die **relevanten Objekte** (Aufbau und Verhalten) des Problems und die zwischen ihnen bestehenden Beziehungen ermittelt werden (→ **aufgaben- und kommunikationsorientierte** Zerlegung)

Grundkonzepte der OOP : Objekte und Klassen (2)

- **Klassen**

  ◇ *Der **Aufbau**, die **Eigenschaften** und **Fähigkeiten** (Verhaltensweisen) von **Objekten** werden durch **Klassen** beschrieben.*

  ◇ Eine Klasse legt die Datenkomponenten (Datenstruktur) und die Methoden zur Bearbeitung der Daten (Memberfunktionen) für eine **Menge gleichartiger Objekte** – d.h. Objekte mit gemeinsamen Merkmalen und gleichen Fähigkeiten, die diese von anderen Objekten unterscheiden – fest.

  ◇ Ein spezielles Objekt der durch eine Klasse definierten Objektmenge wird auch **Instanz** genannt.
  Es unterscheidet sich von einem anderen Objekt (einer anderen Instanz) der gleichen Klasse nur durch seinen jeweiligen Zustand, d.h. den Werten seiner Datenkomponenten.

  ⇒ Die **Klasse** entspricht dem **Datentyp** prozeduraler Programmiersprachen,
  während eine **Instanz** (ein spezielles **Objekt** dieser Klasse) einer **Variablen** entspricht.

  ◇ Eine Klasse kann formal als **Erweiterung** einer **Struktur** (`struct`-Typ) **in C** aufgefasst werden.
  Gegenüber einer nur aus Datenkomponenten bestehenden Struktur besitzt eine Klasse zusätzlich Funktionskomponenten.

  ◇ **Beispiel** : Klasse **Uhr**

Uhr	
Datenkomp:	`actTime`

Funktionskomp :	`setTime(...)`
	`tick()`
	`displayClock()`

  ◇ **Jedes Objekt** (Instanz) einer Klasse hat einen **eigenen** (inneren) **Zustand**.
  → Die **Datenkomponenten** existieren für **jedes Objekt** (Unterschied zu Modulen der prozeduralen Programmierung).
  Sie werden erst geschaffen, wenn das Objekt generiert wird (→ "Variablendefinition").

  ◇ Die **Methoden** (Funktionen) existieren dagegen **nur einmal pro Klasse**.
  Sie werden durch die Definition der Klasse (→ "Typdefinition") geschaffen.
  → Auch wenn es gar keine Objekte dieser Klasse gibt, existieren die Methoden.
  Jedes Objekt einer Klasse arbeitet mit demselben Code (→ ***Code Sharing***)

  Beim Aufruf einer Methode für ein spezielles Objekt, wird dieser eine Referenz auf das Objekt als verborgener Parameter übergeben. Dadurch kann die Methode zu sämtlichen Komponenten (also auch den Datenkomponenten) dieses Objekts zugreifen.
  → Durch den **Aufruf der Methode** wird diese einem **speziellen Objekt zugeordnet**.

1.3 Grundkonzepte der OOP : Kapselung

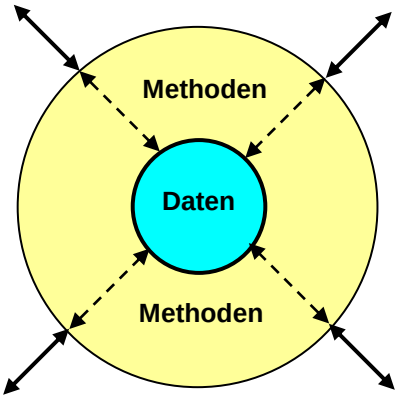
- **Kapselung (*Encapsulation*):**

  ◇ Der **Benutzer** eines **Objekts** (allg.: Anwenderprogramm, speziell : anderes Objekt) braucht seinen **genauen Aufbau nicht** zu **kennen.**
  Ihm müssen lediglich die **Methoden**, die er für eine **Interaktion** mit dem Objekt **benötigt**, d.h. über die er die Fähigkeiten des Objekts aktivieren und dessen Zustand verändern kann, **bekannt** zu sein.
  Von der **internen Darstellung** der den jeweiligen Objektzustand festlegenden **Daten** braucht er dagegen **keinerlei Kenntnis** zu haben.

  ◇ Nur die **Funktionen** (= Methoden) eines **Objekts**, die zu seiner **Verwendung** tatsächlich benötigt werden, werden **allgemein zugänglich**, d.h. öffentlich (***public***), gemacht.
  Sie bilden das **Interface** (Protokoll), über das zu dem Objekt kontrolliert zugegriffen werden kann, d.h. sie bilden seine **Schnittstelle** zur "Außenwelt".
  Die **Daten** (u. gegebenenfalls reine Hilfs- und Verwaltungsfunktionen) sind nur Komponenten des Objekts selbst zugänglich, d.h. privat (***private***).
  Der "Außenwelt" gegenüber bleiben sie verborgen → Sie sind nach außen **gekapselt**.

  Hierdurch wird sichergestellt, dass zu einem Objekt nur über eine **wohldefinierte Schnittstelle zugegriffen** werden kann.→ **Klassen-Schnittstelle**
  Zugriffe zu Interna, die nur zur internen Realisierung und Verwaltung des Objekts dienen, sind nicht möglich
  → Vermeidung von Fehlern durch Verhinderung eines direkten und unkontrollierten Zugriffs

  ⇒ **Trennung von Interface (Schnittstelle) und Implementierung**.



  ◇ Die Kapselung bewirkt außerdem eine **Datenabstraktion** (***data abstraction***) :
  Eine Datenstruktur ist - nach außen - nicht mehr an eine bestimmte Implementierung gebunden, sondern wird allein über die auf sie anwendbaren Operationen (Methoden, Funktionen) definiert.
  (→ **abstrakter Datentyp**, **ADT**)

  → *Eine **Änderung der Implementierung** - bei **gleichbleibendem Interface** – hat **keinen Einfluß** auf den **Anwendungscode**.*

1.4 Grundkonzepte der OOP : Vererbung

- **Vererbung (*Inheritance*) :**

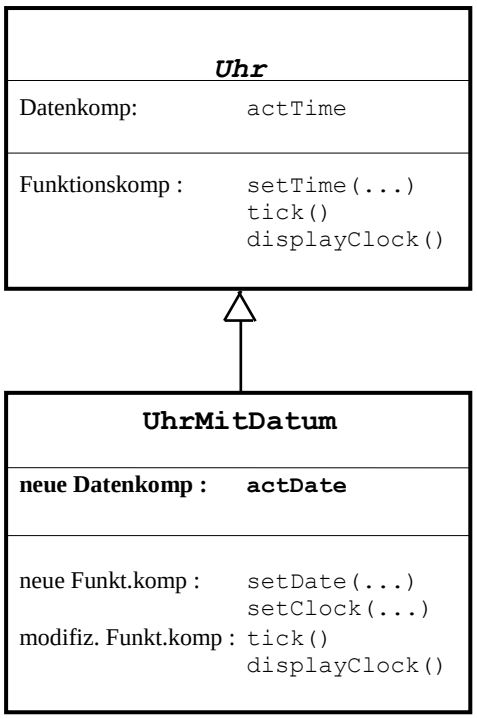
  ◇ **Weitergabe von Eigenschaften** (Daten und Funktionen) eines **Objekts** an ein **anderes Objekt**.
  Zusätzlich zu den ererbten Eigenschaften kann ein Objekt **neue spezifische Eigenschaften** (Daten und Funktionen) besitzen bzw. bestimmte **Eigenschaften modifizieren**.
  → *Schaffung einer **neuen Art** von Objekten durch **Erweiterung** einer bestehenden Art von Objekten.*

  ◇ Die Vererbung führt zum Aufbau von **Klassenhierarchien** :
  Eine neue Klasse wird aus einer - oder mehreren - bereits definierten Klasse(n) **abgeleitet**.
  vorhandenene Klasse : **Basisklasse**, Elternklasse, Oberklasse
  neue Klasse : **abgeleitete Klasse**, Kindklasse, Unterklasse

  Die abgeleitete Klasse **erbt** die **Daten** und **Methoden** der Basisklasse(n).
  Dabei können geerbte Methoden **abgeändert** ("überschrieben") werden.
  Zusätzlich kann die abgeleitete Klasse **neue Daten** und **Methoden** besitzen.
  → Abänderung und Ergänzung im Sinne einer weiteren **Spezialisierung**

  ◇ **Beispiel :** Ableitung der Klasse **UhrMitDatum** von der Klasse **Uhr**

  neue Datenkomponenten: `actDate`
  neue Funktionskomponenten: `setDate(), setClock()`
  geänderte Funktionskomponenten: `tick(), displayClock()`



◇ Ein **Objekt** einer **abgeleiteteten Klasse** kann **immer** auch als – spezielles - **Objekt** der **Basisklasse**(n) betrachtet werden: z.B. ist jede `UhrMitDatum` auch eine `Uhr`.

◇ **Einfache Vererbung** : Ableitung einer Klasse von nur **einer** Basisklasse.

Mehrfachvererbung : Ableitung einer Klasse von **mehreren** Basisklassen (nur C++, nicht von Java, C#, Python)

◇ Durch Vererbung übertragene Methoden (Funktionen) existieren nur einmal (→ ***Code Sharing***). Dies erleichtert Änderungen und Erweiterungen an bestehenden Klassenhierarchien.

1.5 Grundkonzepte der OOP : Polymorphie

- **Polymorphie (*Polymorphism*) :**

  ◇ **Verwendung** des **gleichen Namens** für **unterschiedliche** - aber miteinander verwandte - **Dinge**.
  (griech. : Vielgestaltigkeit)

  ◇ Polymorphie in der **OOP** ermöglicht, dass **verschiedenartige Objekte** (unterschiedlicher aber durch Vererbung miteinander verwandter Klassen) unter einem **gemeinsamen Oberbegriff** (Basisklasse) betrachtet und bearbeitet werden können.(→ **Generalisierung**)

  Beispiel: Sowohl Objekte der Klasse `Uhr` als auch Objekte der Klasse `UhrMitDatum` lassen sich als `Uhr`-Objekte behandeln.

  ◇ In der Basisklasse definierte Funktionen können in abgeleiteten Klassen mit dem **gleichen Namen** und der **gleichen Parameterliste** (d.h. **gleichem Interface)** erneut definiert werden.
  → **Überschreiben** von Funktionen (*Function Overwriting*), **virtuelle Funktionen**.
  Dadurch wird das durch den Aufruf einer deratigen Funktion bewirkte Verhalten eines Objekts in der abgeleiteten Klasse abgeändert.
  Der gleiche (Funktions-)Name kann somit zur Spezifizierung einer Gruppe ähnlicher - aber doch unterschiedlicher – Operationen (Methoden) verwendet werden.
  → Die **gleiche** durch den Funktions-Namen ausgedrückte **Botschaft** kann - an **unterschiedliche Objekte** gerichtet – zum Aufruf **unterschiedlicher Methoden** führen.
  Die in einem konkreten Fall ausgeführte Operation (Methode) hängt von der tatsächlichen Klasse des Objekts ab, an das die Botschaft gerichtet ist (→ "**Ein Interface - mehrere Methoden**").
  Also nicht die Botschaft, d.h. der Aufruf, bestimmt, welche Methode (Funktion) ausgeführt wird, sondern der Empfänger der Botschaft.

  Beispiel : Die Botschaft "`displayClock()`" an ein UhrMitDatum-Objekt gerichtet, bewirkt die Ausgabe der Uhrzeit und des Datums, während sie bei einem Uhr-Objekt nur zur Ausgabe der Uhrzeit führt.

  ⇒ ***Polymorphie** erlaubt durch die Schaffung eines **Standardinterfaces** die **einheitliche Verarbeitung unterschiedlicher Objekte**, die über **gemeinsame Grundfähigkeiten** verfügen. Dadurch wird die **Beherrschung einer größeren Komplexität** ermöglicht.*

  ◇ Beim Aufruf virtueller (überschreibbarer) Methoden wird die **Zuordnung** der tatsächlich aufgerufenen **Methode** zur **Botschaft** (Methodenname) erst zur **Laufzeit** vorgenommen. Dies bezeichnet man als "**späte Bindung**" ("***late binding***").
  Bei einer Zuordnung bereits zur **Compilezeit** spricht man von "**früher Bindung**" ("***early binding***").

  ◇ In einer erweiterten Betrachtungsweise ermöglicht **Polymorphie** auch das – nicht nur in der OOP eingesetzte – **Überladen** von **Funktionen** (Methoden) und **Operatoren** :

  ▻ **Überladen von Funktionen** (*Function Overloading*) :
  Mehrere Funktionen können den **gleichen Namen** besitzen, sofern ihre **Parameterliste** (Signatur) **unterschiedlich** ist.
  Durch den gleichen Namen wird auch hier eine Art Standard-Interface (das sich aber nicht auf die Parameter erstreckt) bereitgestellt, über das sich mehrere unterschiedliche aber meist miteinander

verwandte Methoden aufrufen lassen.
Die speziell angewandte Methode hängt hier von den beim Aufruf übergebenen Daten (Parametern) ab.

- **Überladen von Operatoren** (*Operator Overloading*) :
  Operatoren können zur Anwendung auf **unterschiedliche Datentypen umdefiniert** werden.
  → Definition spezieller **Operatorfunktionen** (in Java nicht möglich)

1.6 Klassenbeziehungen

- **Vererbungsbeziehung**

  ◇ Beziehung zwischen Klassen, deren Komponenten sich teilweise überdecken

  ◇ Eine abgeleitete Klasse erbt die Eigenschaften und Fähigkeiten (Komponenten) der Basisklasse(n).
  "ist"-Beziehung → ein Objekt der abgeleiteten Klasse ist auch ein Objekt der Basisklasse(n)

  ◇ Ordnungsprinzip bei der Spezifikation von Klassen.
  → **Generalisierung / Spezialisierung**

  ◇ Ein Spezialfall der Vererbung ist die **Implementierung** (Basisklasse definiert nur ein Zugriffsinterface)

- **Nutzungsbeziehungen**

  ◇ Unter einer (statischen) Nutzungsbeziehung versteht man eine in einem konkreten Anwendungsbereich geltende Beziehung zwischen Klassen, deren Instanzen voneinander Kenntnis haben und die dadurch miteinander **kommunizieren** können
  → Nutzungsbeziehungen sind notwendig für die **Interaktion von Objekten**

  ◇ **Assoziation**
  - Spezielle Beziehung zwischen Klassen bzw Objekten, bei der die Objekte **unabhängig** voneinander existieren und **lose** miteinander **gekoppelt** sind
    Beispiel: einem Objekt wird die Referenz auf ein anderes Objekt in einer Methode als Parameter übergeben und nur in dieser Methode verwendet
  - **Navigationsrichtung** : legt die Kommunikationsrichtung und die Richtung, in der ein Objekt der einen Klasse ein Objekt der anderen Klasse referieren kann, fest.
    bidirektional (Kommunikation in beiden Richtungen möglich) oder unidirektional
  - **Kardinalität** (*multiplicity*): bezeichnet die mögliche Anzahl der an der Assoziation beteiligten Instanzen einer Klasse.

  ◇ **Aggregation**
  - Spezielle Beziehung zwischen Klassen bzw Objekten, bei der die Objekte der einen Klasse **Bestandteile** (Komponenten) eines oder mehrerer Objekte der anderen Klasse sind.
    → zwischen den Objekten besteht eine **feste Kopplung**
  - **"hat"**-Beziehung bzw **"ist Teil von"**-Beziehung
  - Das "umschließende" Objekt bildet einen Container für das bzw die enthaltene(n) Objekt(e)
  - Aggregation kann als **Spezialfall der Assoziation** aufgefaßt werden.
  - eine Aggregation ist i.a. eine **unidirektionale** Beziehung (Navigation vom umschließenden Objekt zu den Komponenten-Objekten)

  Je nach dem **Grad der Kopplung** unterscheidet man :

  ▻ **einfache Aggregation**
  Das umschließende Objekt und die Komponenten sind **nicht existenzabhängig**
  Eine Komponente kann zusätzlich noch **weiteren** umschließenden Objekten der gleichen oder einer anderen Klasse zugeordnet sein.
  Bei Löschung des umfassenden Objekts bleiben die Komponenten unabhängig davon erhalten.
  Beispiel: Objekt wird vom umfassenden Objekt erzeugt, Referenz darauf wird anderem Objekt übergeben

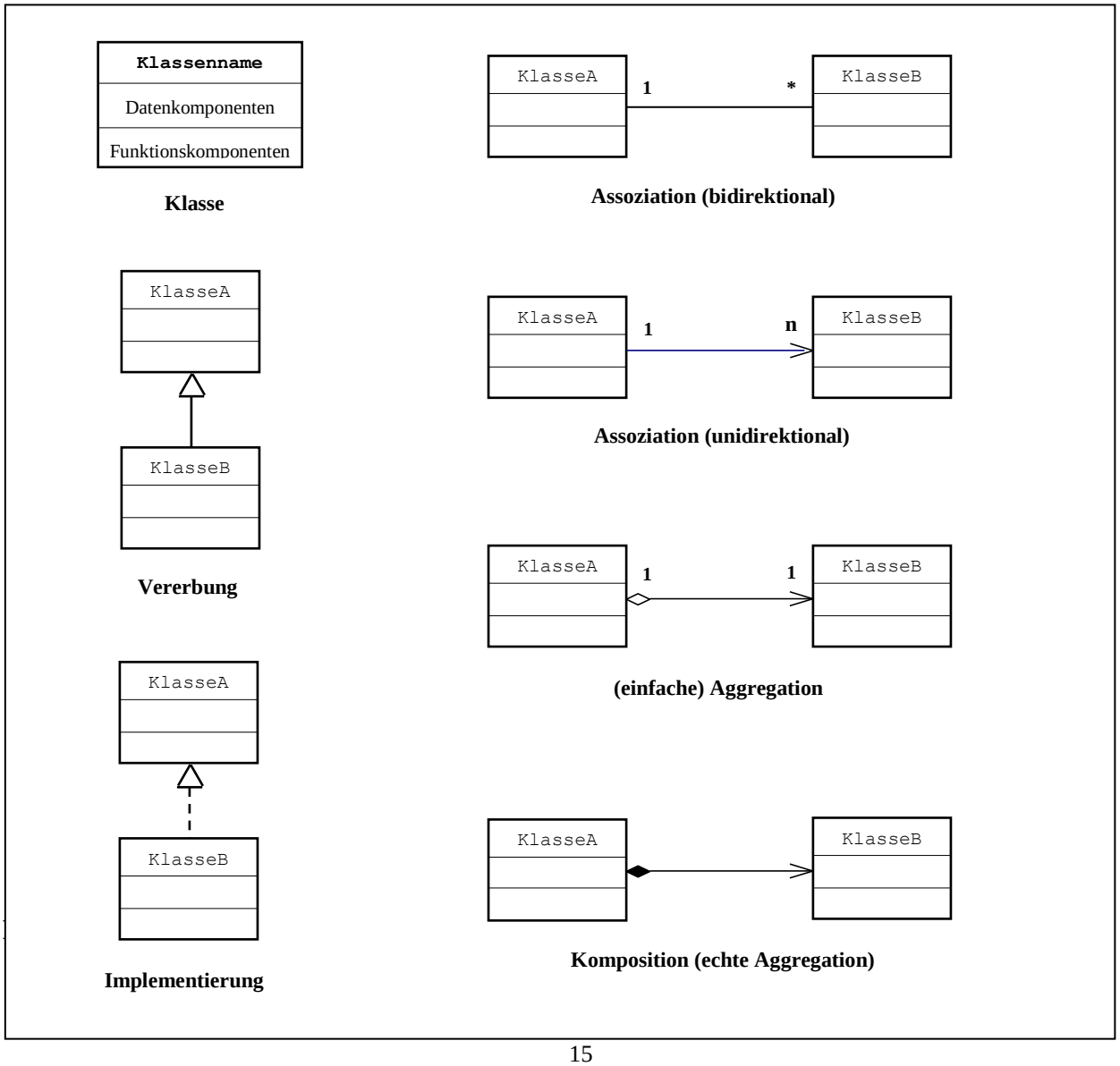
- ▻ **echte Aggregation** (**Komposition**, *composite aggregation)*
  Die Komponenten können **nur einem** umfassenden Objekt zugeordnet sein und nur **innerhalb** diesem **existieren**. Bei Löschung des Aggregats werden auch die Komponenten gelöscht.
  Beispiel: Objekt wird von umfassenden Objekt erzeugt und nur dort verwendet

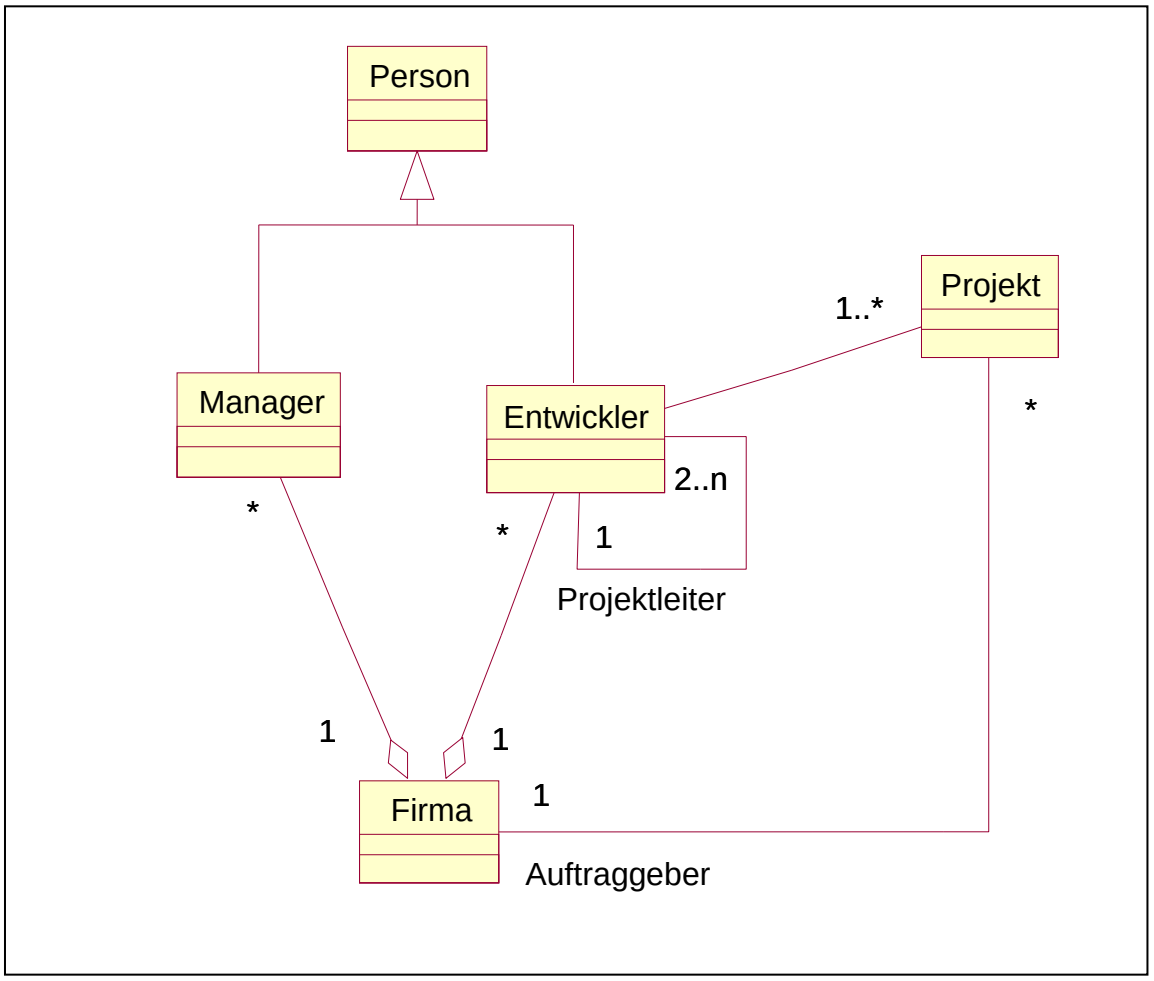
◇ **Anmerkung**:

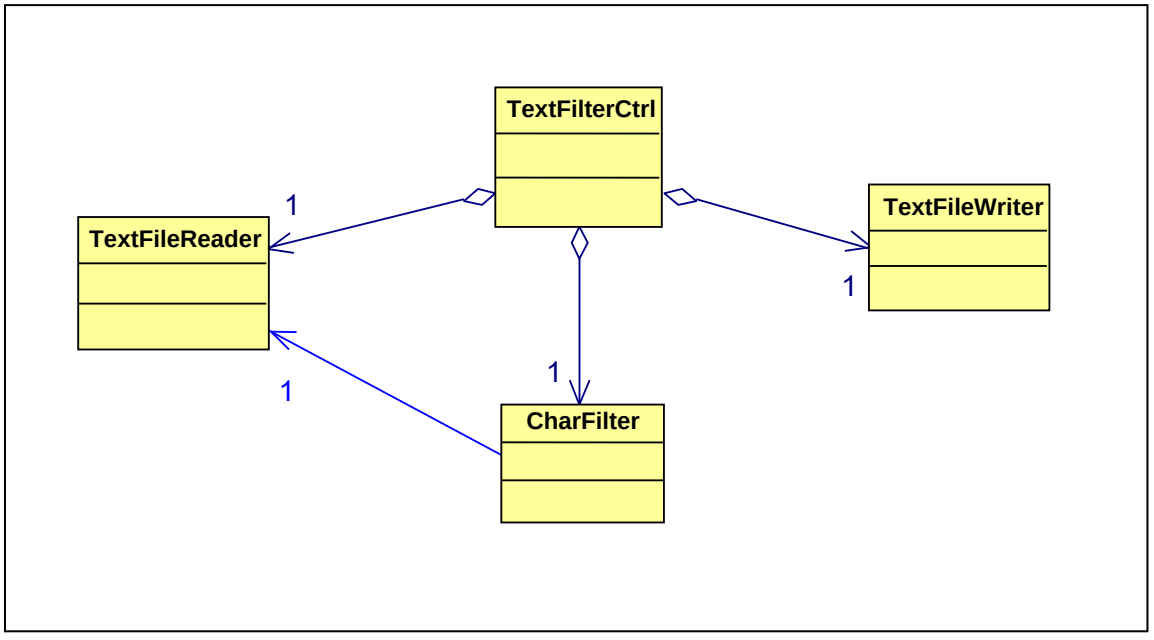
In der Praxis kann es im Einzelfall sehr schwierig sein, zwischen Assoziation und Aggregation und den verschiedenen Formen der Aggregation zu unterscheiden.

Klassenbeziehungen (2)

- **Klassendiagramm**
  - ◇ Die (statischen) **Beziehungen** (Vererbungsbeziehungen und Nutzungsbeziehungen) zwischen den in einem OOP-System zusammenwirkenden **Klassen** lassen sich durch **Klassendiagramme** beschreiben.
  - ◇ Ein Klassendiagramm ist eines der in der **UML** (*Unified Modelling Language*) zusammengefassten graphischen Darstellungsmittel. Es ermöglicht die Erstellung eines **detaillierten statisches Systemmodells**
  - ◇ Zur Erhöhung der Übersichtlichkeit kann die Gesamtheit der Klassen eines Systems auf mehrere **Teildiagramme** aufgeteilt sein
- **Anmerkung :**
  - ◇ Die **dynamischen Beziehungen** zwischen den **Objekten** eines OOP-Systems werden durch **Sequenzdiagramme**, einer anderen Diagrammart der UML, beschrieben. Sie werden hier nicht weiter betrachtet.
- **Elemente eines Klassendiagramms**




Person
Manager
Entwickler
Projekt
1..*
*
2..n
1
Projektleiter
*
*
1
1
1
Firma
Auftraggeber


TextFilterCtrl
TextFileReader
TextFileWriter
1
1
1
1
CharFilter

2 C++ Allgemeines

2.1 Entwicklung der Programmiersprache C++

- C++ ist eine Weiterentwicklung der Programmiersprache C.
  ⇒ C ist - bis auf wenige unbedeutende Ausnahmen - als Untermenge in C++ enthalten.

  Erweiterung von C
  - um Elemente zur Realisierung der Objektorientierten Programmierung(OOP)
  - um zusätzliche nicht-objektorientierte Sprachelemente

  ⇒ C++ ist eine Kombination aus einer prozeduralen und einer objektorientierten Programmiersprache
  ⇒ hybride Programmiersprache.

  Man kann in C++ sowohl prozedural (strukturiert) als auch objektorientiert programmieren.

- Entwicklung der Objektorientierten Programmierung:

  1967 SIMULA, erstmalige Definition des Begriffs "Klasse"
  1974 SMALLTALK, entwickelt von Kay/Goldberg/Ingalls (Palo Alto Research Center): Programmierumgebung bestehend aus Betriebssystem, Sprachinterpreter und graphische Benutzeroberfläche.
  ab 1980 Erweiterung konventioneller Sprachen um objektorientierte Elemente
  z.B. **C++**, CLOS (Common Lisp Object System), Object-PASCAL (Apple),
  1995 **JAVA** (Sun Microsystems)
  PYTHON(Open Source**)**
  2000 **C#** ("C sharp"-Microsoft)

- Entwicklung von C++:
  1980 "C with Classes", entwickelt von Bjarne Stroustrup (Bell Labs von AT&T)
  1983 Änderung des Namens in C++
  1998 ANSI/ISO-Standard für C++

  2006 (Technischer Report 1)

  Enthalten sind im TR1 u. a. reguläre Ausdrücke,[8] verschiedene intelligente Zeiger,[9] ungeordnete assoziative Container,[10] eine Zufallszahlenbibliothek,[11] Hilfsmittel für die C++-Metaprogrammierung, Tupel[12] sowie numerische und mathematische Bibliotheken.[13] Außerdem sind sämtliche Bibliothekserweiterungen der 1999 überarbeiteten Programmiersprache C (C99) in einer an C++ angepassten Form enthalten.[14]

  2011 C++11- Bibliotheken für Nebenläufigkeit (Threads), Mehrprozessorsysteme.
  Zu den weitreichenderen Spracherweiterungen gehört die Typinferenz zur Ableitung von Ergebnistypen aus Ausdrücken und die sogenannten *r-value references*, mit deren Hilfe sich als Ergänzung zu dem bereits vorhandenen *Kopieren* von Objekten dann auch ein *Verschieben* realisieren lässt.

2.2 Unterschiede zwischen C und C++

Vorbemerkungen :
Grundsätzlich enthält C++ die Sprache C als Teilmenge. Allerdings existieren einige wenige Unterschiede zwischen C und den entsprechenden Sprachelementen von C++. Diese können dazu führen, daß ein gültiges C- Programm kein gültiges C++-Programm ist bzw. sich ein C-Programm anders als ein quellcodeidentisches C++-Programm verhält.

Funktionsdeklaration mit leerer Parameterliste

C: stellt eine Funktionsdeklaration alter Art dar, die keinerlei Informationen über die Funktionsparameter enthält. Die Funktion kann beliebige Parameter besitzen.

C++: stellt den Function Prototype einer parameterlosen Funktion dar

```
int f1();    ist äquivalent mit
int f1(void);
```

Funktionen mit einem von void verschiedenen Funktionstyp

C: müssen nicht unbedingt tatsächlich einen Funktionswert zurückgeben

C++: müssen unbedingt immer einen Funktionswert zurückgeben
(Allerdings erzeugen einige Compiler lediglich eine Warnung, wenn die Funktion ohne return-Anweisung beendet wird; eine Beendigung mit einer return-Anweisung ohne Rückgabewert-Ausdruck wird dagegen immer als Fehler betrachtet)

Implizite Konvertierung eines void-Pointers (void *) in einen typgebundenen Pointer

z.B.
```
char *cp;
cp = malloc(20);     /* malloc() erzeugt einen void* */
```

C: ist zulässig

C++: ist unzulässig, explizite Typkonvertierung erforderlich

Unterschiede zwischen C und C++ (2)

Mit einer struct-, union- oder enum-Vereinbarung eingeführte Typnamen

C: dienen nur zusammen mit den Schlüsselworten struct, union bzw. enum als Typbezeichnung.

⇒
```
enum BOOL {FALSE, TRUE};
            enum BOOL lvar;
```

C++: dienen allein bereits als Typbezeichnung

⇒
```
enum BOOL {FALSE, TRUE};
BOOL lvar;/* ist äquivalent mit enum BOOL lvar; */
```

Die Werte eines Aufzählungstyps (= Aufzählungskonstante)

C: sind int-Werte

⇒ Aufzählungstyp-Variablen dürfen int-Werte zugewiesen werden

⇒
```
enum BOOL {FALSE, TRUE};
enum BOOL b;
b=1;                    /* zulässig in C */
sizeof(FALSE);          /* == sizeof(int) */
```

C++: bilden den Wertevorrat eines eigenen Typs, sie werden in Ausdrücken automatisch in int-Werte umgewandelt, umgekehrt müssen int-Werte aber nicht automatisch in Werte eines Aufzählungstyps umgewandelt werden

⇒ Aufzählungstyp-Variablen dürfen prinzipiell keine int-Werte zugewiesen werden, allerdings erzeugen viele Compiler lediglich eine Warnung und nehmen eine automatische Typumwandlung vor

⇒
```
enum BOOL {FALSE, TRUE};
enum BOOL b;
b=1;              /* prinzipiell nicht zulässig in C++ */
sizeof(FALSE);    /* == sizeof(enum BOOL), dieser Wert */
                  /* kann von sizeof(int) abweichen    */
```

Unterschiede zwischen C und C++ (3)

- Lokale Vereinbarungen

  C: nur zulässig zu Beginn eines Blocks (Verbundanweisung bzw Rumpf einer Funktion)

  C++: überall zulässig, können unmittelbar vor dem Ort, an dem das vereinbarte Objekt verwendet wird, erfolgen

```
⇒    end = a+2*b;
     int i;
     for (i=1; i<=end; i++)
     ... ;
```

  Grund : Vereinbarungen sind in C++ Anweisungen
  (⇒ Vereinbarungsanweisung, declaration statement)
  Sogar der Anfangsausdruck im Steuerblock einer for-Anweisung kann durch eine initialisierte Variablendefinition ersetzt werden:

```
⇒    end = a+2*b;
     for (int i=1; i<=end; i++)
       ……;
```

  Die lokale Vereinbarung (für i) gilt bis zum Ende des Blocks, in dem sie erfolgt ist.

  Ausnahme : Eine unter einer Bedingung erfolgte Vereinbarung

  - Eine Vereinbarung im if- oder else-Zweig einer if-Anweisung gilt nur innerhalb des entsprechenden Bedingungszweiges, außerhalb des Zweiges ist sie ungültig.
  - Eine Vereinbarung in einem case-"Zweig" einer switch-Anweisung gilt nur innerhalb der switch-Anweisung (auch in den folgenden case-"Zweigen"), außerhalb der switch-Anweisung ist sie ungültig.

```
⇒    if (end > 0)
       for (int i=1; i<=end; i++)
        ...;
     if (i==end)  /* Fehler!, i außerhalb if-Zweig nicht bekannt*/
          ...;
```

Unterschiede zwischen C und C++ (4)

- const-Variable

  C: sind reine Variable, deren Wert lediglich nicht geändert werden darf, da sie keine Konstanten sind, dürfen sie nicht in konstanten Ausdrücken vorkommen

```
⇒    const int anz = 3;
     char wort[anz];      /* unzulässig in C */
```

  C++: sind gleichzeitig - nicht änderbare - Variable und Konstante, sie besitzen alle Attribute von Variablen (Typ, Wert, Speicherklasse, Adresse), sie dürfen aber auch - wie mit define definierte symbolische Konstante - in konstanten Ausdrücken vorkommen; vorausgesetzt, es handelt sich nicht um dynamisch initialisierte Konstante.

```
⇒    const int anz = 3;
     char wort[anz];            /* zulässig in C++ */

⇒    void g(int n) {
⇒       const int m=2*n; /* dynamisch initialisierte Konstante */
        float xyz[m];    /* auch in C++ ein Fehler ! */
     }
```

- Initialisierung von const-Variablen bei ihrer Definition

  C: nicht unbedingt erforderlich (spätere Wertzuweisung zulässig !)

  C++: unbedingt erforderlich

3 Nicht OOP - orientierte Erweiterungen in C++

3.1 Kommentare in C++

- In C++ existieren zwei Möglichkeiten zur Darstellung von Kommentaren :
  Der Standard-C-Kommentar :
  Jede zwischen den Zeichenkombinationen /* und */ eingeschlossene Zeichenfolge. Ein derartiger Kommentar kann sich über mehrere Zeilen erstrecken. Solche Kommentare dürfen nicht geschachtelt werden.
  Ein spezieller C++-Einzeilen-Kommentar :
  Er wird durch die Zeichenkombination //, die an beliebiger Position in einer Zeile stehen kann, eingeleitet. Jede danach kommende Zeichenfolge bis zum Zeilenende wird vom Compiler ignoriert.

- Beipiel :

```
/* Dies ist ein Programm, in dessen Quellcode beide in C++ möglichen
  Kommentar-Arten vorkommen */

#include <stdio.h>        // Einbinden Header-Datei
int main()  {             // Function Header
   printf("\nEs gibt in C++ zwei Kommentar-Arten\n");
   return 0;
}                         // Ende des Programms
```

  - Es ist zulässig, einen C++-Einzeilen-Kommentar innerhalb eines Standard-C-Kommentars zu schachteln:

```
/* Dies ist ein mehrzeiliger Standard-C-Kommentar,
   der einen // C++-Einzeilenkommentar
   enthält  */
```

- Üblicherweise werden in C++-Programmen längere - sich über mehrere Zeilen erstreckende - Kommentare als Standard-C-Kommentare realisiert, während für kurze Kommentierungen der C++-Einzeilen-Kommentar verwendet wird.

3.2 Einfache Konsolen-Ein-/Ausgabe mittels I/O-Operatoren in C++

- Neben der Möglichkeit der Verwendung der ANSI-C-Standardfunktionen wie scanf() und printf() bietet die Standardbibliothek von C++ eine weitere Methode zur Konsolen-Ein/Ausgabe an.

  Diese zweite Methode arbeitet mit I/O-Operatoren und nutzt - auch wenn es in der einfachen Anwendung nicht direkt zum Ausdruck kommt - Konzepte der objektorientierten Programmierung, wie z.B. die Bildung von Klassen und das Überladen von Operatoren, aus.

- Ähnlich wie bei der ANSI-C-Methode erfolgt die Ein-/Ausgabe von/zu "Streams", die allerdings als Objekte innerhalb einer Klassenhierarchie definiert werden.
  U.a. gibt es die - in der C++-Headerdatei <iostream > deklarierten - vordefinierten Streams

  - **cout** für die Konsolenausgabe (entspricht stdout) und
  - **cin** für die Konsoleneingabe (entspricht stdin)

- Als I/O-Operatoren dienen

  - der Übernahmeoperator >> für die Eingabe von einem Stream und
  - der Übergabeoperator << für die Ausgabe in einen Stream.

  Diese Operatoren entstehen durch Überladen des Rechtsschiebe- bzw Linksschiebe-Operators (deklariert in der Headerdatei <iostream >).

- $\Rightarrow$ die Anwendung der alternativen C++-Methode zur Konsolen-Ein-/Ausgabe erfordert das Einbinden der Headerdatei
  <iostream >

  Um diese neuen I/O-Stream Klassen verwenden zu können, ist zusätzlich die Angabe des zugehörigen Namensraums nowendig. Die geschieht mit der Direktive:

```
using namespace std;
```

Einfache Konsolen-Ein-/Ausgabe mittels I/O-Operatoren in C++ (2)

- Eingabe von der Konsole mittels des Ausdrucks :
  ```
  cin  >>  variable
  ```
  variable kann eine beliebige Variable eines der vordefinierten arithmetischen Datentypen oder eine char-Array-(String-)Variable sein.

  Beispiel :
  ```
  int i;
  cin >> i;          /* entspricht : scanf("%d", &i); */
  ```

  Da obiger Ausdruck als Wert eine Referenz auf cin liefert und der Operator >> links⇒rechts-assoziativ ist, lassen sich mit einem Ausdruck auch die Werte für mehrere Variable einlesen:

  ```
  cin >> var1 >> var2 ... >> varn
  ```

  Beispiel:
  ```
  int i;
  char wort[80];
  cin >> i >> wort; /* entspricht scanf("%d%s", &i, wort); */
  ```

  Wie bei scanf() dienen Whitespace-Character als Trennzeichen zwischen den Eingabefeldern.
  ⇒ In eine char-Variable kann kein Blank, Newline oder Tab eingelesen werden.

- Ausgabe an die Konsole mittels des Ausdrucks :
  ```
  cout << ausdruck
  ```
  ausdruck kann jeder beliebige gültiger C++-Ausdruck sein

  Beispiel :
  ```
  cout << "Hello World !\n"; /* printf("Hello World !\n"); */
  ```

  Da obiger Ausdruck als Wert eine Referenz auf cout liefert und der Operator <<
  Links ⇒ rechts-assoziativ ist, lassen sich mit einem Ausdruck auch mehrere Werte ausgeben:
  ```
  cout << ausdr1 << ausdr2 ... << ausdrn
  ```

  Beispiel:
  ```
  int i = 5;
  char *msg = "Wert von i : ";
  cout << msg << i << '\n';  /* printf("%s%d\n", msg, i); */
  ```

- Wo erforderlich wird der von einem Ein- bzw Ausgabe-Ausdruck zurückgelieferte Wert implizit in einen void*- bzw int-Wert (Status) umgewandelt. Dieser Wert ist NULL (bzw 0), wenn das Dateiende erreicht ist bzw. ein Fehler aufgetreten ist.
  Dies erlaubt Überprüfungen, wie z.B.:

  ```
  while (cin >> wert) { /* ¨¨¨ */ }
  ```

- Formatierung der Ein-/Ausgabe:
  Formatierung ist möglich, aber etwas umständlicher zu handhaben als bei scanf() und printf(), ihr Verständnis setzt Kenntnisse der OOP voraus, daher wird an dieser Stelle noch nicht weiter auf sie eingegangen.

```
//Einfache Ein-/Ausgabe in C++
//k2_cin_cout.cpp

#include <iostream>
using namespace std;

int main()
{
    int x=3;
    double d=3.10;

    //Ausgabe
    cout << "x=" << x << '\n';
    cout << "d=" << d << '\n';


    //Einfache Eingabe
    cout << "Bitte geben Sie eine Ganzzahl ein\n";
    cin >> x;
    cout << "x=" << x << '\n';

    return 0;
}
```

3.3 Referenzen in C++

- Eine Referenz ist ein alternativer Name für eine vorhandene Variable (ein vorhandenes Objekt). Manchmal spricht man von einer "Referenzvariablen", obwohl eine Referenz selbst keine Variable ist, sondern auf eine Variable verweist.

- Vereinbarung (Deklaration) von Referenzen :
  Verwendung des &-Operators
  Die Initialisierung mit dem zu referierenden Objekt ist notwendig.
  ⇒Als Initialisierer ist ein sogenannter Lvalue anzugeben.

  Anmerkung: Ein Lvalue ist ein Ausdruck, mit dem ein Objekt angesprochen wird (der also auf der linken Seite einer Wertzuweisung stehen kann).
  Dies bedeutet auch, dass eine Referenz weder auf ein konstante noch auf ein temporäre Ausdrücke verweisen kann!

  Syntax :

  Beispiele :

```
int a;
int& r = a;          // r und a sind Namen für dasselbe int-Objekt
int& s = r;          // s ist ein weiterer Name für dieses int-Objekt
char cfeld[] = "Hallo World !";
char (&fr)[] = cfeld;        // fr ist ein weiterer Name für cfeld
                             // Klammerung notwendig wegen Priorität
char& f12 = cfeld[12];       // f12 ist ein Name für cfeld[12]
char* cp = fr;
char*& rp = cp;      // cp und rp sind Namen für denselben char-
                    // Pointer, der auf den Anfang des char-Arrays
                     // cfeld zeigt
```

Ausnahme von der Notwendigkeit der Initialisierung : Extern-Deklaration einer Referenz
Beispiel : extern int& r2;

Bei einer Referenz auf eine Konstante darf der Initialisierer auch ein Ausdruck sein, der kein Lvalue ist (konstanter bzw. erst zur Laufzeit auswertbarer Ausdruck). In einem derartigen Fall wird ein temporäres Objekt erzeugt und mit dem Wert des Initialisierers initialisiert. Die Referenz wird zum - dann einzigen - Namen dieses Objekts.

Beispiel:

```
const double& rd = 1.5;          // rd ist Name für temporäres Objekt
void examples(int p) {
    const int& ir = 2*p;         // ir ist Name für temporäres Objekt
    // ……
}
```

Referenzen in C++ (2)

- Eigenschaften von Referenzen

  - Die Vereinbarung einer Referenz erzeugt kein neues Objekt sondern legt eine alternative Bezeichnung für ein existierendes Objekt fest.

  - Eine Referenz kann während der Lebensdauer des referierten Objekts nicht geändert werden.

  - Eine Referenz weist zwar gewisse Ähnlichkeiten zu einem Pointer auf (Verweis auf Objekt im Speicher), sie ist aber kein Pointer. Sie wird auch nicht wie ein Pointer verwendet:
    - eine Referenz belegt keinen Arbeitsspeicher ⇒ von einer Referenz läßt sich keine Adresse ermitteln; d.h. es gibt keinen "Zeiger auf Referenz"
    - es gibt keine "Referenz-Arithmetik"
    - es gibt keine Objekt-Bildung einer Referenz mit dem *-Operator
    - man kann keine Arrays von Referenzen bilden

  - Eine Referenz wird vielmehr genauso verwendet wie die Variable, auf die sie verweist.

    Beispiele zu Referenzen:

```
//k2_ref_main.cpp
#include <iostream>
 using namespace std;

int main()
{
     int a=1;
     int& r = a;         // r und a sind Namen für dasselbe int-Objekt

     cout << "a : " << a << "   r : " << r <<  '\n';
     r++; // a wird incrementiert
     cout << "a : " << a << "   r : " << r << '\n';

     return 0;
}
```

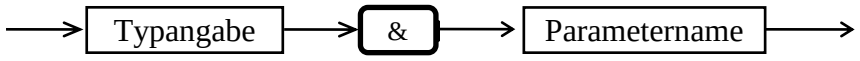
Ausgaben:

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

Referenzparameter in C++ (3)

WICHTIG: Parameterübergabe an Funktionen ist die Hauptanwendung von Referenzen in C++.

- Deklaration von Referenzparametern :
  - Syntax :



  - Beispiele :

```
int f1(int& i);       // i ist eine Referenz auf eine int-Variable
int f2(double& r);    // r ist eine Referenz auf eine double-Variable
```

- Wirkung :
  - Bei einem formalen Referenzparameter wird nicht der Wert sondern - automatisch - die Adresse des korrespondierenden aktuellen Parameters übergeben (Die Anwendung des Adreßoperators auf den aktuellen Parameter ist aber nicht nur nicht notwendig, sondern sogar ein Fehler).
  - Jeder Zugriff zum formalen Parameter bedeutet einen Zugriff zum aktuellen Parameter.
  - Es wird keine lokale Kopie des Werts des aktuellen Parameters angelegt.

    $\Rightarrow$ der aktuelle Parameter kann durch die Funktion direkt verändert werden.
- Beispiel :

  // Funktion zum Tausch zweier Integerwerte

```
//k2_swap.cpp
#include <iostream>
using namespace std;

void swap(int& x, int& y) {
   int hilf;
   hilf=x;
   x=y;
   y=hilf;
}

int main() {
      int i=5;
      int j=8;
      swap(i,j);
      cout << "i : " << i << '\n';
      cout << "j : " << j << '\n';
      return 0;
}
```

- Anmerkung :
  - Referenzparameter können auch dann sinnvoll sein, wenn ihr Wert nicht geändert werden soll $\Rightarrow$ vor allem bei größeren Objekten ist die Übergabe einer Adresse effizienter als die Übergabe des Wertes.
  - In einem derartigen Fall sollte der Parameter als Referenz auf eine Konstante deklariert werden.
    z.B.: `int f1(const int& i);`
    Einem derartigen Parameter darf aktuell auch ein Nicht-Lvalue (Wert eines Ausdrucks ohne Adresse) zugewiesen werden. Für diesen wird eine temporäre Variable erzeugt, deren Adresse übergeben wird. Für Variable-Referenzparameter ist dies nicht zulässig.

3.4 Der Scope (Resolution) Operator in C++

- Ein globales Objekt (Variable oder Funktion), das denselben Namen wie ein lokales Objekt trägt, ist während der Gültigkeit des lokal definierten Namens verborgen (name hiding), d.h. zu dem globalen Objekt kann in C nicht unter seinem Namen zugegriffen werden.
  Beispiel:

```
#include <iostream >
using namespace std;
int x = 10;  // globale Variable x
int main() {
   int x = 20;                                    // lokale Variable x
   x++;                                           // Veränderung lokales x
   cout  << "x : " << x << '\n';                  // Ausgabe lokales x (=21)
   return 0;
}
```

- In C++ hebt in einem derartigen Fall der Scope (Resolution) Operator (Geltungsbereichsoperator, Bereichsoperator):: das Verborgensein des global vereinbarten Objektes auf.
  Er ermöglicht damit einen Namens-Zugriff zu dem - eigentlich verborgenen - globalen Objekt.
  Beispiel:

```
#include <iostream>
using namespace std;
int x = 10;  // globale Variable x
int main() {
   int x = 20;                                    // lokale Variable x
   x++;                                           // Veränderung lokales x
   ::x++;                                         // Veränderung globales x
   cout  << "x lok : " << x << '\n';               // Ausgabe lokales x (=21)
   cout  << "x glo : " << ::x << '\n';             // Ausgabe globales x (=11)
   return 0;
}
```

- Der Scope Resolution Operator läßt sich auf alle Namen, nicht nur auf Namen für Variable sondern z.B. auch auf Typnamen und Namen für Funktionen, anwenden.

- Der Scope Resolution Operator läßt sich nur auf verborgene globale Namen, nicht jedoch auf verborgene lokale Namen, die in einem übergeordneten Block definiert sind, anwenden.

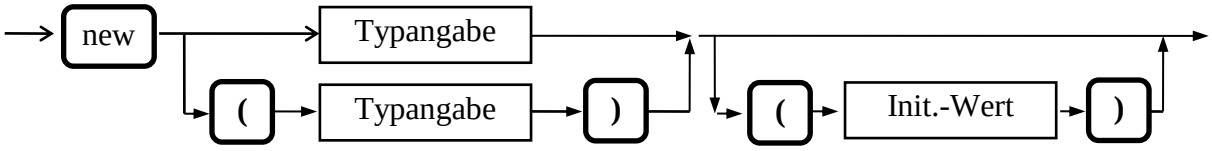
- Der Scope Resolution Operator hat die höchste Priorität. Seine Prioritätsebene liegt noch über der höchsten Priorität der C-Operatoren.

- Es existiert auch eine zweistellige Form des Scope (Resolution) Operators zur Bildung von vollqualifizierten Namen von Elementen aus Namensbereichen und Klassen (s. später).

3.5 Der Operator new in C++

- Der unäre Operator new dient zur dynamischen Speicherallokation.
  Er stellt eine Alternative zur Anwendung der - auch in der Standardbibliothek von C++ enthaltenen - Funktionen malloc() und calloc() dar. ⇒ Objekt-Erzeugungs-Operator

- Syntax :



Beispiele :

```
float* fp = new float;
struct listel {
   double wert;
   listel* next;
};

listel* neu_element(double w) {
   listel* neu;
   neu = new listel;
   if (neu != NULL) {
      neu->wert=w;
      neu->next=NULL;
   }
   return neu;
}
```

- Wirkung :
  - Versuch der Allokation von Speicher für ein Objekt des angegebenen Typs.
  - Kann der Speicher allokiert werden, so wird ein Pointer auf den Beginn des allokierten Bereichs erzeugt.
  - Kann der Speicher wegen Speichermangel nicht allokiert werden, so wird - implementierungsabhängig - entweder der Null-Pointer erzeugt oder die Exception bad_alloc ausgeworfen.
  - Der von einem new-Ausdruck gelieferte Pointer hat bereits den richtigen - der Typangabe entsprechenden - Objekttyp. Eine bei malloc() in C++ erforderliche explizite Typkonvertierung ist daher nicht erforderlich.
  - Bei Array-Typen wird ein Pointer auf das erste Element des Arrays erzeugt. Das Pointer-Objekt ist vom Typ des Array-Objekts :

```
new int[10]      ist vom Typ int *
```

Wie bei mittels malloc() allokierten Speicherbereichen existiert ein mittels new erzeugtes Objekt - unabhängig von Blockgrenzen - bis es explizit zerstört wird (⇒ Speicherfreigabe mittels Operator delete).

Der Operator new in C++ (2)

- Ein mittels new erzeugtes Objekt kann initialisiert werden:
  Angabe des Initialisierungswertes als allgemeiner Ausdruck in runden Klammern nach der Typangabe.

- Ohne Angabe von Initialisierungswerten hat das erzeugte Objekt einen undefinierten Wert.

- Arrays können nicht initialisiert werden.

Beispiele :

```
1) double* dp;
    dp=new double(1.0);        // Initialisierung mit 1.0
```

2.Beispiel

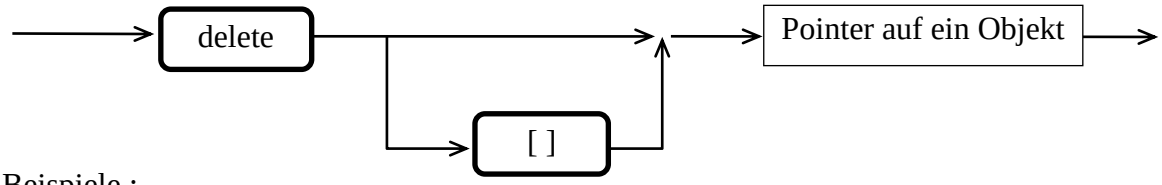
```
#include <iostream>
using namespace std;

struct complex {
   double re;
   double im;
};

complex* newcomplex(complex init) {
   complex* complptr;
   complptr=new complex (init);  // Initialisierung mit Parameter
   if (complptr == NULL)
         cout << "\nAllokationsfehler !\n";
   return complptr;
}

int main() {
   complex* comp;
   complex in = {1.5, -2.5};
   comp=new complex(in);
   // ……
}
```

3.6 Der Operator delete in C++

- Der unäre Operator delete dient zur Freigabe von Speicher, der mittels **new** dynamisch allokiert worden ist.
  ⇒ Objekt-Zerstörungs-Operator

  Wie new eine Alternative zur Anwendung der Funktionen malloc() bzw calloc() darstellt, ist delete die entsprechende Alternative zur Anwendung der Funktion free().

- Syntax :



Beispiele :

```
delete neu;
delete complptr;
delete []cp;
```

- Wirkung :
  Das durch den Operanden von delete referierte Objekt wird zerstört, d.h. der von dem Objekt belegte Speicher wird freigegeben. Wenn der Operand nicht ein mittels new erzeugter Pointer ist, ist die Wirkung undefiniert. Dieser Fall kann zum Programmabsturz führen.
  Ausnahme: Operand ist der NULL-Pointer. In diesem - zulässigen - Fall zeigt der Operator delete keinerlei Wirkung.

- Die Form delete [ ] pointer ist notwendig, wenn der Speicher für ein dynamisch allokiertes Array freigegeben werden soll.

- Ein delete-Ausdruck ist vom Typ void, d.h. ein derartiger Ausdruck erzeugt keinen Wert. Er darf deswegen nur in der Form einer Ausdrucksanweisung angewendet werden.

- Delete auf NULL-Pointer ist unschädlich.

- Delete auf undefinierten Pointer führt zum Absturz.

3.7 Default-Parameter in C++

- In C++ kann man Default-Werte für Funktions-Parameter festlegen. Dies erlaubt es Funktionen mit weniger als den bei ihrer Definition festgelegten Parametern aufzurufen. Für die beim Funktionsaufruf nicht angegebenen aktuellen Parameter werden dann die festgelegten Default-Werte übergeben.

- Ein Parameter-Default-Wert wird - analog zur Initialisierung von Variablen - durch einen Initialisierungsausdruck in **der Parameter-Deklaration** festgelegt. Er ist in der Funktions-Deklaration (Function Prototype) bzw. in der Funktions-Definition, falls im Modul keine Deklaration der Funktion enthalten ist, anzugeben.
  Beispiel : `void func(double r, int a=0, int b=1);`
  Diese Funktion darf mit drei, zwei oder einem Parameter aufgerufen werden. Folgende Funktionsaufrufe sind daher z.B. zulässig :

```
func(3.14, 3, -8);
func(1.25, 6);          // entspricht : func(1.25, 6, 1);
func(-2.5);             // entspricht : func(-2.5, 0, 1);
```

- Es dürfen nur für in der Parameterliste am Ende stehende Parameter Default-Werte festgelegt werden, d.h. nach Parametern mit DefaultWerten dürfen weitere Parameter, die keine Default-Werte erhalten, nicht mehr angegeben werden. Wenn der erste Parameter einen Default-Wert besitzt, müssen daher auch für alle folgenden Parameter Default-Werte festgelegt werden.
  Beispiele :

```
int fwrong(double, char * =NULL, int);    // Fehler!
char *fbad(int i=0, char *satz);          // Fehler!
void fok(int=0, int=0);                   // richtig!
```

- Einmal festgelegte Default-Parameter-Werte können in einer weiteren Funktionsvereinbarung (Deklaration oder Definition) nicht umdefiniert werden. Auch eine Wiederholung der gleichen Festlegung ist nicht zulässig.

  Beispiele :

```
void pr(int val, int base=10);
// ......
void pr(int val, int base=16);          // Fehler !
// ......
void pr(int val, int base=10);          // Fehler !
```

  Zulässig ist dagegen die Festlegung von Default-Parametern in einer weiteren Funktionsvereinbarung, wenn diese in vorhergehenden Vereinbarungen derselben Funktion noch nicht enthalten sind.
  Beispiel :

```
void dr(char *, FILE *);
// ......
void dr(char *, FILE * = stdout);     // zulässig
```

Default-Parameter in C++ (2)

- Der Initialisierungsausdruck zur Festlegung eines Parameter-DefaultWertes darf keine lokalen Variablen und keine anderen Parameter enthalten (nur Konstante und globale Variable). Er wird in der Umgebung der Funktionsdeklaration festgelegt und bei jedem Funktionsaufruf ausgewertet. Beispiele:

```
int i;
void f() {
  int i;
  extern void g(int x=i);    // Fehler, lokale Variable !
  // ¨¨¨¨
}

int a;
int fehl(int a, int b=a);         // Fehler, Parameter !
int b=1;
int d(int);
int g(int x=2*d(b));                   // richtig, globales b !
void h() {
  b=2;
  {
     int b=3;
     g();                          // g(2*d(::b)) ⇒ g(2*d(2)) !
  }
}
```

- Beispiel für Anwendung von Default-Parametern :
  Verwendung des Default-Parameters als Flag zur Kennzeichnung eines Sonderfalls.
  // Hier : Funktion zur Ermittlung des Flächeninhalts eines Rechtecks
  // Sonderfall : Quadrat (⇒ Parameter breite = 0)

```
//k2_default
#include <iostream >
using namespace std;
double flaeche(double laenge, double breite = 0) {
   if (!breite)
     breite=laenge;

   return laenge*breite;
}
int main() {
   cout << "Flaeche Rechteck 10.0*3.5: " << flaeche(10.0, 3.5) << '\n';
   cout << "Flaeche Quadrat    5.0*5.0: " << flaeche(5.0) << '\n';
   return 0;
}
```

```
Bildschirmausgabe:
    Flaeche Rechteck 10.0*3.5: 35
    Flaeche Quadrat  5.0*5.0: 25
```

3.8 Inline-Funktionen in C++

- Einer Funktionsdefinition in C++ kann der Funktions-Spezifizierer (function specifier) inline vorangestellt werden. ⇒ Inline-Funktion
  Hierdurch wird der Compiler "gebeten", einen Funktionsaufruf durch den Code des Funktionsrumpfes zu ersetzen, d.h. die Funktion wie ein Makro zu expandieren. Der Compiler kann der Bitte nachkommen oder sie ignorieren. Sinnvoll ist die Inline-Expandierung i.a. nur bei sehr kurzen Funktionen, bei denen der Aufruf- und Rückkehr-Mechanismus im Vergleich zum eigentlichen Funktionscode sehr aufwendig wäre. Ab einer gewissen Größe und Komplexität wird der Compiler eine Inline-Funktion daher wie eine "normale" Funktion behandeln und nicht wie ein Makro expandieren.
  Sehr kleine Funktionen können von Compilern auch ohne explizites „inline“ inline realisiert werden.
  Die Details werden über Compiler-Flags gereglt, die vom Benutzer einzustellen sind.
- Beispiele :

```
inline int max(int x, int y) { return x>y ? x : y; }
inline void bitset(int& x, unsigned bit) { x = x | (1 << bit) ; }
inline void swap(int& x, int& y) { int h=x; x=y; y=h; }
```

  Aus einem Aufruf:

```
m=max(a,b);
```

  wird dann tatsächlich:

```
m= a>b ? a : b;
```

- Im Gegensatz zu "normalen" Funktionen muß vor dem Aufruf einer Inline-Funktion ihre Definition erfolgt sein. Wenn der Compiler nur ihre Deklaration kennt, kann er wegen fehlender Kenntnis ihres Codes keine Expandierung vornehmen. Dies bedeutet gleichzeitig, daß Inline-Funktionen nicht in andere Module exportiert werden können, also nicht die Speicherklasse extern haben können.

- Inline-Funktionen können die vom Preprozessor bearbeiteten parameterisierten Makros ersetzen. Sie sind diesen vorzuziehen, da sie in der Anwendung wesentlich weniger fehleranfällig sind (Typprüfung der Parameter, keine falschen Ausdrücke wegen fehlender Klammern usw.).

3.9 Überladen von Funktionen in C++

- C++ ermöglicht es, im gleichen Gültigkeitsbereich mehrere Funktionen gleichen Namens aber unterschiedlicher Parameterliste zu vereinbaren. ⇒ Überladen von Funktionen (Function Overloading).
  Bei einem Funktionsaufruf wählt der Compiler an Hand der aktuellen Parameter die jeweils richtige Funktion aus. Die Anzahl, Reihenfolge und Typen der Funktionsparameter bilden die Signatur einer Funktion. Die Signatur dient zusammen mit dem Funktionsnamen zur Identifikation einer Funktion. Entsprechend sind in den vom Compiler erzeugten Symboltabellen codierte Funktionsbezeichner, die zusätzlich zum eigentlichen Quellcode-Funktionsnamen eindeutige Angaben über die Parameter enthalten, abgelegt.

- Die unterschiedlichen Funktionen gleichen Namens müssen sich in den Typen oder/und der Anzahl der Parameter unterscheiden. Zusätzlich können sich die Funktionen auch im Funktionstyp (Typ des Rückgabewertes) unterscheiden. Ein unterschiedlicher Funktionstyp allein reicht nicht zur Unterscheidung der Funktionen aus und ermöglicht somit kein Überladen von Funktionen.

- Praktisch wird das Überladen von Funktionen angewendet für Funktionen, die gleiche oder ähnliche Aufgaben nur mit unterschiedlichen Parametertypen bzw -anzahl realisieren. Dadurch läßt sich die Komplexität eines Quellprogramms verringern und damit seine Klarheit und Übersichtlichkeit erhöhen. (Tatsächlich zeigt sich im Überladen von Funktionen bereits eines der Grundkonzepte der OOP ⇒ Polymorphie).

```
//Beispiel: k2_overload

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

void swap(double& a, double& b)  {      // swap(double, double)
  double h=a;
  a=b;
  b=h;
}

int main() {
   int i=5, j=13;
   double x=4.5, y=-12.9;

   swap(i,j);                  // Aufruf von swap(int, int)
   swap(x,y);                  // Aufruf von swap(double, double)
   // ....
   return 0;
}
```

Überladen von Funktionen in C++ (2)

- Der Compiler muß in der Lage sein, bei einem Funktionsaufruf eine eindeutige Auswahl unter den überladenen Funktionen zu treffen :

  * Die Funktionen, die überladen werden sollen, müssen sich in den Parametertypen "ausreichend" unterscheiden.
    Z.B. können für Parameter vom Typ T und Referenz auf Typ T (T&) jeweils Variable vom Typ T als aktuelle Parameter verwendet werden ⇒ keine ausreichende Unterscheidung

```
⇒    int f(int i);
     int f(int& ri);                    // Fehler !
```

  * Aufzählungstypen sind in C++ eigenständige Typen, die sich untereinander und von int unterscheiden
    ⇒ ausreichende Unterscheidung

```
z.B.:  enum BOOL {FALSE, TRUE};
       void f(int i)   { /*  ....  */ }
       void f(BOOL b)  { /*  ....  */ }      // zulässig !
```

  * Die aktuellen Parameter beim Funktionsaufruf müssen eine eindeutige Identifikation der auszuwählenden Funktion zulassen. Falls aktuelle Parameter angegeben werden, die im Typ von den formalen Parametern abweichen, können durch die dann notwendigen impliziten Typkonvertierungen Mehrdeutigkeiten auftreten ⇒ Fehler. Gegebenenfalls müssen explizite Typkonvertierungen angegeben werden. Beispiel:

```
int max(int a, int b);
double max(double a, double b);
int main()  {
    // ....
    cout << max(3, 4.5);           // Fehler ! , mehrdeutig !
    cout << max((double)3, 4.5);   // ok ! , eindeutig !
    // ....
}
```

- Auch ein Überladen von Funktionen mit Default-Parametern ist möglich. Allerdings besteht hier eine besonders große Gefahr von Mehrdeutigkeiten.

```
z.B.   void g(char *str, int x=0);
       void g(char *str, float x=1.0);
       int main()  {
           // ....
           g("hallo");               // Fehler !, mehrdeutig !
           g("hallo", 1);            // ok !, eindeutig !
           // ....
       }
```

Beispiele zum Überladen von Funktionen in C++

```
#include <iostream>
using namespace std;
void date(const char *str);                      // Datum als String
void date(int tag, int monat, int jahr);  // Datum als 3 int-Werte
int main() {
    date("23.8.1996");
    date(23, 8, 1996);
    return 0;
}

void date(const char *str) {
    cout << "\nDatum : " << str << '\n';
}

void date(int tag, int monat, int jahr) {
    cout << "\nDatum : " << tag << '.' << monat ;
    cout << '.' << jahr << '\n';
}
```

```
#include <iostream >
using namespace std;
void f1(int a) {
    cout << "\nParameter a : " << a << '\n';
}

void f1(int a, int b) {
    cout << "\nSumme a + b : " << a+b << '\n';
}

int main() {
    f1(10);
    f1(10, 30);
    return 0;
}
```

```
#include <iostream>
using namespace std;
inline int max(int a, int b) { return a>b ? a : b; }
inline long max(long a, long b) { return a>b ? a : b; }
inline double max(double a, double b) { return a>b ? a : b; }

int main() {
    cout << '\n' << max(3, -5);
    cout << '\n' << max((double)3, 4.5);
    cout << '\n' << max(123000, 5L) << '\n';
    return 0;
}
```

3.10 Datentyp bool in C++

- Im Entwurf für ANSI-C++ wird ein logischer Datentyp eingeführt:

  Datentyp **bool**.

  Er wird als eigenständiger vorzeichenbehafteter Ganzzahl-Typ ("unique signed integral type") definiert, dessen Wertevorrat aus den beiden Werten true und false besteht. In arithmetischen Ausdrücken kann ein bool-Wert automatisch in einen int-Wert umgewandelt werden (integral promotion):

```
false  ⇒ 0
true   ⇒ 1
```

  Ein numerischer Wert oder ein Pointer-Wert kann implizit in einen bool-Wert umgewandelt werden:

```
==0   bzw.   ==NULL        ⇒    false
!=0   bzw.   !=NULL        ⇒    true
```

  Weiterhin ist festgelegt, daß Vergleichsoperatoren einen bool-Wert erzeugen.

- In vielen C++-Compilern ist der Datentyp bool noch nicht implementiert. In derartigen Fällen läßt er sich leicht durch einen selbstdefinierten Aufzählungstyp nachbilden :

```
typedef enum { false, true } bool;
```

oder

```
enum bool { false, true };
```

4 Klassen

Einfaches OOP-Demonstrationsprogramm in C++

```
/* ------------------------------------------------------------------ */
/* Programm Counter                                                   */
/* ------------------------------------------------------------------ */
/* Einfaches OOP-Demonstrationsprogramm in C++                        */
/* ------------------------------------------------------------------ */
/* Klassendefinition, Implementierung und Applikation in einer Datei */

//Typdefinition einer Klasse
#include <iostream>
using namespace std;

class Counter
{
  public:
     //Counter();              //Konstruktor
     Counter(int stand=0);     //Konstruktor mit einem Default- Parameter
     ~Counter();               //Destruktor
                               //Beim Anlegen eines Objektes wird ein
                               // Konstruktor automatisch aufgerufen
     void count();
     void reset();
     void preset(int start_wert);
     void display() const;    //Methode darf nur lesend
                              //auf die Datenelemente zugreifen
  private:
     void ausgabe_stand() const;    //Demo private Hilfsfunktion
     int m_Zaehlerstand;            //privates Datenelement
};
```

```
//Implementierung der Klasse Counter
/*
Counter::Counter()  //Konstruktor
{
   reset();
}*/

Counter::Counter(int stand)
{
   preset(stand);
}

void Counter:: count()
{
   m_Zaehlerstand++;
}

void Counter::reset()
{
   m_Zaehlerstand=0;
}

void Counter::preset(int stand)
{
   m_Zaehlerstand= stand;
}
```

```
void Counter::display() /*const*/
{
   ausgabe_stand();
   //cout << "Zaehlerstand: " << m_Zaehlerstand << endl;
}

void Counter::ausgabe_stand() /*const */
{
   cout << "Zaehlerstand: " << m_Zaehlerstand << endl;
}
/*
Counter::~Counter()
{
   cout << "Im Destruktor von Counter" << endl;
}
*/
```

```
//Applikation
int main()
{
       //Anlegen eines Objektes (Variablen) vom Typ Counter
       Counter c1;      //Typname Objektname

       //Aufruf einer Memberfunktion von c1
       c1.display();
       c1.count();
       c1.display();

       return 0;
}
```

4.1 **Definition von Klassen in C++**

- Eine Klasse (class) ist ein benutzer-definierter Datentyp, der - ähnlich einer Structure in C - aus Komponenten (members) aufgebaut ist.
  Diese Komponenten können sein :
  - Daten (allgemeiner : Objekte) beliebigen Typs und
  - Funktionen zur Manipulation dieser Daten (Objekte)
  Für den Zugriff zu den einzelnen Komponenten können Beschränkungen festgelegt werden.

- Zur Kennzeichnung der Zugriffsbeschränkungen dienen Zugriffs-Specifier (access specifier) :
  private: Zu der betreffenden Komponente dürfen nur Funktionen, die selbst Komponente derselben Klasse sind (member functions), (sowie Freund-Funktionen dieser Klasse) zugreifen. Ein Zugriff von außen ist nicht zulässig.
  protected: Zu der betreffenden Komponente dürfen nur Funktionen zugreifen, die selbst Komponente derselben Klasse oder Komponente einer von dieser Klasse abgeleiteten Klasse sind (sowie Freund-Funktionen dieser Klasse und der von ihr abgeleiteten Klassen).
  public: Zu der betreffenden Komponente kann ohne Beschränkung, also auch von außen, zugegriffen werden.

- Eine Klassen-Definition entspricht im wesentlichen einer StructureVereinbarung in C.
  Unterschiede :
  - Verwendung des Wortsymbols class (statt struct);
  - Neben Vereinbarungen für Datenkomponenten (data members,Variable, allgemeiner : Objekte) können - und werden im allgemeinen - auch Vereinbarungen für Funktionskomponenten (member functions, Methoden) enthalten sein;
  - Angabe von Zugriffsbeschränkungen (access specifier);
  - (Die Definition namenloser Klassen ist zwar zulässig, i.a. aber nicht üblich).

  Funktionskomponenten werden i.a. durch eine Deklaration (Function Prototype) angegeben. Die Definition der entsprechenden Funktionen muß dann außerhalb der Klassendefinition vorgenommen werden.

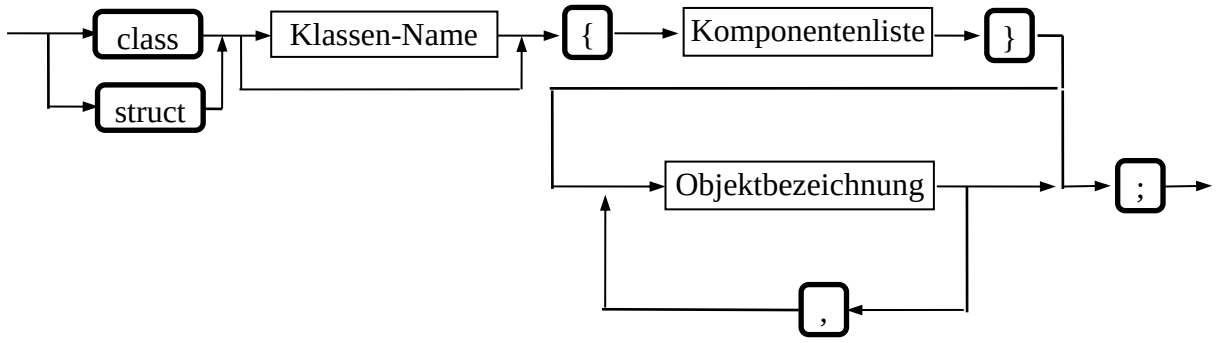
  Es ist auch zulässig statt Funktionsdeklarationen Funktionsdefinitionen anzugeben. Diese Funktionen werden dann vom Compiler als inline-Funktionen übersetzt.

  Es ist sinnvoll und üblich Komponenten gleicher Zugriffsbeschränkung unter einmaliger Voranstellung des entsprechenden „access specifiers“ zu Gruppen zusammenzufassen.
  Die Wirkung eines access specifiers gilt solange bis sie durch einen anderen access specifier aufgehoben wird.
  Defaultmäßig - ohne Angabe eines „access specifiers“ - sind die Komponenten einer mittels class definierten Klasse private.

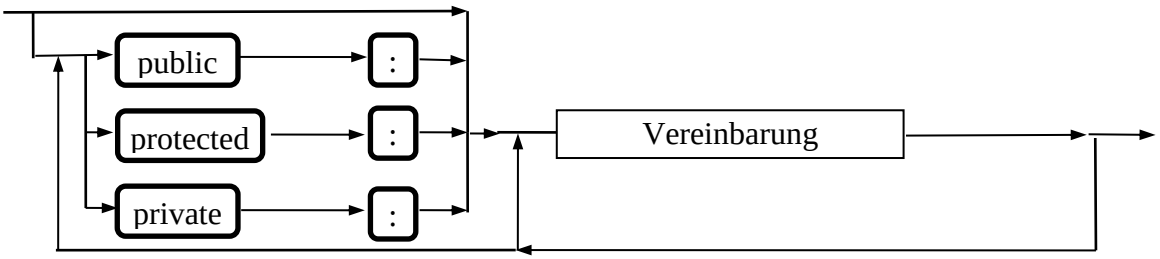
- Statt mit class können Klassen auch mit struct definiert werden. Einziger Unterschied : Defaultmäßig sind die Komponenten public.

Definition von Klassen in C++ (2)

- Syntax (vereinfacht)



Komponentenliste :



- Beispiele :

```
class Counter  {                  // Klasse "Zähler"
                                  // private-Komponenten
     unsigned long act_count;     // aktueller Zählerstand
   public :                       // public-Komponenten
     void reset();                // Rücksetzen des Zählers (auf 0)
     void preset(unsigned long);  // Setzen des Zählers auf def. Wert
     void count();                // Zählen
     unsigned long value();       // Ausgabe aktueller Zählerstand
};

struct CharStack       {          // Klasse "Stack für char"
                                  // public-Komponenten
     void init();                 // Initialisieren des Stack-Pointers
     void push(char);             // Ablage auf Stack
     char pop();                  // Rückholen vom Stack
   private :                      // private-Komponenten
     char cstck[80];              // Speicherbereich des Stacks
     int tos;                     // Index des Top of Stack
};
```

Definition von Klassen in C++ (3)

- Neben Daten- und Funktions-Komponenten-Vereinbarungen kann eine Klassendefinition noch weitere Vereinbarungen enthalten (z.B. Typdefinitionen, Using-Deklarationen, Freund-Deklarationen).

- Die durch enthaltene Typdefinitionen eingeführten Typen bzw. (mittels typdef) Typnamen (eingebettete Typen, nested types) und Aufzählungskonstante (Member-Konstante, member constants) werden ebenfalls als Klassenkomponenten betrachtet.

- Der Verfügbarkeitsbereich (scope) Klasse:
  Jede Klasse legt einen eigenen Verfügbarkeitsbereich (Geltungsbereich, scope) fest. Dieser umfaßt die Klassendefinition selbst sowie alle Memberfunktionen, auch wenn sie "extern" (außerhalb der Klassendefinition) definiert sind.
  Alle innerhalb der Klasse definierten Komponenten können nur innerhalb dieses Klassen-Verfügbarkeitsbereichs eigenständig (mit ihrem Namen allein) verwendet werden. Insbesondere können Memberfunktionen zu allen Klassenkomponenten direkt über den Komponentennamen alleine zugreifen (Ausnahme : Zugriff zu nichtstatischen Komponenten in statischen Memberfunktionen).

  Außerhalb des Klassen-Verfügbarkeitsbereichs kann zu den Klassenkomponenten
  - nur über konkrete Objekte mit dem Element-Operator (.) bzw Objektelement-Operator (->)
  - bzw. über den mit Hilfe des Scope Resolution Operators (::) gebildeten vollqualifizierten Namen zugegriffen werden:

  Voll-qualifizierter Komponentenname : klassenname::komponentenname

Beispiel :

```
class Tuer  {
 public:
     enum Zustand { fehlend, offen, geschlossen };
     // ...
};

void func()  {
 Tuer::Zustand zVar = Tuer::offen;
 // ...
}
```

4.2 **Instanzen von Klassen (Objekte) in C++**

- Ein durch eine Klassen-Definition eingeführter Klassen-Name kann - als Typname - zur Definition von statischen oder dynamischen Instanzen (Exemplaren) dieser Klasse (Variablen!) verwendet werden. Diese Instanzen bezeichnet man üblicherweise als Objekte (im engeren Sinne). Entsprechend dieser Bezeichnungsweise wird auch hier i.a. unter einem Objekt die Instanz einer Klasse verstanden.
  Beispiele :

```
Counter eventzaehler;         //statisches Objekt
CharStack cs1;                //statisches Objekt
Counter* pZaehl;
pZaehl = new Counter;         //dynamisches Objekt  (liegt im Heap)
```

- Realisierung :
  Die Datenkomponenten einer Klasse sind jeweils an ein Objekt der Klasse gebunden (Ausnahme: statische Komponenten). Für sie wird erst durch die Definition eines Objekts Speicher allokiert. Sie existieren also jeweils pro Objekt.
  Die Funktionskomponenten (member functions, Methoden) sind dagegen an die Klasse gebunden. Sie existieren nur einmal pro Klasse.

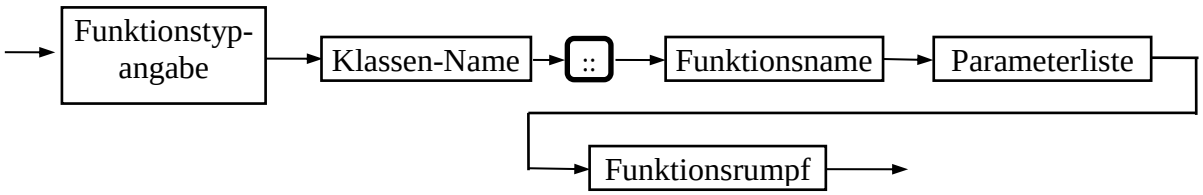
- Grundsätzliche Operationen mit Objekten als Ganzes
  1. Zuweisung eines Objektes an ein anderes Objekt
     Beide Objekte müssen derselben Klasse angehören. Die Zuweisung bewirkt eine bitweise Kopie des Speicherplatzes, den das eine Objekt belegt, in den Speicherplatz des anderen Objekts.
     $\Rightarrow$ Die Datenkomponenten beider Objekte sind nach der Zuweisung identisch.
     Problem: Komponenten, die Pointer auf andere Speicherbereiche sind (u.U. können belegte Speicherbereiche nicht mehr erreichbar sein und damit auch nicht mehr freigegeben werden, während andererseits die Zerstörung der Objekte zur doppelten Freigabe desselben Speichers führen kann).
     Abhilfe : Überladen des Zuweisungs-Operators für diese Klasse

  2. Übergabe eines Objektes als Parameter an eine Funktion
     Es findet Wertübergabe statt. Es wird ein neues - temporäres - Objekt der Klasse erzeugt (der formale Parameter !), das mit dem Wert des aktuellen Parameters initialisiert wird. Auch hier findet eine bitweise Kopie aller Datenkomponenten statt. Bei Beendigung der Funktion wird dieses temporäre Objekt wieder zerstört. Dabei können ähnliche Probleme wie im Fall der Zuweisung auftreten.
     Abhilfe : Selbstdefinierter Copy-Konstruktor

  3. Rückgabe eines Objektes als Funktionswert
     Erzeugung eines temporären Objekts, das mit allen Komponenten des zurückzugebenden Objekts initialisiert wird. Dieses temporäre Objekt wird nach der Verwendung des Rückgabewerts zerstört.
     $\Rightarrow$ ähnliche Probleme und Abhilfe wie im Fall der Parameterübergabe.

4.3 **Member-Funktionen von Klassen in C++**

- Syntax der "externen" Definition von Member-Funktionen :
  Wird eine Member-Funktion (member function) außerhalb der Klassendefinition definiert - was in der Regel der Fall ist -, so muß ein Zusammenhang zwischen der Funktion und der Klasse, zu der sie gehört, angegeben werden. Dies geschieht durch Voranstellen des Klassen-Namens vor den Funktionsnamen unter Verwendung des Scope (Resolution) Operators.



Beispiele:

```
void Counter::preset(unsigned long start) {
     act_count=start;
}

char CharStack::pop() {
     if (tos==0) {
      cout << "Stack ist leer\n";
      return 0;          // Rückgabe von 0 bei leerem Stack
     }
     else  {
       tos--;            // tos zeigt auf den ersten freien Platz
      return cstck[tos];
     }
}
```

- Aufruf von Memberfunktionen:
  Memberfunktionen können nur über ein Objekt, d.h. als Komponente eines Objektes aufgerufen werden. Dies erfolgt - analog zum Zugriff zu Datenkomponenten - unter Voranstellung der Objektbezeichnung und des Operators "." (Element-Operator) bzw. eines Objektpointers und des Operators "-> ".

  Beispiele :

```
     C = cs1.pop();            // cs1 ist Objekt der Klasse CharStack
     pZaehl->preset(1000L);    // pZaehl ist Pointer auf Objekt der Klasse
                               // Counter
```

- Der Aufruf einer Member-Funktion über (für) ein Objekt entspricht der Übermittlung einer Botschaft an das Objekt.

Demonstrationsprogramm zu Klassen u. Member-Funktionen in C++

```
/* ------------------------------------------------------------------  */
/* Header-Datei CharStack.h                                            */
/* ------------------------------------------------------------------  */
/* Definition der Klasse CharStack          */
/* ------------------------------------------------------------------  */
#ifndef CHAR_STACK
#define CHAR_STACK
#define SIZE 10                  // Groesse des Stacks

class CharStack  {               // Klasse "Stack für char"
  public:                        // public-Komponenten
   void init();                  // Initialisieren des Stackpointers
   void push(char);              // Ablage eines Characters auf Stack
   char pop();                   // Rückholen eins Characters vom Stack
  private:                       // private-Komponenten
   char cstck[SIZE];             // Speicherbereich des Stacks
   int tos;                      // Index des Top des Stack (Stackpointer)
};
#endif
```

```
/* ------------------------------------------------------------------  */
/* Quelldatei CharStack.cpp                                            */
/* ------------------------------------------------------------------  */
/* Implementierung der Klasse CharStack      */
/* ------------------------------------------------------------------  */
#include  "CharStack.h"
#include <iostream>

void CharStack::init()  {
  tos=0;                      // tos zeigt immer auf ersten freien Platz
}

void CharStack::push(char c) {
  if (tos==SIZE)  cout << "Stack ist voll !\n";
  else  {
     cstck[tos]=c;
     tos++;
  }
}

char CharStack::pop()  {
  if (tos==0)  {
      cout << "Stack ist leer !\n";
      return 0;
  }
  else {
      tos--;
      return cstck[tos];
  }
}
```

```
/* ------------------------------------------------------------------  */
/* Quelldatei  CStack.cpp                                              */
/* ------------------------------------------------------------------  */
#include  "CharStack.h"
#include <iostream >

int main()  {
  CharStack st;
  char z;
  bool ende=false;

  st.init();
  cout << "\nDemo-Programm char-Stack";
  cout << "\nPush (u)  Pop (o)  Quit (q)\n\n";

  do  {
    cout << "? "; cin >> z;
    switch (z)  {
      case 'u':
      case 'U': cout << "Push Zeichen ? "; cin >> z;
                st.push(z);
                break;
      case 'o':
      case 'O': z=st.pop();
                if (z!=0)
                 cout << "Pop Zeichen  : " << z << '\n';
                break;
      case 'q':
      case 'Q': ende=true;
                break;
    }
  } while(!ende);

  return 0;
}
```

Aufrufbeispiel :

```
Demo-Programm char-Stack
Push (u)  Pop (o)  Quit (q)
? o
Stack ist leer !
? u
Push Zeichen ? A
? u
Push Zeichen ? b
? u
Push Zeichen ? C
? o
Pop Zeichen  : C
? o
Pop Zeichen  : b
? o
Pop Zeichen  : A
? o
Stack ist leer !
?   q
```

Member-Funktionen von Klassen in C++ (2)

- Innerhalb der Member-Funktionen kann auf alle Komponenten des aktuellen Objekts allein mit dem Komponentennamen zugegriffen werden. Das aktuelle Objekt ist dasjenige, über das die Memberfunktion aufgerufen wurde.

- Der Pointer **this**
  Eine Member-Funktion kann auch das Objekt, über das sie aufgerufen wurde, als Ganzes referieren. Dies ist möglich, weil jeder Member-Funktion beim Aufruf automatisch als verborgener Parameter ein Pointer auf das aktuelle Objekt, übergeben wird. Dieser Pointer steht unter dem Namen "this" zur Verfügung (this ist ein reserviertes Wort). Er ist ein konstanter Pointer, der beim Funktionsaufruf initialisiert wird und danach nicht mehr verändert werden kann.

  $\Rightarrow$ implizite Parameter-Deklaration : X* const this; // X sei Klassen-Name

  Beispiel :
  ```
  CharStack st;
  ....
  st.push('A');    // innerhalb von push gilt: this==&st
  ```

  Der Zugriff zu den Komponenten des aktuellen Objekts allein über den Komponentennamen stellt genaugenommen eine abkürzende Schreibweise für den Zugriff unter Verwendung des Pointers this dar.
  Innerhalb von CharStack::push() sind beispielsweise die beiden folgenden Anweisungen äquivalent:

  ```
  tos++;
  this->tos++;
  ```

- Explizite Anwendung des Pointers this:

  1. In Member-Funktionen, die direkt Pointer auf Objekte der eigenen Klasse manipulieren (typisch: Funktionen zur Bearbeitung verketteter Datenstrukturen)

  2. In Member-Funktionen, die Funktionen aufrufen, denen das aktuelle Objekt übergeben wird.

  3. Wenn ein Zeiger oder eine Referenz auf das aktuelle Objekt zurückgegeben wird (return this; bzw. return *this;)

  4. Wenn Komponenten des aktuellen Objekts durch gleichnamige lokale Variablen oder formale Parameter der Member-Funktion verdeckt sind, die Member-Funktion aber zu diesen Komponenten zugreifen soll. (eine derartige Namensgleichheit sollte aber nach Möglichkeit vermieden werden, schlechter Programmierstil !)

Beispiele zur Anwendung des Pointers this in C++

```
class Dlist   {         // Klasse "Listenelement doppelt verkettete Liste"
   public:
     void insert(Dlist* );   // Einfügen Listenelement nach aktuellem Element
     // ¨¨¨                  // weitere Member-Funktionen
   private:
     int inhalt;             // Inhalt des Listenelements
     Dlist* next;            // Pointer auf nächstes Listenelement
     Dlist* back;            // Pointer auf vorheriges Listenelement
 };

 void Dlist::insert(Dlist* neu)  {
   neu->next=next;
   neu->back=this;           // aktuelles Listenelement
   next->back=neu;
   next=neu;
 }
```

```
 class Ratio   {                  // Klasse "Rationale Zahl"
                                  // dargestellt durch Zähler und Nenner
   public:
     Ratio& plusGleich(Ratio&);   // Add. ration. Zahl zu akt. ration. Zahl
     // ¨¨¨                       // weitere öffentliche Member-Funktionen
   private:
     long zaehler;
     long nenner;
     void kuerze(void);           // Zähler und Nenner teilerfremd machen
 };

 Ratio& Ratio::plusGleich(Ratio& x)  {
   zaehler=zaehler*x.nenner + nenner*x.zaehler;
   nenner*=x.nenner;
   kuerze();
   return *this;
 }
```

```
 class Artikel    {                     // Klasse "Verkaufsartikel"
   public:
     void store(char *, long, float);   // Speicherung der Artikeldaten
     // ¨¨¨                             // weitere Member-Funktionen
   private:
     char name[30];
     long nummer;
     float preis;
 };

 void Artikel::store(char *name, long nummer, float preis) {
     strcpy(this->name, name);          // Namensgleichheit ist schlechter
     this->nummer=nummer;               // Programmierstil
     this->preis=preis;
}
```

Member-Funktionen von Klassen in C++ (3)

- Der Zugriff zu Komponenten des aktuellen Objekts, die durch gleichnamige lokale Variablen oder formale Parameter der Member-Funktion verdeckt sind, läßt sich statt mittels des Pointers this auch mittels des Scope (Resolution) Operators erreichen.
  ⇒ Anwendung des voll-qualifizierten Klassen-Komponenten-Namens
  `klassenname::komponentenname`

  Beispiel :

```
class Artikel  {                                  // Klasse "Verkaufsartikel"
 public:
   void store(char*, long, float);                // Speicherung der Artikeldaten
   // ¨¨¨                                         // weitere Member-Funktionen
 private:
   char name[30];
   long nummer;
   float preis;
 };

 void Artikel::store(const char *name, long nummer, float preis) {
   strcpy(Artikel::name, name);                  // Namensgleichheit ist schlechter
 Artikel::nummer=nummer;                         // Programmierstil
   Artikel::preis=preis;
 }
```

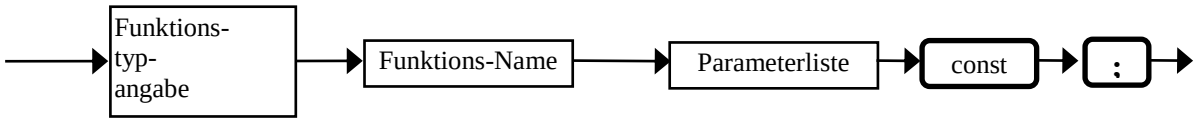
- Überladen von Member-Funktionen
  Member-Funktionen lassen sich - analog zu normalen Funktionen - auch überladen.
  Beispiel :

```
// Klasse "Rationale Zahl" dargestellt durch Zähler und Nenner
class Ratio {
  public:
    Ratio& plusGleich(const Ratio&);  // Add. ration. Zahl zu akt. ration. Zahl
    Ratio& plusGleich(long);    // Add. ganze Zahl zu akt. ration. Zahl
    // ¨¨¨                      // weitere öffentliche Member-Funktionen
  private:
    long zaehler;
    long nenner;
    void kuerze(void);          // Zähler und Nenner teilerfremd machen
};

Ratio& Ratio::plusGleich(const Ratio& x) {
  zaehler=zaehler*x.nenner + nenner*x.zaehler;
  nenner*=x.nenner;
  kuerze();
  return *this;
}

Ratio& Ratio::plusGleich(long x) {
  zaehler+=x*nenner;
  kuerze();
  return *this;
}
```

4.3.1 Konstante Member-Funktionen in C++

- Eine Member-Funktion kann mit dem Zusatz „const“ vereinbart werden. Dadurch wird sichergestellt, daß die Funktion das aktuelle Objekt nicht versehentlich verändern kann. Hierfür ist das Schlüsselwort „const“ sowohl bei der Funktionsdeklaration als auch bei der Funktionsdefinition im Funktionskopf unmittelbar nach der Parameterliste anzugeben.

- Syntax (Funktionsdeklaration) :



Beispiel:

```
class Counter  {                        // Klasse "Zaehler"
 public:
     unsigned long value() const;       // Ausgabe aktueller Zählerstand
     // ¨¨¨                             // weitere Member-Funktionen
 private:
     unsigned long act_count;           // aktueller Zählerstand
};

unsigned long Counter::value() const {
 return act_count;
}
```

- Member-Funktionen, die nur Werte zurückgeben sollen, sollten vorsichtshalber als const vereinbart werden. Bei normalen Funktionen ist der Zusatz const nicht zulässig.

- Über konstante Objekte dürfen nur konstante Member-Funktionen aufgerufen werden. Alle anderen - normalen - Memberfunktionen gelten als potentielle Verletzer der Konstantheit.

Beispiel :

```
class X {
 public:
     void writewert(int i) { wert=i; }
     int readwert() const { return wert; }
 private:
     int wert;
};

void f(const X& beisp)  {          // Parameter beisp ist konstantes Objekt
    int w;
    beisp.writewert(5);            // unzulässig
    w=beisp.readwert();            // zulässig
}
```

Konstante Member-Funktionen in C++ (2)

- Innerhalb einer konstanten Member-Funktion ändert sich der Typ des Pointers this in:

```
const X* const this;        //  konstanter Zeiger auf konstantes Objekt
```

Für die Member-Funktion ist das aktuelle Objekt also ein konstantes Objekt, auch wenn es sich tatsächlich um ein nicht-konstantes Objekt handelt. Über diesen Pointer ist dadurch weder explizit noch implizit (Zugriff zu Komponenten des aktuellen Objektes !) eine Änderung des Zeigerobjekts bzw. einer seiner Komponenten möglich.

Anmerkung: Es ist auch nicht möglich, innerhalb einer konstanten Member-Funktion eine nicht-konstante Member-Funktion aufzurufen, da der implizit übergebene this-Pointer bei beiden Funktionen einen unterschiedlichen Typ hat.

- Ausnahme :
Da Datenkomponenten, die „mutable“ spezifiziert sind, auch in konstanten Objekten geändert werden dürfen, können konstante Memberfunktionen derartige Komponenten modifizieren.

Beispiel :

```
class X       {
 public:
    void cwritewert(int i) const { wert=i; }        // unzulässig
    void cwritezahl(int i) const { zahl=i; }        // zulässig
    int readwert(void) const     { return wert; }
 private:
    int wert;
    mutable int zahl;
};

void f(const X& beisp)  {
 int w;
 beisp.cwritezahl(7);              // zulässig
 w=beisp.readwert();
}
```

4.4 Konstruktoren in C++

- Ein Konstruktor ist eine spezielle Member-Funktion einer Klasse, die bei der Erzeugung (statisch oder dynamisch) eines Objekts dieser Klasse automatisch aufgerufen wird ⇒ Konstruktor-Funktion. Grundsätzlich kann ein Konstruktor beliebige Aktionen ausführen. Sein Hauptzweck besteht aber in der Durchführung von Initialisierungen für das erzeugte Objekt (z.B. Initialisierung der Datenkomponenten). Die Ausführung von Aktionen durch den Konstruktor, die nicht direkt mit der Initialisierung verbunden sind, sollte daher vermieden werden (schlechter Programmierstil).

- Syntaktische Eigenschaften von Konstruktoren:
  * gleicher Name wie der Klassenname,
  * kein Funktionstyp (auch nicht void), damit auch keine Rückgabe eines Funktionswertes,
  * Parameter sind zulässig

  Konstruktoren sind normalerweise public.

- Beispiele :

```
 class Ratio {
   public:
     Ratio();          // Konstruktor
     // ....           // weitere öffentliche Member-Funktionen
   private:
     long zaehler;
     long nenner;
   };

Ratio::Ratio() {       // Definition des Konstruktors
   zaehler=0L;         // Initialisierung der Datenkomponenten
   nenner=1L;
}

Ratio bzahl;           // automat. Aufruf des Konstruktors ⇒ Initialisierung
```

```
class Datum  {
    public:
      Datum(int, int, int);       // Konstruktor
      // ....                     // weitere öffentliche Member-Funktionen
   private:
      int tag, monat, jahr;
};

Datum::Datum(int t, int m, int j) {
   tag=t;
   monat=m;
   jahr=j;
}
```

Konstruktoren in C++(2)

- Besitzt die Konstruktor-Funktion Parameter, so sind bei jeder Objekterzeugung (Definition bzw Allokation) entsprechende aktuelle Parameter anzugeben. Für die Objektdefinition existieren hierbei zwei Syntaxformen:

```
klassenname objektname = klassenname(aktpar_liste);
klassenname objektname(aktpar_liste);
```

  Die Angabe von zu wenig oder gar keinen Parametern ist fehlerhaft.

  Beispiele (Definition der Klasse Datum vorausgesetzt):

```
Datum weihnachten=Datum(25,12,1997);
Datum geburtstag(29, 2, 1980);
Datum *dp=new Datum(1,1,2000);
Datum heute;                                    // Fehler !
```

- Für Konstruktor-Funktionen mit nur einem Parameter existiert eine weitere Syntaxform für die Objektdefinition :

```
klassenname objektname = aktpar;
```

  Beispiel :

```
class String  {
      public:
        String(int);       // Konstruktor
        // ¨¨¨                   // weitere öffentl. Member-Funktionen
      private:
        char* sp;
        int maxlen;
        int aktlen;
 };

 String::String(int len) {
      sp=new char[len+1];  // Allok. des Speicherpl. für eigentl. String
      sp[0]='\0';
      maxlen=len;
      aktlen=0;
 }

 String s3=String(50);      // Erzeugung eines Strings der Länge 50
 String s1(100);            // Erzeugung eines Strings der Länge 100
 String s2=200;             // Erzeugung eines Strings der Länge 200
```

- Prinzipiell kann ein Konstruktor, **der mit einem Parameter aufgerufen werden kann**, als Funktion zum Konvertieren des Parameter-Typs in den Klassen-Typ betrachtet werden (dient auch zur automat. Typkonvertierung). Er wird deshalb auch **Konvertierungskonstuktor** genannt.
- Ein Konstruktor, der ohne Parameter aufgerufen werden kann, wird als **Default-Konstruktor** bezeichnet. Enthält die Klassendefinition keinen Konstruktor, so wird ein **Standard-Default-Konstruktor** aufgerufen, der nichts tut (keine Initialisierung!).
- Ein Konstruktor kann **nicht** wie andere Member-Funktionen als Komponente eines Objekts aufgerufen werden (keine Neu-Initialisierung eines existierenden Objekts!), sondern nur "freischwebend" zur Erzeugung eines anonymen, temporären Objekts (anwendbar auf der rechten Seite von Wertzuweisungen, als Funktionsparameter, als Funktionswert u.ä.).

Demonstrationsprogramm zu Konstruktoren in C++

```
/* ------------------------------------------------------------------    */
/*     Programm KONSTR                                                    */
/* ------------------------------------------------------------------    */
/* Demonstrationsprogramm zu Konstruktoren in C++                         */
/* ------------------------------------------------------------------    */
  #include <iostream >
  using namespace std;

  class My  {
    public:
      My(int);
      void myprint() const;
    private:
    int wert;
  };

  My::My(int w)  {
    cout << "Im Konstruktor, wert : " << w << ' ';
    wert=w;
  }

  void My::myprint(void) const  {
    cout << " Objekt.wert : " << wert << ' ';
  }

  My dummy(My p) {
    My du(11);
    cout << "\nK11 : ";
    p.myprint();
    cout << "\nK12 : ";
    du.myprint();
    cout << "\nK13 : ";
    return My(13);
  }

  void main() {
    My mu(1);
    My *mp;
    cout << "\nK1  : ";
    mu.myprint();
    cout << "\nK2  : ";
    mp=new My(2);
    mp->myprint();
    cout << "\nK3  : ";
    mu=dummy(My(9));
    cout << "\nK4  : ";
    mu.myprint();
  }
```

```
F:\RT\CPP\VORL>konstr
Im Konstruktor, wert : 1
K1  :  Objekt.wert : 1
K2  : Im Konstruktor, wert : 2  Objekt.wert : 2
K3  : Im Konstruktor, wert : 9 Im Konstruktor, wert : 11
K11 :  Objekt.wert : 9
K12 :  Objekt.wert : 11
K13 : Im Konstruktor, wert : 13
K4  :  Objekt.wert : 13
```

Konstruktoren in C++ (3) Defaultparameter

- Konstruktoren können auch Default-Parameter besitzen. Hierdurch wird es möglich, bei der Objekterzeugung eine unterschiedliche Anzahl von Parametern anzugeben. Beispiel :

```
class Ratio {
   public:
      Ratio(long=0, long=1);           // Konstruktor
      // ¨¨¨               // weitere öffentliche Member-Funktionen
   private:
      long zaehler;
      long nenner;
      void kuerze();
};

Ratio::Ratio(long z, long n)  {         // Definition des Konstruktors
   zaehler=z; nenner=n;
}

//Beispiele für gültige Objekt-Definitionen:
Ratio azahl;               // Initialisierung mit 0, 1
Ratio bzahl(5);            // Initialisierung mit 5, 1
Ratio czahl(7,3);          // Initialisierung mit 7, 3
```

- Konstruktoren können auch überladen werden. Hiermit ergibt sich gegebenenfalls eine weitere - allgemeinere - Möglichkeit, bei der Objekterzeugung eine unterschiedliche Anzahl von Parametern anzugeben. Beispiel :

```
class Ratio  {
   public:
      Ratio();         // Default-Konstruktor
      Ratio(long);          // Konstruktor mit einem Parameter
      Ratio(long, long);    // Konstruktor mit zwei Parametern
      // ¨¨¨            // weitere öffentliche Member-Funktionen
   private:
      long zaehler;
      long nenner;
      void kuerze();
};

Ratio::Ratio() {      // Definition des Default-Konstruktors
   zaehler=0L; nenner=1L;
}

Ratio::Ratio(long z) {          // Def. des Konstruktors mit einem Par.
   zaehler=z; nenner=1L;
}

Ratio::Ratio(long z, long n) {  // Def. des Konstruktors mit zwei Par.
   zaehler=z; nenner=n;
}

// Beispiele für gültige Objekt-Definitionen :
Ratio azahl;          // Initialisierung mit 0, 1
Ratio bzahl(5);       // Initialisierung mit 5, 1
Ratio czahl(7,3);     // Initialisierung mit 7, 3
```

Konstruktoren in C++ (4)

- Konstruktoren dürfen andere Member-Funktionen ihrer Klasse aufrufen.

  Beispiel :

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

class String  {
  public:
     String(int, const char * =""");    // Konstruktor
     void set(const char *);     // Member-Funktion zum String-Zuweisen
     // ....                     // weitere öffentliche Member-Funktionen
  private:
     char *cp;
     int maxlen;
     int aktlen;
};

String::String(int len, const char *s) {
    maxlen = len;
    cp=new char[maxlen+1];
    set(s);                     // Aufruf einer Member-Funktion
}

void String::set(const char *s)  {
    if (strlen(s)<= (unsigned) maxlen)  {
        strcpy(cp, s);
        aktlen=strlen(s);
    }
    else
        cout << "\nFehler : String ist zu lang\n";
}

String s1(30, "Hallo, World");
String s2(80);
String s3(10, "Fachbereich Elektrotechnik");      // ⇒ Fehlermeldung
```

- Weitere Eigenschaften von Konstruktoren :
  * Falls Komponenten einer Klasse selbst vom Klassen-Typ sind, werden bei einer Objekterzeugung zuerst die Konstruktoren der Komponenten aufgerufen.
    Ohne Verwendung einer Initialisierungsliste (siehe später) wird für jede Komponente automatisch deren Defaultkonstruktor aufgerufen. (siehe Bsp. RCDemo)

  * Die Konstruktoren von Array-Komponenten werden in der Reihenfolge aufsteigender Indices aufgerufen.

Beispiel zum Konstruktoraufruf bei Klassen mit Komponenten eines Klassen-Typs

```
/* ------------------------------------------------------------------     */
/* Programm RCDEMO  (k3_14)                                               */
/* ------------------------------------------------------------------     */
/* Komplexe Zahlen bei denen Real- und Imaginärteil                    */
/* Rationale Zahlen (Zähler, Nenner) sind                              */
/* ------------------------------------------------------------------     */
/* Demonstration des Konstruktoraufrufes bei Klassen, die         */
/* Komponenten eines Klassentyps haben                                 */
/* ------------------------------------------------------------------     */
  #include <iostream >
   using namespace std;

  class Ratio  {
      public:
        Ratio(long=0L, long=1L);    // Konstruktor für Ratio
        // ¨¨¨                      // weitere öffentliche Member-Funktionen
      private:
        long zaehler;
        long nenner;
  };

  class Rcomplex  {
      public:
        Rcomplex();                 // Konstruktor für Rcomplex
        // ¨¨¨                      // weitere öffentliche Member-Funktionen
      private:
        Ratio re;
        Ratio im ;
  };

  Ratio::Ratio(long z, long n)  {
      zaehler=z; nenner=n;
      cout << "\nInit Ratio-Objekt mit : " << z << ',' << n << '\n';
  }

  Rcomplex::Rcomplex()  {          // tue nichts
      cout << "\nInit Rcomplex-Objekt\n";
  }

  int main(void)  {
      Rcomplex rc1;          // rc1 wird initialisiert mit re = im = 0,1
      return 0;
  }
```

```
G:\>rcdemo
Init Ratio-Objekt mit : 0,1
Init Ratio-Objekt mit : 0,1
Init Rcomplex-Objekt
```

4.5 **Konvertierende und nicht-konvertierende Konstruktoren in C++**

- Ein Konstruktor, der mit einem einzelnen Parameter aufgerufen werden kann, definiert auch eine Konvertierung des Parameter-Typs in den Klassen-Typ des Konstruktors (⇒ konvertierender Konstruktor). Ein derartiger Konstruktor kann somit auch zur Realisierung impliziter (durch den Compiler) bzw. expliziter (Cast-Operator, Werterzeugungs-Operator, static_cast) Typkonvertierungen aufgerufen werden; dies auch in Fällen, in denen die damit mögliche Typkonvertierung wenig sinnvoll ist.

Beispiel :

```
class String  {
   public:
      String(int);     // Konstruktor,  Konvertierung int ⇒ String
      // ¨¨¨
};

String::String(int len) { /* ¨¨¨ */ }

void f()  {
 String s(40);                       // Expliziter Aufruf als Konstruktor
 String t=10;                        // Implizite Typkonvertierung
 s=20;                               // Implizite Typkonvertierung
 s=(String)25;                       // Explizite Typkonvertierung
 s=String(30);                       // Explizite Typkonvertierung
 // ¨¨¨
}
```

- Der implizite Aufruf eines Konstruktors - und damit seine versehentliche Verwendung für gegebenenfalls wenig sinnvolle implizite Typkonvertierungen - wird verhindert, indem man den Konstruktor mit dem vorangestellten Funktions-Spezifizierer „explicit“ deklariert (⇒ nicht-konvertierender Konstruktor):
  Der Funktions-Spezifizierer „explicit“ ist nur bei der Konstruktor-Deklaration innerhalb der Klassendefinition anzugeben, in einer eventuellen Definition außerhalb der Klassendefinition ist er nicht zu wiederholen. Ein nicht-konvertierender Konstruktor kann nur explizit verwendet werden, allerdings auch für explizite Typkonvertierungen.

Beispiel :

```
class String  {
 public:
   explicit String(int);      // Konstruktor,  Konvert. int ⇒ String
   // ¨¨¨
};
String::String(int len) { /* ¨¨¨ */ }

void f()  {
 String s(40);                       // Expliziter Aufruf  ⇒ zulässig
 String t=10;                        // Implizite Konvert. ⇒ Fehler !
 s=20;                               // Implizite Konvert. ⇒ Fehler !
 s=(String)25;                       // Explizite Konvert. ⇒ zulässig
 // ¨¨¨
}
```

4.6 **Destruktoren in C++**

- Ein Destruktor ist das Gegenstück zum Konstruktor. Er ist eine weitere spezielle Member-Funktion einer Klasse. Er wird automatisch bei Beendigung der Lebensdauer eines Objekts der Klasse aufgerufen, unmittelbar vor der Freigabe des Speicherplatzes für das Objekt (⇒ Destruktor-Funktion). Auch ein Destruktor kann grundsätzlich beliebige Aktionen ausführen. Sein Zweck besteht aber im Rückgängigmachen von Initialisierungen, die mit dem Konstruktor durchgeführt wurden.
  Typische Anwendung : Freigabe von Speicher, der durch den Konstruktor allokiert wurde.

  Anmerkung: der Speicher für das Objekt selber wird durch den Destruktor nicht freigegeben.

- Syntaktische Eigenschaften von Destruktoren :
  * Destruktor-Name ist der Klassenname mit vorangestelltem ~ ("Komplement des Konstruktors")
  * kein Funktionstyp (auch nicht void), damit auch keine Rückgabe eines Funktionswertes
  * keine Parameter
  * Überladen ist nicht möglich
  * Destruktoren dürfen nicht static sein

  Beispiel :

```
class String  {
   public:
     String(int);         // Konstruktor
     ~String();           // Destruktor
     // ¨¨¨               // weitere öffentl. Member-Funktionen
   private:
     char* sp;
     int maxlen;
     int aktlen;
};

String::String(int len) {
   sp=new char[len+1];    // dyn. Allok. des Speichers für eigentl. String
   sp[0]='\0';
   maxlen=len;
   aktlen=0;
}

String::~String()  {
   delete[] sp;          // Freigabe des mit Konstr. dyn. allok. Speichers
}

int main()  {
   String str(80);       // Aufruf des Konstruktors
   // ¨¨¨¨
   return 0;             // Aufruf des Destruktors
}
```

Destruktoren in C++ (2)

- Eigenschaften von Destruktoren :
  * Ein Destruktor darf andere Member-Funktionen seiner Klasse aufrufen
  * Der Aufruf von Destruktoren erfolgt in umgekehrter Reihenfolge der zugehörigen Konstruktoraufrufe (z.B. wird der Destruktor einer Klasse vor den eventuellen Destruktoren von Klassen-Komponenten aufgerufen).

4.7 Copy-Konstruktor in C++

- Bei der Definition (Erzeugung!) eines Objekts ist es möglich, dieses durch Angabe eines anderen Objekts zu initialisieren.
  Beispiel :

```
Ratio rz1(7, 3);
Ratio rz2=rz1;
```

Ermöglicht wird dies durch einen - implizit vom Compiler erzeugten Standard-Copy-Konstruktor, der bei dieser Art Initialisierung automatisch aufgerufen wird. Dieser spezielle Konstruktor kopiert die Werte sämtlicher Komponenten des (existierenden) unverändert in die entsprechenden Komponenten des (neu erzeugten) Zielobjekts.

⇒ das Zielobjekt ist eine identische Kopie des Quellobjekts.

Der Copy-Konstruktor wird auch automatisch aufgerufen in zwei weiteren Initialisierungs-Situationen:

* Übergabe eines Objekts als Parameter an eine Funktion (Initialisierung des - neu erzeugten - Parameters mit dem übergebenen Objekt)
* Rückgabe eines Objekts als Funktionswert
  (Initialisierung eines - neu erzeugten - temporären Objekts mit dem Rückgabewert)

- In allen drei Initialisierungssituationen kann die identische Kopie des Quellobjekts in das - neu erzeugte - Zielobjekt Probleme aufwerfen. Dies ist z.B. immer dann der Fall, wenn das Objekt einen Pointer auf dynamisch allokierten Speicher enthält, wie z.B bei der Klasse String. Durch die komponentenweise Kopie enthalten Quell- und Ziel-Objekt den gleichen Pointer, zeigen also auf den gleichen Speicherbereich. Eine Änderung (oder Freigabe) dieses Speichers bei einem Objekt bewirkt eine identische Änderung (bzw Freigabe) des Speichers beim anderen Objekt.

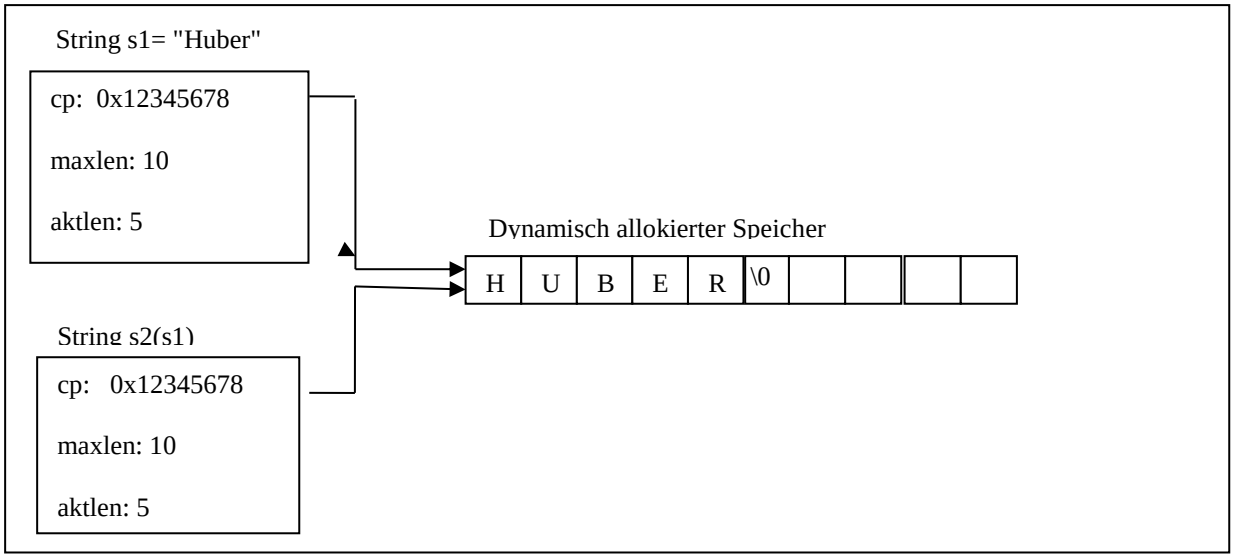


Bild: Standard-Copy-Konstruktor

Um derartige Probleme zu vermeiden, muß explizit ein eigener Copy-Konstruktor für die Klasse definiert werden.
Wenn ein eigener Copy-Konstruktor definiert ist, wird bei Initialisierungen dieser - statt des Default-Copy-Konstruktors - automatisch aufgerufen.
Er muss dafür sorgen, dass **nicht der Zeiger sondern der dynamische Speicher kopiert wird**. Es ergibt sich somit folgendes Bild:

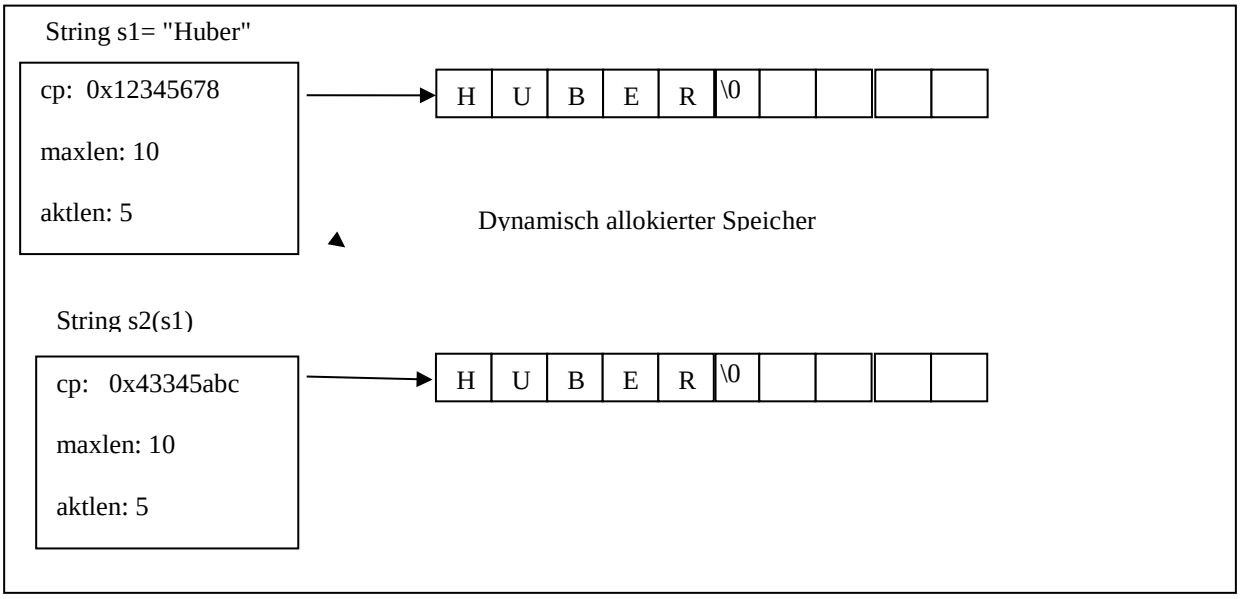


Bild:Eigener Copy-Konstruktor

- Ein Copy-Konstruktor hat genau einen Parameter, der eine Referenz auf ein konstantes Objekt der jeweiligen Klasse sein muß.
  Die allgemeine Form des Function Prototypes lautet: **klassenname(const klassenname &)**;

Beispiel :

```
class String  {
  public:
    String(int);                          // "normaler" Konstruktor
    String(const String&);                // Copy-Konstruktor
    ~String();                            // Destruktor
    // ¨¨¨              // weitere öffentl. Member-Funktionen
  private:
    char *sp;
    int maxlen;
    int aktlen;
};
```

- Anmerkung : Der Copy-Konstruktor wird nur bei Initialisierungen aufgerufen, nicht jedoch bei Zuweisungen.

Demonstrationsprogramm zum Copy-Konstruktor in C++ (1)

```
/* --- Headerdatei SimpStrg.h -------------------------------------------*/
/*      Definition der Klasse String
#ifndef SIMP_STR
#define SIMP_STR
class String  {
  public:
    // ....
    String(const char* ="");         // Konstruktor
    String(const String&);           // Copy-Konstruktor
    ~String(void);                   // Destruktor
    char* get() const { return cp; } // ermöglicht Zugriff zum String
  private:
    char *cp;
    int len;
};
#endif
```

```
/* --- Quelldatei SimpStrg.cpp ------------------------------------------*/
/*     Implementierung der Klasse String              */

#include <iostream >
#include <cstring>
#include <cstdlib>
#include "SimpStrg.h"
using namespace std;

  String::String(const char *s)  {    // "normaler" Konstruktor
    len=strlen(s);
    if ((cp=new char[len+1])==NULL) {
      cout << "\nAllokations-Fehler\n";
      exit(1);
    }
    strcpy(cp, s);
    cout << "  normaler Konstruktor : " << '\"' << cp << "\"\n";
  }

  String::String(const String& so)  { // Copy-Konstruktor
    len=so.len;
    if ((cp=new char[len+1])==NULL) {            // Allokation eines eigenen
      cout << "\nAllokations-Fehler\n";          // Speicherbereichs
      exit(1);
    }

    strcpy(cp, so.cp);
    cout << "  Copy-Konstruktor : " << '\"' << cp << "\"\n";
  }

  String::~String()  {                // Destruktor
    cout << "  Destruktor : " << '\"' << cp << "\"\n";
    delete [] cp;
  }
```

Demonstrationsprogramm zum Copy-Konstruktor in C++ (2)

```
/* --- Quelldatei CKDEMO.cpp ---------------------    */
/*      Anwendung der Klasse String                   */
#include <iostream >
#include "SimpStrg.h"
using namespace std;

  void show(String x) {               // keine Member-Funktion !!
     char* s=x.get();
     cout << s ;
  }
  int main(void)  {
  String a("Hallo ");
  String b("Du da ! ");
  String c=b;
  show(a);
  show(b);
  show(c);
  return 0;
  }
```

Ausgaben des Programmes DKDEMO:

```
  normaler Konstruktor : "Hallo "
  normaler Konstruktor : "Du da ! "
  Copy-Konstruktor : "Du da ! "
  Copy-Konstruktor : "Hallo "
  Hallo
  Destruktor : "Hallo "
  Copy-Konstruktor : "Du da ! "
  Du da !
  Destruktor : "Du da ! "
  Copy-Konstruktor : "Du da ! "
  Du da !
  Destruktor : "Du da ! "
  Destruktor : "Du da ! "
  Destruktor : "Du da ! "
  Destruktor : "Hallo "
```

4.8 **Konstruktoren mit Initialisierungsliste (C++)**

- Eine Klasse X kann auch Komponenten enthalten, die selbst von einem Klassen-Typ M sind.
  Bei einer Objekterzeugung werden zuerst die Konstruktoren der Komponenten aufgerufen. Ohne weitere Angaben wird für jede Komponente automatisch deren Defaultkonstuktor ausgeführt. Dadurch werden die Komponenten zunächst mit Defaultwerten initialisiert. Ihre endgültigen Werte erhalten sie bei der Ausführung des Konstruktors der Klasse X durch Zuweisung.
  Der Aufruf von Defaultkonstruktoren für Komponenten vom KlassenTyp (Teilobjekten) hat verschiedene Nachteile:
  - Ein Teilobjekt wird zunächst mit Defaultwerten versehen. Erst anschließend erhält es durch Zuweisung die richtigen Werte. Diese teilweise überflüssigen Aktionen beeinträchtigen die Performance des Programms.
  - Konstante Objekte oder Referenzen können nicht als Teilobjekte deklariert werden, da eine nachträgliche Zuweisung nicht möglich ist.
  - Klassen für die kein Defaultkonstruktor vorhanden ist, können nicht als Typ für Teilobjekte verwendet werden.

  Um diese Nachteile nicht in Kauf nehmen zu müssen, wird eine Syntax benötigt, die es erlaubt Teilobjekte explizit zu initialisieren. Zu diesem Zweck dienen Initialisierungslisten. (Siehe auch Initialisierung von Komponenten der Basisklasse im Kapitel Vererbung)
- Eine **Initialisierungsliste** ist im Funktionskopf der Konstruktor-Definition (nicht jedoch in der Deklaration) anzugeben. Hierfür gilt eine entsprechend erweiterte Syntax:

```
klassenname::klassenname(parameterliste) : initialisierungsliste  {
   // Rumpf der Konstruktor-Funktion
}
```

  Aufbau von „initialisierungsliste“:
  komponentenname(parameter, ...), komponentenname(parameter, ...), ...

```
/* ------------------------------------------------------------------ */
/* Programm RCDEMO mit Initialisierungsliste (k3_14)                  */
/* ------------------------------------------------------------------ */
/* Komplexe Zahlen bei denen Real- und Imaginärteil                   */
/* Rationale Zahlen (Zähler, Nenner) sind                             */
/* Demonstration des Konstruktoraufrufes bei Klassen, die             */
/* Komponenten eines Klassentyps haben                                */
/* ------------------------------------------------------------------ */
  #include <iostream>
  using namespace std;
   class Ratio  {
      public:
        Ratio();                    //Default-konstruktor
        Ratio(long,long);
        //Ratio(long =0L, long=1L); // Konstruktor für Ratio
        // ¨¨¨                      // weitere öffentliche Member-Funktionen
      private:
        long zaehler;
        long nenner;
   };
```

```
class Rcomplex  {
      public:
        Rcomplex();                 // Konstruktor für Rcomplex
        Rcomplex(long z_re,long n_re,long z_im,long n_im);
        // ¨¨¨                      // weitere öffentliche Member-Funktionen
      private:
        Ratio re;
        Ratio im ;
  };


  Ratio::Ratio()  {
      zaehler=0; nenner=1;
      cout << " \nDefaultkonstuktor fuer Ratio" <<endl;
  }

  Ratio::Ratio(long z, long n)  {
      zaehler=z; nenner=n;
      cout << "\nKonstruktor fuer Ratio"<<endl;
      cout << "Init Ratio-Objekt mit : " << z << ',' << n << '\n';
  }

  Rcomplex::Rcomplex()  {     // tue nichts
    cout << "\nDefault-Konstruktor fuer Rcomplex" <<endl;
    //cout << "\nInit Rcomplex-Objekt"<<endl;
  }

  //Keine gute Lösung, Nachteile siehe Skriptum Initialisierungsliste
/*Rcomplex::Rcomplex(long z_re,long n_re,long z_im,long n_im)
  {
    cout <<"\nKonstruktor mit Parametern fuer Rcomplex" <<endl;
    //mit Hilfe von Konstruktoraufrufen  bzw. set-Funktionen für Ratio
    re=Ratio(z_re,n_re); //Zuweisungen nicht immer möglich
    im=Ratio(z_im,n_im);
  }
*/

  //Konstruktor mit Initialisierugsliste ist optimale Lösung
  Rcomplex::Rcomplex(long z_re,long n_re,long z_im,long n_im)
  :re(z_re,n_re),im(z_im,n_im)
  {
    //restliche Aktionen
  }

  int main()  {
     Rcomplex rc1;          // rc1 wird initialisiert mit re = im = 0,1
     Rcomplex rc2(1,2,3,4);
      return 0;
  }
```

Konstruktoren mit Initialisierungsliste (C++) (2)

- Die Einträge in der Initialisierungsliste führen zum Aufruf des Konstruktors der jeweiligen Komponente. Die Reihenfolge der Konstruktoraufrufe wird durch die Komponenten-Reihenfolge in der Klassen-Definition und nicht durch die Reihenfolge in der Initialisierungsliste bestimmt. Die Komponenten-Konstruktoren werden vor der Ausführung der Anweisungen im Rumpf des Konstruktors aufgerufen.

- Eine Initialisierungsliste kann auch Einträge für Komponenten eines einfachen Datentyps enthalten.
  ⇒ Der Konstruktor für die Klasse Ratio kann alternativ auch folgendermaßen formuliert werden:

```
Ratio::Ratio(long z, long n) : zaehler(z), nenner(n)  {

}
```

- Referenz- und const-Komponenten einer Klasse können nur mittels einer Initialisierungsliste und nicht durch Zuweisungen im Rumpf der Konstruktor-Funktion initialisiert werden.

Beispiel:

```
class String {
    public:
       String(int);
       // ¨¨¨                      // weitere öffentliche Member-Funktionen
    private:
       char *sp;
       const int maxlen;
       int aktlen;
};

String::String(int len) : maxlen(len) {
    sp=new char[len+1];
    sp[0]='\0';
    aktlen=0;
}
```

4.9 **Initialisierung von Arrays aus Klassenobjekten in C++**

- Definition von Arrays :
  Für jedes Array-Element erfolgt ein eigener Konstruktor-Aufruf, wobei jeweils ein eigener Parametersatz vorliegen kann. Die einzelnen Konstruktoraufrufe werden - jeweils durch Kommata getrennt - mit geschweiften Klammern in einer Liste zusammengefaßt. Die Zuordnung zu den einzelnen Array-Elementen erfolgt - wie bei Arrays mit einfachen Element-Typen entsprechend der Reihenfolge in der Liste. Für Elemente, für die kein Konstruktoraufruf angegeben ist, was nur am Ende des Arrays möglich ist, wird der Default-Konstruktor aufgerufen. Ein expliziter Aufruf des Default-Konstruktors ist allerdings auch für Komponenten am Anfang oder in der Mitte des Arrays möglich.
  Der Aufruf eines Konstruktors mit einem Parameter kann durch die Angabe des Parameters allein ersetzt werden.

  Beispiele :

```
class Ratio  {
  public:
     Ratio();
     Ratio(long);
     Ratio(long, long);
     // ……
};
// ……
Ratio rfeld1[5] = { Ratio(1,5), Ratio(), Ratio(7), 9 };
Ratio rfeld2[]  = { Ratio(2,11), Ratio(5,9), 3, 11 };
Ratio rfeld3[7];
```

- Bei der dynamischen Allokation von Arrays mittels new, können keine expliziten Konstruktoraufrufe angegeben werden. Für jedes Element wird immer der Default-Konstruktor aufgerufen.

- Klassenkomponenten, die Arrays von Klassenobjekten sind :
  Für alle Elemente derartiger Klassenkomponenten wird ebenfalls immer der Default-Konstruktor aufgerufen. Sie lassen sich damit nicht explizit initialisieren, d.h. für sie darf kein Eintrag in der Initialierungsliste vorhanden sein.

  Beispiel :

```
#define MAXLEN 100
class Ratiovek  {
  public:
     Ratiovek(int);
     // ……                              //weitere öffentliche Member-Funktionen
  private:
     int aktlen;
     Ratio rfeld[MAXLEN];        // Initialisierung mit Ratio()
};

Ratiovek::Ratiovek(int len) : aktlen(len<MAXLEN ? len : MAXLEN) { }
```

4.10 Freund-Funktionen in C++

- Eine Funktion, die nicht Member-Funktion der Klasse ist, kann zum Freund dieser Klasse erklärt werden.

  $\Rightarrow$ Freund-Funktion (friend function)

  Eine Freund-Funktion hat - wie eine Member-Funktion - den vollen Zugang zu allen privaten Komponenten (private und protected) der Klasse. Als Freund-Funktionen sind sowohl "freie" Funktionen als auch Member-Funktionen anderer Klassen möglich. Eine Funktion kann Freund-Funktion mehrerer Klassen sein.
  Damit sind Funktionen möglich, die zu den privaten Komponenten von zwei und mehr Klassen zugreifen können.
  Member-Funktionen können das nicht, da sie nicht gleichzeitig Komponenten von zwei Klassen sein können.
  Freund-Funktionen werden nicht an abgeleitete Klassen vererbt.

- Deklaration einer Freund-Funktion
  * innerhalb der Klassendefinition durch Angabe eines Function Prototypes mit vorangestelltem Schlüsselwort friend,
  * kann sowohl im öffentlichen als auch im privaten Teil der Klassendefinition erfolgen (kein Unterschied).

  Beispiel :

```
class matrix;           // Vorwärts-Deklaration einer Klasse
class vector  {
  public:
    vector(int);
    // ……
    friend vector multiply(const matrix&, const vector&, int);
  private:
    int akt_n;
    double b[50];
};

class matrix {
  public:
    matrix(int);
    // ……
    friend vector multiply(const matrix&, const vector&, int);
  private:
    int akt_n;
    vector a[50];
};

// ……
vector multiply(const matrix& m, const vector& v, int n) {
  // direkter Zugriff zu Komponenten sowohl von m als auch von v
}
```

Freund-Funktionen in C++ (2)

- Wird eine Member-Funktion einer anderen Klasse als Freund-Funktion deklariert, so ist ihr "voll qualifizierter" Funktionsname - mit Klassenname und Scope Resolution Operator - wie folgt anzugeben:

  `klassenname::funktionsname`

  Beispiel :

```
class moritz;                  // Vorwärts-Deklaration einer Klasse
class max  {
   public:
      // ……
      int aergern(max, moritz);
   private:
      // ……
};
class moritz  {
   public:
      // ……
      friend int max::aergern(max, moritz);
   private:
      // ……
};
```

- Freund-Funktionen einer Klasse sind nicht Komponenten dieser Klasse.
  Sie dürfen daher nicht als Komponente eines Objekts dieser Klasse aufgerufen werden.
  Beispiel (s. vorher):

```
vector bv, rv;
matrix am;
rv=bv.multiply(am, bv, 10);      // Fehler !!!
rv=multiply(am, bv, 10);   // richtig
```

- Member-Funktionen können zu den Komponenten ihrer Klasse allein über den Komponentennamen zugreifen. Freund-Funktionen benötigen jedoch ein Objekt der Klasse zum Komponentenzugriff. Im Allgemeinen wird ihnen dieses als Parameter übergeben oder lokal definiert bzw. erzeugt.
  Beispiel:

```
vector multiply(const matrix& m, const vector& v, int n)  {
   vector r(n);
   for (int i=0; i<n; i++) {
      r.b[i]=0;
      for (int j=0; j<n; j++)
         r.b[i]+=m.a[i].b[j]*v.b[j];
      }
   return r;
}
```

4.11 Statische Klassenkomponenten in C++

- Klassenkomponenten können auch mit dem zusätzlichen Schlüsselwort static vereinbart werden.
  ⇒ statische Klassenkomponenten

- Im Unterschied zu normalen Datenkomponenten werden statische Datenkomponenten nicht für jedes Objekt der Klasse sondern nur einmal für die Klasse selbst angelegt ⇒ Klassenvariable.
  Eine statische Datenkomponente wird zu Programmbeginn für die gesamte Programmlaufzeit angelegt. Sie existiert damit bereits bevor ein Objekt der Klasse erzeugt wird.
  Eine statische Datenkomponente ist allen Objekten der Klasse zugänglich. Ein Objekt kann zu ihr - über die Member-Funktionen - genauso wie zu seinen normalen Datenkomponenten zugreifen.
  Im Prinzip sind statische Datenkomponenten globale Variable, deren Gültigkeitsbereich aber auf die Klasse, in der sie vereinbart sind, beschränkt ist. (Die Verwendung normaler globaler Variabler innerhalb von Member-Funktionen verstößt gegen das Prinzip der Kapselung.)

- Statische Datenkomponenten werden innerhalb der Klassendefinition nur deklariert. Ihre Definition (Speicherplatzallokation) muß außerhalb der Klassendefinition auf der globalen Ebene erfolgen. Hierbei ist dem Komponentennamen der Klassenname und der Scope Resolution Operator voranzustellen :

```
typangabe klassenname::komponentenname;
```

  Das Schlüsselwort „static“ darf bei der Definition nicht angegeben werden. Defaultmäßig werden statische Datenkomponenten bei der Definition mit 0 initialisiert. Eine explizite Initialisierung mit einem anderen Wert ist - auch bei private-Komponenten - zulässig.
  Beispiel :

```
class Exam {
   public:
      // ¨¨¨
      static int iPub;      // Deklaration einer stat. Komp.
   private:
      // ¨¨¨
      static int iPriv;     // Deklaration einer stat. Komp.
};
// Definition der stat. Komponenten
int Exam::iPub;            // Default-Init. mit 0
int Exam::iPriv=3;         // Explizite Initialisierung
```

- Im Unterschied zu normalen Klassenkomponenten kann zu static-Datenkomponenten - falls sie public sind - auch von außen ohne Bindung an ein konkretes Objekt unter Verwendung des Klassennamens und des Scope Resolution Operators zugegriffen werden.
  Beispiel :

```
int main() {    // Definition Klasse Exam s. oben
   int i;
   // ¨¨¨
   i=Exam::iPub;    // Zugriff ohne Bindung an ein Objekt
   // ¨¨¨
}
```

Statische Klassenkomponenten in C++ (2)

- Statische Datenkomponenten eines const-qualifizierten ganzzahligen oder Aufzählungs-Typs können auch innerhalb der Klassendefinition mit einem ganzzahligen konstanten Ausdruck initialisiert werden. In diesem Fall können sie innerhalb ihres Gültigkeitsbereichs selbst in konstanten Ausdrücken auftreten.
  Eine Definition außerhalb der Klassendefinition ist trotzdem erforderlich. Dabei dürfen dann keine Initialisierungswerte angegeben werden.

  Beispiel :

```
class DoubStack  {
// ¨¨¨
  private:
       static const int ciStkSize = 2000;
       double adStack[ciStkSize];
       // ¨¨¨
};

const int DoubStack::ciStkSize;
```

- Lokale Klassen dürfen keine statischen Datenkomponenten besitzen.
- Typische Anwendung statischer Datenkomponenten :
  * Koordinierung des Zugriffs zu von allen Objekten einer Klasse gemeinsam genutzter Resourcen (z.B. Buffer)
  * Zähler für die Anzahl der Objekte einer Klasse
  * Bereitstellung eindeutiger ID-Nummern oder bestimmter Grund-(Default-) Werte für die Objekte einer Klasse

- Anwendungsbeispiel zu statischen Datenkomponenten :
  Realisierung einer Klasse, deren Konstruktor nur die Erzeugung einer bestimmten Anzahl von Objekten zuläßt.

```
class Single  {
  public:
    Single();                          // Konstruktor
    // ¨¨¨
  private:
    static int iAnzObj;                // Deklaration der stat. Komponente
    // ¨¨¨
};

Single::Single()  {                    // Definition des Konstruktors
  if (--iAnzObj<0) {
    cout << "\nAlle zulässigen Objekte bereits angelegt !\n";
    exit(1);
  }
}

int Single::iAnzObj=3;     // Definition u. Init. stat. Komp. mit Anzahl
```

Statische Klassenkomponenten in C++ (3)

- Auch Member-Funktionen können als static deklariert werden. ⇒ statische Member-Funktionen
  Im Unterschied zu normalen Member-Funktionen besitzen statische Memberfunktionen keinen this-Pointer. Sie können daher zu normalen (nicht-statischen) Komponenten nur unter Anwendung der Operatoren „.“ bzw. „->“ auf ein Objekt bzw. einen Objekt-Pointer zugreifen. Der Zugriff zu statischen Komponenten ist dagegen allein über den Komponentennamen möglich.
  Statische Member-Funktionen einer Klasse können ohne Bindung an ein konkretes Objekt dieser Klasse aufgerufen werden. Hierfür muß ihr Funktionsname durch den Klassennamen und den Scope Resolution Operator ergänzt werden:

```
klassenname::funktionsname(parameter)
```

- Typische Anwendung statischer Member-Funktionen :
  Objektunabhängiger Zugriff zu privaten statischen Datenkomponenten einer Klasse.

  Beispiel :

```
class Exam  {
  public:
    // ……
    static int GetPriv(void); // Deklaration stat. Member-Funktion

   private:
    // ……
    static int iPriv;          // Deklaration stat. Datenkomponente
};

int Exam::GetPriv(void) { return iPriv; }
int Exam::iPriv=3;                // Definition der stat. Komponente
                                  // Explizite Initialiisierung
int main()  {
   int i;
   // ……
   i=Exam::GetPriv();             // Funktionsaufruf ohne Bindung
    // ……                        // an ein Objekt
}
```

- Der Zugriff zu einer statischen Member-Funktion als Komponente eines Objekts ist zwar möglich, sollte i.a. aber vermieden werden.

Demonstrationsprogramm zu statischen Klassenkomponenten in C++

```
/* ------------------------------------------------------------------  */
/*                          Programm PERSON                            */
/*  ---------------------------------------------------------------    */
/*          Demonstrationsprogramm zu statischen Klassenkomponenten     */
/* ------------------------------------------------------------------*/

#include <iostream>
#include "SimplStr.h"              // enthält Definition der Klasse String
using namespace std;
class Person  {
  public:
    Person(const String&, const String& =""");        // Konstruktor
    Person(const Person&);                            // Copy-Konstruktor
    ~Person();                                        // Destruktor
    static long anzahl();          // Ermittlung akt. Anzahl aller Objekte
  private:
   String vorn;                    // Vorname
   String name;                    // Nachname
   const long m_pid;               // Personen-ID
   static long aktmaxid;           // aktuelle hoechste ID
   static long anz;                // aktuelle Anzahl aller Person-Objekte
};

// Def. und Init. der stat. Datenkomp.
long Person::aktmaxid=1000;               // Start-ID = 1000
long Person::anz=0;

Person::Person(const String& nn, const String& vn) :        // Konstruktor
  name(nn), vorn(vn), m_pid(aktmaxid++)  {                  // Init.-Liste
  anz++;  /* Personenanzahl erhoehen */
}

Person::Person(const Person& p):                      // Copy-Konstruktor
  name(p.name), vorn(p.vorn), m_pid(aktmaxid++)  {
  anz++;  /* Personenanzahl erhoehen */
}
Person::~Person()  {                      // Destruktor
  anz--;                                  // Personenanzahl erniedrigen
}
long Person::anzahl(){   // Ermittlung akt. Anzahl der Person-Objekte
  return anz;
}

int main(void)  {
  cout << "\nPersonenzahl: " << Person::anzahl() ;
  cout << "\nErzeugung einer neuen Person";
  Person chief("Kohlkopf", "Heini");
  cout << "\nPersonenzahl: " << Person::anzahl() << '\n';
  return 0;
}
```

```
F:\RT\CPP\VORL>person
Personenzahl: 0
Erzeugung einer neuen Person
Personenzahl: 1
```

4.12 Innere Klassen in C++ (1)

- **Definition geschachtelter Klassen**
  - ◇ **Klassendefinitionen** können **geschachtelt** werden → Definition einer Klasse innerhalb einer anderen Klasse.
  - ◇ Eine innerhalb einer anderen Klasse definierte Klasse heißt **innere Klasse** oder eingebettete Klasse (***nested class***). Eine Klasse, innerhalb der eine andere Klasse definiert ist, wird **äußere** oder **umschließende Klasse** (***enclosing class***) genannt.
  - ◇ **Beispiel :** Rückwärts verkettete Liste, Definition des Typs der Listenelemente als innere Klasse

```
class Set                                          // äußere Klasse
{
  public :
    Set() { last=NULL; }
    void insert(int val) { last = new SetMember(val, last); }

    // ...

  private :

    class SetMember                                // innere Klasse
    {
      public :
        SetMember(int val, SetMember* n)
        { memVal=val;
          next=n;
        }
      private :
        int memVal;
        SetMember* next;
    };

    SetMember* last;
};
```

  - ◇ Die Definition der inneren Klasse kann im **`public`-Teil oder** im **`private`-Teil** der äußeren Klasse erfolgen. Im **`private`-Teil** definierte innere Klassen können **nur innerhalb** der **äußeren Klasse** (sowie von Freunden) einschließlich weiterer innerer Klassen verwendet werden, im `public`-Teil definierte Klassen können dagegen auch außerhalb benutzt werden.
  - ◇ Der Name der inneren Klasse ist **lokal** zur äußeren Klasse.
    Die **innere** Klasse befindet sich damit **im Sichtbarkeitsbereich** (Verfügbarkeitsbereich, ***scope***) der **äußeren** Klasse.
    → eine **Verwendung** des Klassennamens **außerhalb der umschließenden** Klasse erfordert die **Qualifizierung** mit deren Namen.

```
class Outer
{
  public :
    class Inner
    {
       // ...
    };
  // ...
};

void func(void)                 // Verwendung der inneren Klasse von außerhalb
{
  Inner yourObj;                // fehlerhaft : Name Inner ist nicht sichtbar
  Outer::Inner myObj;           // ok
  //
```

Innere Klassen in C++ (2)

- **Beziehungen zwischen innerer und äußerer Klasse**

  ◇ Die Beziehungen zwischen innerer und äußerer Klasse **entsprechen** denen zwischen **separaten Klassen** mit der **Besonderheit**, dass sich die **innere** Klasse im **Sichtbarkeitsbereich** (*scope*) der **äußeren** Klasse befindet.
  Das bedeutet, dass die **innere Klasse** die in der **äußeren Klasse** definierten **Namen direkt verwenden** kann, der **gegenseitige Komponentenzugriff** sich aber nach den **üblichen Zugriffsrechteregeln** richtet.

  ⇨ ▻ Es existiert **keine** gegenseitige **`friend`-Eigenschaft** zwischen äußerer und innerer Klasse.

  ▻ **Memberfunktionen** der **äußeren** Klasse haben **keine besonderen Zugriffsrechte** zu den Komponenten der **inneren** Klasse **und umgekehrt**

  ▻ **Freunde** der einen Klasse sind **nicht** automatisch Freunde der anderen Klasse

  ◇ In Memberfunktionen und sonstigen **Vereinbarungen** der **inneren Klasse** dürfen unter Beachtung der Zugriffsrechte ohne konkreten Objektbezug nur **Typnamen**, **statische Komponenten** (Daten und Funktionen) sowie **Aufzählungskonstante** der **äußeren** Klasse verwendet werden.
  Eine Verwendung **nichtstatischer** Daten- oder Funktionskomponenten erfordert einen **konkreten Objektbezug** (Objekt, Referenz oder Pointer auf Objekt)

  ◇ Entsprechendes gilt für **Vereinbarungen** in der **äußeren** Klasse. Dabei müssen Typnamen, sowie die Namen von statischen Komponenten und Aufzählungskonstanten aus der **inneren Klasse** mit deren Namen **qualifiziert** werden.

  ◇ **Beispiel :**

```
class Outer
{
    enum State                                  // private !
    { READY, RUNNING, WAITING, STOPPED };

  public :
    int ioPub;
    static int ioStatPub;

    class Inner
    { public :
        int iiPub;
        static int iiStatPub;
        State st;              // ok : privater Typname kann benutzt werden,
                               //      wenn Name vorher vereinbart ist
        void doSomeIn(int i, Outer* op)
        { st=READY;     // Fehler : private Aufzaehlungskonstante von Outer
          ioPub=i;      // Fehler : nichtstatische Komponente von Outer
          ioStatPub=i;  // ok     : statische public-Komponente von Outer
          ioPriv=i;     // Fehler : private-Komponente von Outer u. nicht statisch
          ioStatPriv=i; // Fehler : private-Komponente von Outer
          op->ioPub=i;  // ok     : Zugriff zu public-Komp über Objekt-Pointer
          op->ioPriv=i; // Fehler : Zugriff zu private-Komponente
        }

      private :
        int iiPriv;
        static int iiStatPriv;
    };

    void doSomeOut(int i, Inner* ip)
    { iiStatPub=i;         // Fehler : Qualifikation mit Klassenname fehlt
      Inner::iiStatPub=i;  // ok     : statische Komponente von Inner
      Inner::iiPub=i;      // Fehler : nichtstatische Komponente von Inner
      Inner::iiStatPriv=i; // Fehler : private Komponente von Inner
      ip->iiPub=i;         // ok     : Zugriff zu public-Komp über Objekt-Pointer
      ip->iiPriv=i;        // Fehler : Zugriff zu private-Komponente
    }

  private :
    int ioPriv;
```

Innere Klassen in C++ (3)

- **Komponentendefinition außerhalb der Klassendefinition**

  ◇ **Memberfunktionen** innerer Klassen können auch **außerhalb der Klassendefinition** definiert werden.
  Statische Datenkomponenten müssen sogar außerhalb der Klassendefinition definiert werden.
  In beiden Fällen muß der **vollqualifizierte** Komponentenname (doppelte Qualifikation !) angegeben werden.

```
class Outer
{
  public :
    class Inner
    {
      public :
        void doSomeIn(int i, Outer* op);
        // ...
      private :
        static int iiStatPriv;
        // ...
    };
  //...
};

void Outer::Inner::doSomeIn(int i, Outer* op)
{ // ... }
```

  ◇ Dabei ist zu berücksichtigen, dass der **Sichtbarkeitsbereich** (*scope*) einer **Klasse** mit ihrem Scope-**Qualifier beginnt** (für die innere Klasse oben : mit `Outer::Inner::`) und sich **bis zum Ende** der jeweiligen Komponenten-**Definition erstreckt**.
  Das hat z.B. Konsequenzen für die Angabe des Rückgabetyps von Memberfunktionen, wenn dies ein innerhalb der äußeren Klasse definierter Typ ist.

```
class Outer
{
  public :
    class In1
    {
      // ...
    };

    class In2
    {
      public :
        In1 func(In2);
        // ...
    };
  // ...
};

In1 Outer::In2::func(In2 x)     // fehlerhaft : Qualifikation für In1 fehlt
{ // ...
}

Outer::In1 Outer::In2::func(In2 x)   // ok !
{ // ...
  return *(new In1);                 // ok : innerhalb der Funktionsdefini-
                                     //      tion ist Qualifikation für In1
```

Innere Klassen in C++ (4)

- **Vorwärtsdeklaration von inneren Klassen**
  - ◇ Eine **innere Klasse** kann innerhalb der Definition einer äußeren Klasse auch zunächst nur **deklariert** werden.
    → **Vorwärtsdeklaration** !
  - ◇ Die **Definition** der **inneren** Klasse muß dann entweder **innerhalb** der **äußeren** Klasse **später** oder **außerhalb** deren Klassendefinition erfolgen.
  - ◇ Bei der zweiten Möglichkeit kann die **Definition** der **inneren Klasse** nach **außen** (gegenüber dem Anwender) **verborgen** werden, da diese in einer eigenen Headerdatei, die dem Anwender nicht zugänglich gemacht wird, enthalten sein kann.
  - ◇ **Beispiel :**

```
class OuterFwd
{
  class InnerFwd2;              // Vorwärtsdeklaration
  // ...
  class InnerFwd1               // spätere Definition
  {
   // ...
  };
  // ...
};


class OuterFwd::InnerFwd2       // Definition in eigener Headerdatei
{
  // ...
};
```

- **Gründe für die Schachtelung von Klassen**
  - ◇ Wenn eine Klasse **nur innerhalb einer anderen Klasse benötigt** wird, kann sie als **innere Klasse** der anderen Klasse definiert werden. Sie tritt **nach außen** überhaupt **nicht in Erscheinung**.
    Dadurch wird die Anzahl der globalen Namen vermindert.
    Sinnvollerweise sollte die Definition der inneren Klasse im **`private`**-Teil der äußeren Klasse erfolgen
    Beispiel : Referenzzählung
  - ◇ Wenn eine Klasse **funktionell zu einer anderen Klasse** gehört, nur im Zusammenhang mit dieser Klasse sinnvoll eingesetzt werden kann, aber durchaus **außerhalb** dieser Klasse **verwendet** wird, sollte sie als **innere Klasse** im **`public`**-Teil der anderen Klasse definiert werden.
    Beispiel : Iteratoren

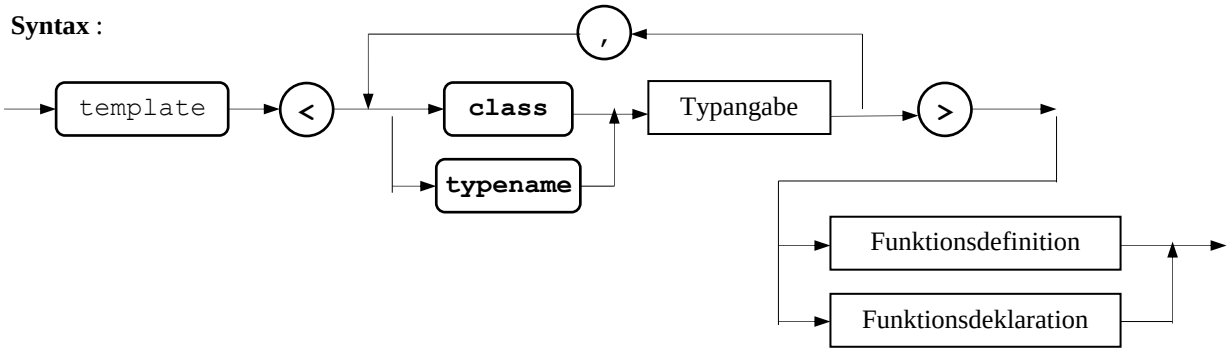
4.13 **Funktions- und Klassen-Templates**

4.13.1. Generische Funktionen (Funktions-Templates)

4.13.2. Generische Klassen (Klassen-Templates)

4.13.1 Generische Funktionen (Funktions-Templates) in C++

- **Allgemeines** :
  - ◇ Generische Funktionen sind **Funktions-Schablonen**, die ein Muster für eine ganze **Gruppe von Funktionen**, die den **prinzipiell gleichen Algorithmus** für jeweils **unterschiedliche Parametertypen** ausführen, definieren.
  - ◇ Die Definition einer generischen Funktion (Funktions-Template, ***function template***) erzeugt noch keinen Code. Passender – den Typen der jeweiligen aktuellen Parameter entsprechender – **Code** wird **vom Compiler** erst **bei der erstmaligen Verwendung der generischen Funktion** mit diesen Parameter-Typen erzeugt (**Instantiierung** des Templates → **instantiierte Funktion**, **Template-Funktion**, ***template function***).
  - ◇ Im Prinzip stellt eine generische Funktion eine Funktion dar, die sich **automatisch selbst überladen** kann. Im Unterschied zu allgemeinen überladenen Funktionen, bei denen jede Funktion prinzipiell einen anderen Algorithmus realisieren und eine unterschiedliche Anzahl von Parametern haben kann, haben alle Versionen einer generischen Funktion die gleiche Anzahl von Parametern und realisieren den gleichen Algorithmus - nur eben mit unterschiedlichen Parametertypen.
  - ◇ In der Vereinbarung einer generischen Funktion werden die **Parametertypen**, die sich **ändern** können, als (Template-) **Parameter** festgelegt.
- **Vereinbarung einer generischen Funktion** :
  - ◇ **Syntax** :



  - ◇ Typangabe ist ein **formaler Typ-Parameter** (**Template-Parameter**). Jeder Typ-Parameter kann innerhalb der **Parameterliste der Funktionsvereinbarung** als Platzhalter für eine **aktuelle Typangabe** verwendet werden (Festlegung des Typs eines Funktionsarguments).
  - ◇ Ein Template-Parameter kann auch verwendet werden
    - ▻ zur **Festlegung** des **Rückgabetyps** der Funktion
    - ▻ zur **Festlegung** des Typs **lokaler Variabler** der Funktion
  - ◇ **Beispiele** :

```
template <class T> void swap(T& a, T&b)
{
  T h=a;
  a=b;
  b=h;
}
template <typename TYPE1, typename TYPE2>
void myfunc(TYPE1 x, TYPE2 y)
{
  cout << x << "   " << y << '\n';
}
template <class T1, class T2>
T2 convert(T1 w)
{
  return (T2)w;
}
```

Generische Funktionen (Funktions-Templates) in C++ (2)

- **Aufruf von Template-Funktionen**

  ◇ Eine Template-Funktion wird **bei** ihrem **erstmaligen Aufruf generiert**. (→ **Implizite Instantiierung**)
  Für die **Angabe des Funktionsnamens** existieren hierfür zwei Möglichkeiten :
  ▻ Verwendung nur des Template-Namens als Funktionsnamen.
  ▻ Ergänzung des Template-Namens um die Angabe aktueller Template-Parameter

  ◇ Bei **Aufruf** einer Template-Funktion **allein** mit dem **Template-Namen** müssen die zu verwendenden aktuellen Template-Parameter aus den aktuellen Funktions-Parametern (Funktions-Argumenten) ermittelbar sein.
  Das bedeutet,
  - sämtliche Template-Parameter müssen zur Festlegung des Typs von Funktions-Parametern dienen
  - die Typen der aktuellen Funktions-Parameter müssen – ohne Anwendung impliziter Typkonvertierungen – eine **eindeutige Template-Instantiierung** ermöglichen.

  ◇ Bei **Aufruf** einer Template-Funktion mit dem um eine **aktuelle Parameterliste ergänzten Template-Namen** legt die Parameterliste die aktuellen Template-Parameter fest.
  In diesem Fall können gegebenenfalls implizite Typkonvertierungen der aktuellen Funktions-Parameter in die aktuellen Template-Parameter stattfinden.
  Dienen Template-Parameter allein zur Festlegung des Rückgabetyps der Funktion und/oder allein zur Festlegung des Typs von lokalen Funktions-Variablen muß diese Form des Aufrufs verwendet werden.

- **Templates bei mehreren Programm-Modulen**

  ◇ Bei der Übersetzung eines Template-Funktions-Aufrufs (d.h. bei der Erzeugung einer Template-Funktion) muss dem Compiler der Code des Funktions-Templates vorliegen.

  ◇ Soll dasselbe Funktions-Template in mehreren Modulen verwendet werden, ist es zweckmässig den **Template-Code** (also die Template-Definition) in eine **Headerdatei** aufzunehmen, die dann von den verwendenden Modulen eingebunden werden muss.

  ◇ Eine Template-Funktion, die nicht als `inline` definiert worden ist, wird wie eine normale globale Funktion behandelt. Sie lässt sich daher aus verschiedenen Modulen aufrufen. Andererseits darf sie in einem Programm nur einmal definiert sein.
  Der o.a. Mechanismus bewirkt aber zunächst, dass eine Template-Funktion in jedem Modul, in dem sie aufgerufen wird, auch erzeugt (d.h. definiert) wird.
  Der Linker ist dafür verantwortlich, dass – bis auf eine – alle weiteren Instanzen einer derartigen Funktion wieder entfernt werden.

- **Überladen generischer Funktionen** :

  ◇ Ein Funktions-Template läßt sich auch **explizit überladen**.

  ◇ Das Überladen kann erfolgen mit
  - einem **weiteren Funktions-Template**, das eine **unterschiedliche Funktions-Parameter-Liste** besitzt.
  - einer **Nicht-Template-Funktion**, die eine von dem Funktions-Template **abweichende Parameter-Liste** besitzt.
  - einer **Nicht-Template-Funktion**, deren **Parameter-Liste** mit der Parameter-Liste einer auch aus dem Template instantiierbaren Template-Funktion **übereinstimmt**.
    In diesem Fall "überschreibt" die explizit überladende Funktion die spezielle – ihren Parametertypen entsprechende – Template-Funktion (Instanz des Funktions-Templates).
    → Definition einer **expliziten Spezialisierung** des Funktions-Templates.

  ◇ ANSI-C++ sieht zur **Kennzeichnung** einer **expliziten Template-Spezialisierung** den Vorsatz

  `template<>` vor, der der Funktions-Vereinbarung vorangestellt werden kann.

Generische Funktionen (Funktions-Templates) in C++ (3)

- **Demonstrationsprogramm `genfunc1`**

```
// ------------------------------------------------------------------
// Programm genfunc1     (C++-Quelldatei genfunc1_m.cpp)
// Beispiel zu generischen Funktionen und zu zusätzlicher explizit
// überladender Funktion (explizite Spezialisierung)
// ------------------------------------------------------------------

#include <iostream>
using namespace std;


template <class TY> TY max(TY a, TY b)
{
  return a>b ? a : b;
}


template <typename TYPE1, typename TYPE2>
TYPE2 convert(TYPE1 w)
{
  return (TYPE2)w;
}


template<>                          // explizite Spezialisierung :
char *max(char *a, char *b)         // "überschreibt" generische Version
{                                   // von char *max(char *, char *)
  int i=0;
  while (a[i]==b[i] && a[i]!=0)
    i++;
  return a[i]>b[i] ? a : b;
}


int main(void)
{
  cout << endl << "int-max        : " << max<int>(7, -2);
  cout << endl << "double-max     : " << max(3.14, 27.9);
  cout << endl << "char-max       : " << max('a', 'z');
  cout << endl << "string-max     : " << max("Hausdach", "Haus");
  cout << endl << "long-max       : " << max(-256000L, 4L);
  cout << endl << "long-max       : " << max<long>(-256000L, 4);
  cout << endl << "convert (d->i) : " << convert<double, int>(45.14);
  cout << endl;
```

- ***Ausgabe des Programms***

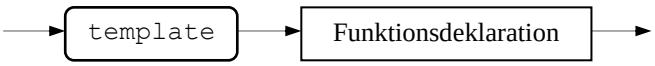
```
int-max        : 7
double-max     : 27.9
char-max       : z
string-max     : Hausdach
long-max       : 4
long-max       : 4
convert (d->i) : 45
```

Generische Funktionen (Funktions-Templates) in C++ (4)

- **Explizite Instantiierung eines Funktions-Templates**

  ◇ ANSI-C++ sieht auch eine explizite Instantiierung eines Funktions-Templates vor.
  Hierbei wird der Code einer Template-Funktion erzeugt, ohne dass die Funktion aufgerufen wird.

  ◇ **Syntax** :



  Beispiel :
```
template <class TY>           // Template-Definition
TY max(TY a, TY b) { return a>b ? a : b;}
template int max(int, int);   // explizite Template-Instantiierung
```

  ◇ Eine explizite Template-Instantiierung sollte zweckmässigerweise in einer `cpp`-Datei und nicht in einer Headerdatei stehen. Diese `cpp`-Datei kann dann auch die Template-Definition enthalten.

  ◇ Eine explizit instantiierte Funktion kann **in** anderen **Übersetzungseinheiten verwendet** (aufgerufen) werden, **ohne** dass sie **erneut generiert** werden muss.
  Für ihre Verwendung ist lediglich eine – häufig in eine Headerdatei gestellte – **Extern-Deklaration** erforderlich :
  - entweder der jeweiligen Template-Funktion
  - bzw. (als gemeinsame Deklaration für alle Template-Funktionen) des Funktions-Templates

- **Beispiel zur expliziten Instantiierung eines Funktions-Templates**

```
// C++-Headerdatei templmax2.h
// Nur Deklaration des Funktions-Templates max<>

template <class TY> TY max(TY a, TY b);
```

```
//C++-Quelldatei templmax.cpp
//Explizite Instantiierung eines Funktions-Templates

#include "templmax2.h"     // Einbinden der Headerdatei hier nicht erforderlich
template <class TY> TY max(TY a, TY b)
{ return a>b ? a : b;}

template int max(int, int);
template char max(char, char);
template double max(double, double);
template long max(long, long);
```

```
// Programm genfunc2     (C++-Quelldatei genfunc2_m.cpp)
// Beispiel zur Verwendung der expliziten Instantiierung eines Funktions-Templates
#include <iostream>
#include "templmax2.h"
using namespace std;

int main(void)
{ cout << endl << "int-max    : " << max(7, -2);
  cout << endl << "double-max : " << max(3.14, 27.9);
  cout << "char-max   : " << max('a', 'z');
  // cout << endl << "string-max : " << max("Hausdach", "Haus"); // keine
  //                                                            // Instantiierung!
  cout << endl << "long-max   : " << max<long>(-256000L, 4);
  cout << endl;
  return 0;
```

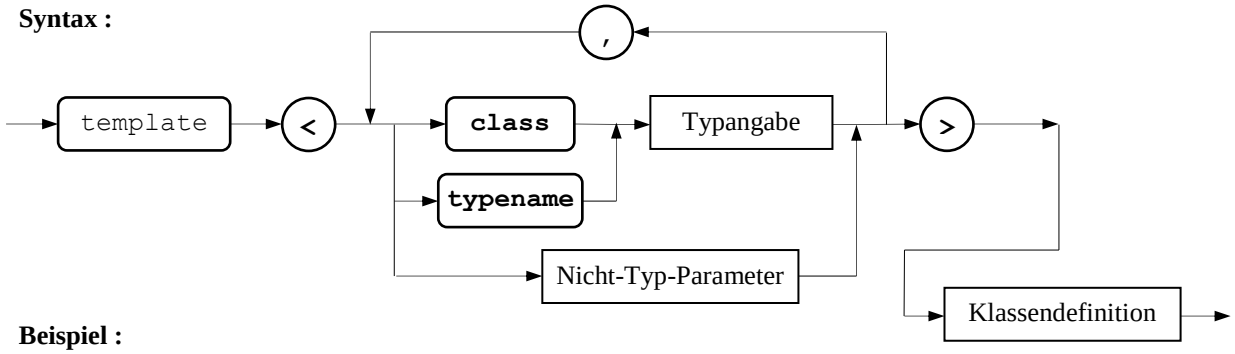
4.13.2 Generische Klassen (Klassen-Templates) in C++

- **Allgemeines :**

  Generische Klassen (Klassen-Templates, *class templates*) sind **Schablonen**, die ein **Definitions-Muster** für eine ganze **Gruppe von Klassen**, die **prinzipiell gleich aufgebaut** sind und deren **Member-Funktionen** den jeweils **prinzipiell gleichen Algorithmus** realisieren, festlegen.
  Die einzelnen nach diesem Muster konstruierbaren Klassen unterscheiden sich lediglich im **Typ einer oder mehrerer Komponenten** (und in den dadurch gegebenen Auswirkungen auf die Member-Funktionen).
  Die **Typen**, in denen sich die **einzelnen Klassen unterscheiden**, werden bei der **Definition** einer generischen Klasse als **formale Parameter** festgelegt. (Generische Klassen werden daher auch als **parameterisierte Typen** bezeichnet).

- **Definition einer generischen Klasse :**

  ◇ **Syntax :**



  ◇ **Beispiel :**

```
template <class T> class Stack  // Generische Klasse "Stack für Typ T"
{                               // (Klassen-Template)
  public:
    Stack (int);          // Konstruktor
    void push(T);         // Ablage eines Elements auf Stack
    T pop(void);          // Rückholen eines Elements vom Stack
  private:
    int size;             // Groesse des Stacks
    T *stck;              // Pointer auf Speicherbereich des Stacks
    int tos;              // Index des Top des Stacks (Stackpointer)
};
```

- **Template-Klassen :**

  ◇ Die **Definition einer generischen Klasse** generiert noch **keine konkrete Klasse** (und erzeugt noch keinen Code für Member-Funktionen). Dies erfolgt erst bei der **erstmaligen Verwendung** ihres Namens (des **Klassen-Template-Namens**) zusammen **mit aktuellen Parametern**
  → **implizite** (z.B. in einer Objekt-Vereinbarung) oder **explizite Instantiierung** des Klassen-Templates.
  Jeder **unterschiedliche** Satz **aktueller Parameter** definiert eine **neue Klasse**
  (→ instantiierte Klasse, **Template-Klasse**, *template class*).

  ◇ Der **Klassen-Template-Name** gefolgt von **in spitzen Klammern eingeschlossenen aktuellen Parametern** (Parameter-Zusatz) stellt den **Namen einer konkreten Klasse** dar und kann genauso wie jeder andere Klassenname verwendet werden (→ **Template-Klassen-Name**).

  ◇ **Syntax :** **`klassen-template-name<akt_par_liste>`**

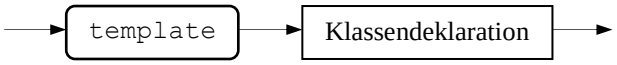
  ◇ **Beispiele :**

```
//implizite Template-Instantiierungen
Stack<int>    is(20);       // Stack fuer int-Werte
     Stack<char*>  cps(30);      // Stack fuer char-Pointer
           Stack<double> ds(50);       // Stack fuer double-Werte
```

  ◇ **Schlüsselwort** `typename` Mit dem Schlüsselwort `typename` legt man fest, dass ein nachfolgender Identifier ein Typ sein muss

Generische Klassen (Klassen-Templates) in C++ (2)

- **Explizite Instantiierung eines Klassen-Templates**
  - ◇ **Syntax** :



Beispiel : `template class Stack<int>;` `// Stack fuer int-Werte`

- **Verwendung des Klassen-Template-Namens**
  - ◇ **Außerhalb** der **Definition der generischen Klasse** darf der **Klassen-Template-Name nicht allein** sondern nur **zusammen mit** einem **Parameter-Zusatz** verwendet werden (gegebenenfalls mit den formalen Parametern, s. Definition der Member-Funktionen generischer Klassen).
  - ◇ **Innerhalb** der Klassen-Definition **darf** - muß aber nicht - bei **Eindeutigkeit** der **Parameter-Zusatz** mit den formalen Parametern **auch weggelassen** werden.
  - ◇ **Beispiel :**

```
template <class T> class List       // Generische Klasse : "Element für
{                                   // lineare Liste von Werten des Typs T"
  public :
    List(T);                        // Konstruktor
    ~List(void);                    // Destruktor
    void add(List<T> *node);        // oder :  ...(List *node);
    List<T> *getNext(void) const;   // oder :  List *get...
    T getData(void)const;
  private :
    T data;
    List<T> *next;                  // oder :  List *next;
};
```

- **Explizite Spezialisierung einer generischen Klasse**
  - ◇ Eine **generische Klasse** kann durch die **explizite Definition** einer **konkreten Klasse** für einen speziellen aktuellen Parametersatz "**überladen**" werden. Kennzeichnung nach ANSI-C++ : Vorsatz **`template<>`**
    Sofern die explizite Definition vor Anwendung der konkreten Klasse erfolgt, hat sie **Vorrang** vor einer Instantiierung der generischen Klasse für den speziellen Parametersatz.
  - ◇ Für eine explizit definierte konkrete Klasse müssen auch **alle Member-Funktionen** – auch wenn sie im Namen und Aufbau denen der generischen Klasse entsprechen – **gesondert definiert** werden.
  - ◇ **Beispiel :**

```
template <class T> class List    // Definition generische Klasse
{ /* ... */ };

template<>                      // Kennzeichnung einer
class List<char*>               // explizite Spezialisierung
{ public:                       // "Überladen" der generischen Klasse
    List<char*>(char *);
    ~List();
    void add(List<char*> *node);
    List<char*> *getNext(void) const;
    char *getData(void) const;
  private:
    char *data;
    List<char*> *next;
};
```

Generische Klassen (Klassen-Templates) in C++ (3)

- **Member-Funktionen generischer Klassen**

  ◇ Für **jede** aus einer generischen Klasse erzeugten **konkreten Klasse** wird ein **eigener Satz Member-Funktionen** angelegt.

  ◇ Jede Member-Funktion einer generischen Klasse ist **implizit** eine **generische Funktion** (Funktions-Template) mit der **gleichen Typ-Parameterliste** wie die **generische Klasse**.
  Bei ihrer **Definition außerhalb der Klassendefinition** muß sie daher **als Funktions-Template** (Beginn ebenfalls mit **`template < ... >`**) formuliert werden.

  Beispiel :

```
template <class T> class List
{ public:
    List(T);                            // Konstruktor
    void add(List<T> *node);            // Element zur Liste hinzufügen
    T getData(void)const;               // Rückgabe Komponente data
  // ...
};

template <class T> List<T>::List(T d)  // generischer Konstruktor
{ data=d; next=NULL; }

template <class T> void List<T>::add(List<T> *node)
{ node->next=this; next=NULL; }

template <class T> T List<T>::getData() const
{ return data; }
```

  ◇ Für **spezielle aktuelle Parameter** - also für eine spezielle konkrete Klasse (spezielle Instanz des Klassen-Templates) – kann eine **Member-Funktion** auch **überladen** werden.
  Bei einer expliziten Spezialisierung müssen alle Memberfunktionen gesondert definiert, d.h. überladen werden.

  Beispiel :

```
List<char *>::List(char *s)     // Konstruktor für konkrete Klasse
{ data=new char[strlen(s)+1];   // Überladen des generischen Konstruktors
  strcpy(data, s);
  next=NULL;
}
```

  ◇ **Explizite Funktions-Templates** als Member-Funktionen sind auch **möglich**.

- **Befreundete Funktionen generischer Klassen**

  ◇ Sind **nicht implizit** generische Funktionen.

  ◇ Eine befreundete Funktion kann **von Template-Parametern unabhängig** und damit für alle aus dem Klassen-Template erzeugbaren **konkreten Klassen dieselbe** Freund-Funktion sein.

  ◇ Eine befreundete Funktion kann aber auch **von Template-Parametern abhängen** und damit für **jede konkrete Klasse unterschiedlich** sein. In diesem Fall sollte sie **explizit als generische Funktion** definiert werden.

Demonstrationsbeispiel zu generischen Klassen in C++ (1)

```
// Header-Datei  StackTempl.h
// ---------------------------------------------------------------------
// Definition eines Klassen-Templates Stack
// ("einfacher Stack für unterschiedliche Objekte")
#ifndef STACK_TEMPL_H
#define STACK_TEMPL_H
#define DEF_SIZE 10

template <class T> class Stack   // Generische Klasse "Stack für ..."
{                                // (Klassen-Template)
  public:
    Stack (int=DEF_SIZE);        // Konstruktor
    ~Stack();                    // Destruktor
    void push(T);                // Ablage eines Elements auf Stack
    T pop(void);                 // Rückholen eines Elements vom Stack
  private:
    int size;                   // Größe (Tiefe) des Stacks
    T* stck;                    // Pointer auf Speicherbereich des Stacks
    int tos;                    // Index des Top of Stack (Stackpointer)
};

#endif
```

```
// C++-Quell-Datei  StackTempl.cpp
// Implementierung des Klassen-Templates Stack
#include "StackTempl.h"
#include <iostream>
#include <cstdlib>
using namespace std;

template <class T> Stack<T>::Stack(int s)
{ stck=new T[size=s];
  tos=0;
}

template <class T> Stack<T>::~Stack()
{ delete [] stck;
  stck=NULL;
  size=tos=0;
}

template <class T> void Stack<T>::push(T c)
{ if (tos==size)
    cout << "Stack ist voll !\n";
  else
  { stck[tos]=c;
    tos++;
  }
}

template <class T> T Stack<T>::pop(void)
{ if (tos==0)
  { cout << "Stack ist leer !\n";
    return (T)0;
  }
  else
  { tos--;
    return stck[tos];
  }
}
```

Demonstrationsbeispiel zu generischen Klassen in C++ (2)

```
// ------------------------------------------------------------------------
// C++-Quell-Datei templstack_m.cpp
// ------------------------------------------------------------------------
// Programm templstack
// ------------------------------------------------------------------------
// Einfaches Demonstrationsprogramm zu Klassen-Templates in C++
// ------------------------------------------------------------------------
// Verwendung des Klassen-Templates Stack
// ------------------------------------------------------------------------

#include "StackTempl.h"
#include "StackTempl.cpp"   // nur bei impliziter Template-Instantiierung

#include <iostream>
using namespace std;

int main(void)
{
  Stack<char> cs1, cs2(20);
  int i;

  cs1.push('A');
  cs2.push('b');
  cs1.push('C');
  cs2.push('D');
  for (i=1; i<3; i++)
    cout << '\n' << cs1.pop();
  cout << '\n';
  for (i=1; i<3; i++)
    cout << '\n' << cs2.pop();

  Stack<double> ds1(30), ds2;
  ds1.push(3.75);
  ds2.push(-4.7);
  ds1.push(2.3);
  ds2.push(5.83);
  cout << '\n';
  for (i=1; i<3; i++)
    cout << '\n' << ds1.pop();
  cout << '\n';
  for (i=1; i<3; i++)
    cout << '\n' << ds2.pop();
  cout << '\n';
  return 0;
}
```

- **Ausgabe des Programms :**

```
C
A

D
b

2.3
3.75

5.83
-4.7
```

Generische Klassen (Klassen-Templates) in C++ (4)

- **Andere Template-Parameter (Nicht-Typ-Parameter)**

  ◇ Klassen-Templates können auch **Parameter** besitzen, die **keine Typen** sind, sondern normalen Funktionsparametern entsprechen.
  Als entsprechende aktuelle Parameter sind nur zulässig :
  - konstante Ganzzahl- oder Aufzählungstyp-Ausdrücke (bei Nicht-Referenz-Parametern)
  - Adressen von global zugreifbaren Objekten und Funktionen (bei Pointer-Parametern)
  - Referenzen auf Objekte, die global (Speicherklasse `extern`) oder als `static`-Komponente definiert wurden (bei Referenz-Parametern)

  Sinnvoll sind solche Nicht-Typ-Parameter, wenn über sie bestimmte Eigenschaften einer konkreten Klasse oder ihrer Komponenten festgelegt werden können (z.B. die Größe eines Arrays o.ä.)

  ◇ **Beispiel :**

```
template <class T, int i> class Stack
{ public:
    Stack (void);
    void push(T);
    T pop(void);
  private:
    int size;
    T stck[i];  // Größe des Stacks durch Template-Parameter bestimmt
    int tos;
};

template <class T, int i> Stack<T,i>::Stack(void)
{
  size=i;
  tos=0;
}

// ...
int main(void)
{
  Stack<char,20> cs1;   // cs1 und cs2 sind Objekte
  Stack<char,50> cs2;   // unterschiedlicher Klassen
  Stack<char,2*25> cs3; // cs2 u. cs3 sind Objekte der gleichen Klasse
  Stack<double,30> ds;
  // ...
}
```

  ◇ **Zwei** aus **einem Klassen-Template** erzeugte **konkrete Klassen** (Template-Klassen) sind nur **dann gleich**, wenn sie in den **aktuellen Typ-Parametern übereinstimmen** und ihre aktuellen **Nicht-Typ-Parameter gleiche Werte** haben (s. obiges Beispiel).

- **`typedef`-Namen für Template-Klassen**

  ◇ Häufig eingeführt, um die wiederholte Auflistung von Template-Parametern zu vermeiden.

  ◇ **Beispiel :**

```
typedef Stack<char,20>   CharStack20;
typedef Stack<char,50>   CharStack50;
typedef Stack<double,30> DoubleStack30;
// ...
CharStack20 cs1;
CharStack50 cs2;
DoubleStack30 ds;
```

Generische Klassen (Klassen-Templates) in C++ (5)

- **Generische Klassen mit statischen Komponenten**

  ◇ Wenn eine generische Klasse statische Komponenten besitzt, werden für **jede** von ihr erzeugte **Template-Klasse eigene Vertreter** dieser **statischen Komponenten** angelegt.

  ◇ Die **Definition** (Speicherplatz-Reservierung) für die **statischen Datenkomponenten** kann
  - für alle Template-Klassen mittels einer **Template-Vereinbarung** gemeinsam formuliert werden oder
  - für jede Template-Klasse gesondert erfolgen (z.B. zur getrennten Angabe von Initialisierungswerten)

  ◇ **Beispiel** :

```
template <class T> class X
{ //  ...
  public :
    static T s;
  //  ...
};

template <class T> T X<T>::s;     // Template-Vereinbarung für stat. Datenkomp.

double X<double>::s = 5.26;

int main(void)
{
  X<int> xi;        // X<int> besitzt statische. Komponente s vom Typ int
  X<char*> xcp;     // X<const char*> besitzt statische Komponente s vom Typ
                    //   const char*

  X<int>::s=12;
  X<const char*>::s="Hallo !";

  cout << endl << "s von X<int>    : " << X<int>::s;
  cout << endl << "s von X<const char*>  : " << X<const char*>::s;
  cout << endl << "s von X<double> : " << X<double>::s;
  cout << endl;

  return 0;
`
```

- **Klassen-Templates als Template-Parameter**

  ◇ Eine generische Klasse (oder eine generische Funktion) kann auch **Parameter** besitzen, die selbst **Klassen-Templates** sind (→ **Verschachteln von Templates**).

  ◇ **Beispiel :** `Stack<Stack<int> > istackstack;  // Stack von int-Stacks`

  ◇ **Anmerkung** : Die beiden spitzen Klammern '**>**' müssen durch mindestens ein Leerzeichen getrennt werden, damit sie nicht als Operator '**>>**' interpretiert werden.

- **Ort von Template-Definitionen**

  Klassen- (und Funktions-)Templates dürfen **nur** auf der **globalen Ebene** oder **innerhalb** einer **Klasse** bzw eines **Klassen-Templates** (***member templates***) definiert werden.
  → Klassen-Templates können also auch **innere Klassen**, **nicht** jedoch **lokale Klassen**, beschreiben.

Generische Klassen (Klassen-Templates) in C++ (6)

- **Default-Argumente von Templates**

  ◇ Für Template-Parameter (sowohl von Klassen-Templates als auch Funktions-Templates) können auch **Default-Werte** festgelegt werden.
  Dies gilt sowohl für Typ-Parameter als auch für Nicht-Typ-Parameter.

  ◇ Die Festlegung von Default-Parametern erfolgt
  - entweder in der **Template-Definition**
  - oder in einer **Template-Deklaration** in einem Modul

  Beispiel :
  ```
  template <class T =double, int n = 256>
    class Array
    { /* ... */ };
  ```

  ◇ In der **gleichen Übersetzungseinheit** (Modul) darf für ein- und denselben Template-Parameter **nur eine Default-Festlegung** angegeben werden.

  ◇ Default-Werte für Template-Parameter müssen – wie bei Funktions-Parametern – immer **am Ende der Parameterliste** angegeben werden.
  Wird für einen Parameter ein Default-Wert festgelegt, so müssen für alle folgenden Parameter ebenfalls Default-Werte angegeben werden.

  Beispiel :
  ```
  template <class T1 = int, class T2> class B;      // Fehler !
  ```

  ◇ Bei der **Instantiierung** eines Templates mit Default-Parametern können dann die **entsprechenden aktuellen Parameter weggelassen** werden.

  Beispiel :
  ```
  Array<char> buffer;     // Typ : Array<char, 256>
  ```

  ◇ Existieren für alle Template-Parameter Default-Festlegungen und werden auch alle verwendet, kann bei der Angabe der Template-Klasse eine **leere Parameterliste** angegeben werden.

  Beispiel :
  ```
  Array<> polvec;          // Typ : Array<double, 256>
  ```

  Achtung : Die leere Parameterliste (öffnende und schliessende spitze Klammern) muss angegeben werden.

- **Anwendung von Klassen-Templates**

  ◇ Die Hauptanwendung von Klassen-Templates liegt in der Implementierung von **Container-Klassen**.
  Container-Klassen sind Klassen, deren Objekte zur **Verwaltung anderer Objekte** dienen. Die verwalteten Objekte sind entweder als Komponenten enthalten oder werden über enthaltene Pointer-Komponenten referiert.
  Typische Beispiele hierfür sind Stacks, Listen, assoziative Arrays, Mengen u.ä.

  ◇ Container-Klassen besitzen die **besondere Eigenschaft**, daß der **Typ der Objekte**, die sie verwalten, für die Definition der Klasse von **untergeordnetem Interesse** ist.
  Im **Vordergrund** stehen die **Operationen**, die mit den Objekten - unabhängig von ihrem Typ - auszuführen sind.
  Mit entsprechend definierten Klassen-Templates lassen sich dann gleichartige Container-Klassen für "Verwaltungs"-Objekte unterschiedlichsten Typs realisieren.

  ◇ In der **ANSI-C++-Standardbibliothek** sind mehrere derartige Klassen-Templates enthalten :
  u.a. **deque**, **list**, **vector, stack** (→ **STL** – **Standard Template Library**)

5 Überladen von Operatoren in C++

5.1 Allgemeines zum Überladen von Operatoren in C++

- Nur einige der C++-Operatoren lassen sich defaultmäßig auch auf Objekte selbstdefinierter Klassen anwenden.
  U.a. sind für Klassen bzw Klassenobjekte vordefiniert:

=	(Zuweisungs-Operator) (⇒ bitweise Kopie !)
&	(Adreß-Operator)
.	(Element-Operator)
.*	(Komponentenauswahl-Operator)
::	(Scope Resolution Operator, Bereichs-Operator)
sizeof	(Größenermitlungs-Operator)
new	(Objekt-Erzeugungs-Operator)
delete	(Objekt-Zerstörungs-Operator)

- Die meisten der übrigen C++-Operatoren können aber für die Anwendung auf Objekte selbstdefinierter Klassen umdefiniert werden. Auch ein Teil der o.a. für Klassen bzw Klassenobjekte vordefinierten Operatoren lassen sich umdefinieren.
  ⇒ Überladen von Operatoren (Operator Overloading)
  Dadurch besteht die Möglichkeit, Operationen mit Klassenobjekten in der von den einfachen Datentypen her gewohnten Art und Weise zu formulieren.
  Überladbar sind die folgenden Operatoren:

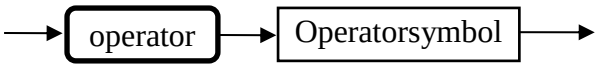
->	[]	()										
++	--	!	~	* (unär)	+ (unär)	- (unär)	& (unär)	new	delete			
->	*	/	%	* (binär)	+ (binär)	- (binär)						
<<	>>	<	<=	>	>=	==	!=	& (binär)	^	|	&&	||
=	*=	/=	%=	+=	-=	<<=	>>=	&=	^=	|=	,	

- Es lassen sich keine neuen Operator-Symbole einführen.
- Ein Überladen (Umdefinieren) der Operatoren für die Standard-Datentypen ist nicht möglich.
- Prinzipiell kann durch Überladen den C++-Operatoren eine beliebige Bedeutung gegeben werden, d.h. die durch sie realisierte Operation kann eine beliebige sein. Nach Möglichkeit sollte sich aber diese an die "standardmäßige" Bedeutung (Bedeutung für die Standard-Datentypen) anlehnen.
  Die beiden Schiebe-Operatoren (<< >>) sind in der C++-Standardbibliothek bereits defaultmäßig überladen: als I/O-Operatoren.
  Um Mißverständnisse zu vermeiden sollte man sie für Objekte selbstdefinierter Klassen auch nur in diesem Sinne überladen und ihnen keine andere Bedeutung geben.
- Die Anzahl der Operanden eines Operators, seine Priorität und seine Assoziativität lassen sich nicht verändern.

5.2 Operatorfunktionen in C++

- Die Wirkungen von Operatoren werden in C++ durch Funktionen beschrieben.
  ⇒ Operatorfunktionen (Operator Functions)
  Das Überladen von Operatoren besteht damit in der Definition entsprechender Operatorfunktionen.

- Funktionsname von Operatorfunktionen :



- Operatorfunktionen können definiert werden
  * als nichtstatische Member-Funktionen einer Klasse
    (Ausnahme: Operatorfunktionen für new und delete sind implizit immer statisch) oder
  * als "freie" ("globale") Funktionen, die dann i.a. für eine oder mehrere Klassen als Freund-Funktion deklariert werden.

- Die Operanden eines Operators werden der Operatorfunktion als Parameter übergeben.
  ⇒ Grundsätzlich besitzen Operatorfunktionen
  * für zweistellige (binäre) Operatoren zwei Parameter
  * für einstellige (unäre) Operatoren einen Parameter

  Wenigstens einer der Parameter muß ein Objekt eines Klassentyps oder eine Referenz darauf sein, d.h. die auschließliche Verwendung von Zeigern als Parameter ist nicht erlaubt.

  - Bei Operatorfunktionen, **die Member-Funktion einer Klasse sind**, ist der erste (linke) bzw. einzige Operand immer das aktuelle Objekt (Objekt über das die Funktion aufgerufen wird). Der entsprechende Parameter wird nicht explizit (sondern implizit durch den this-Pointer) übergeben. Für zweistellige Operanden besitzen diese Operatorfunktionen also nur einen expliziten Parameter, während sie für einstellige Operanden parameterlos sind.

Beispiel:

```
class Ratio {
     public:
        //……
        Ratio operator+(const Ratio&);   // Additionsoperator
        Ratio operator-();               // Negationsoperator
        //……
 };

 Ratio Ratio::operator+(const Ratio& y)  {
     Ratio temp;
     temp.zaehler=zaehler*y.nenner+nenner*y.zaehler;
     temp.nenner=nenner*y.nenner;
     return temp;
  }

 Ratio Ratio::operator-()  {
     Ratio temp=*this;
     temp.zaehler=-temp.zaehler;
     return temp;
  }
 // Ratio a,b,c; c=a+b;  führt zum Aufruf von :  a.operator+(b)
```

Operatorfunktionen in C++ (2)

- **Bei "freien" Operatorfunktionen** entspricht der erste Parameter immer dem linken und der zweite Parameter immer dem rechten Operanden. Benötigt eine "freie" Operatorfunktion direkten Zugriff auf private Komponenten einer Klasse, muss sie für diese Klasse als Freund deklariert werden.
  Beispiel :

```
class Ratio  {
  public:
      //¨¨¨
      friend Ratio operator+(const Ratio&, const Ratio&); // Additionsoperator
      friend Ratio operator-(const Ratio&);              // Negationsoperator
      //¨¨¨
};

Ratio operator+(const Ratio& x, const Ratio& y)  {     //freie Operatorfunktion
  Ratio temp;
  temp.zaehler=x.zaehler*y.nenner+x.nenner*y.zaehler;
  temp.nenner=x.nenner*y.nenner;
  temp.kuerze();
  return temp;
}

Ratio operator-(const Ratio& x)  {              //freie Operatorfunktion
  Ratio temp=x;
  temp.zaehler=-temp.zaehler;
  return temp;
}

// Ratio a,b,c; c=a+b;  führt zum Aufruf von :  operator+(a,b)
```

- Operatorfunktionen werden i.a. nicht explizit aufgerufen, sondern implizit bei der Auswertung entsprechender Ausdrücke. Ein expliziter Aufruf ist allerdings zulässig.
  ⇒ zu obigem Beispiel: c=operator+(a,b); // entspricht c=a+b;

- Weitere Eigenschaften von Operatorfunktionen:
  * Operatorfunktionen dürfen keine Default-Parameter besitzen
  * Operatorfunktionen können wie andere Funktionen überladen werden
    (⇒ unterschiedliche Signatur!)
    Ausnahme: Operatorfunktion für Operator delete

Beispiel:

```
class Ratio {
  public:
     //¨¨¨
     Ratio operator+(const Ratio&);    // Ratio + Ratio
     Ratio operator+(double);    // Ratio + double
     //¨¨¨
 };
```

  * Operatorfunktionen - außer operator=() - werden an abgeleitete Klassen vererbt.

Operatorfunktionen in C++ (3)

- **Operatorfunktionen als Member-Funktion oder "freie" Funktion ?**

Operatorfunktion als Memberfunktion (Methode):

* Die folgenden Operatoren **müssen** durch **Member-Funktionen** überladen werden:

=	(Zuweisung)	nichtstatische Member-Funktionen
[]	(Indizierung)	
()	(Funktionsaufruf)	
->	(Zeigerobjekt-Element)	
typ(), (typ)	(Werterzeugung, Konvertierung)	
new	(Speicherallokation)	statische (implizit) Member-Funktionen
delete	(Speicherfreigabe)	

* Außerdem sind **Member-Funktionen** für folgende Operatoren sinnvoll:

- **Zusammengesetzte Zuweisungen:** += -= *= /=
  Diese Operatoren benötigen als linken Operanden stets das aktuelle Objekt, weil sie den Zustand des aktuellen Objektes verändern.

- **Unäre Operatoren:** - ++ --
  Linken Operanden stets das aktuelle Objekt, ++ -- verändern zusätzlich das aktuelle Objekt

Die Verwendung - auch befreundeter - "freier" Funktionen wäre in diesen Fällen möglich, wäre aber zum einem umständlich und würde das OOP-Prinzip der Kapselung aufweichen, außerdem führen Member-Funktionen i.a. zu einem effizienteren Code

Operatorfunktion als freie Funktion

Freie (globale) Operatorfunktionen werden vorzugsweise dann eingestzt, wenn einer der folgenden Fälle vorliegt:

- Der Operator ist binär und in beiden Fällen „symmetrisch“ z.B die binären arithmetischen Operatoren + - * /

- Der Operator soll für eine fremde Klasse überladen werden, ohne diese Klasse zu verändern
  z.B der Operator << für die Klasse iostream

Beispiel "symmetrische" Typ-Mischung und Operator <<

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

class Ratio {
public:
  // ……
   Ratio operator+(const Ratio&);
   friend Ratio operator+(const Ratio&, double); //auch Member möglich
   friend Ratio operator+(double, const Ratio&); //friend notwendig

        // ……
        friend ostream operator<<(ostream &, const Ratio);
};

ostream operator<<(ostream& os, const Ratio r)
{
os << r.zaehler <<'/' << r.nenner <<endl;
return os;
}

int main(){
  Ratio x(3,5), y;

  y=x+3.7;        // auch bei Member-Funktion möglich
  cout << x << " +  3.7   = " << y <<endl;

  y=3.7+x;        // bei Member-Funktion nicht möglich
 cout <<  "3.7 +  " << x  <<"= " << y<< endl;

  return 0;
}
```

- **Referenzparameter und Referenz als Funktionswert**
  * Operatorfunktionen, die einen Operanden verändern muß der entsprechende Operand als Referenzparameter übergeben werden.
  * Auch für Operanden, die nicht verändert werden, kann ein Referenzparameter sinnvoll sein :
    - die Übergabe einer Referenz (Adresse!) ist i.a. effektiver als die Übergabe des Werts eines Objekts (Kopie jeder Daten-Komponente !)
    - Vermeidung von Problemen, die auftreten können, wenn die Kopie des als Wert übergebenen Parameters durch Destruktor zerstört wird
    - Wenn ein als Referenz übergebener Operand - verändert oder nicht verändert - auch Ergebnis der Operation sein soll, muß er auch als Referenz von der Operatorfunktion zurückgegeben werden.

5.3 Überladen des Zuweisungs-Operators in C++

- Der Zuweisungs-Operator = ist ein binärer Operator.
  Dem linken Operanden (der ein "lvalue operand" sein muß) wird der Wert des rechten Operanden, der durch die Zuweisungsoperation nicht verändert wird, übergeben.

- In C++ ist für jede Klasse ein Default-Zuweisungsoperator vordefiniert. Dieser führt eine komponenten- und damit byteweise Kopie des rechten Operanden in den linken Operanden durch. ($\Rightarrow$ Analogie zum Default-Copy-Konstruktor).
  Es gibt Fälle, bei denen eine strikte byteweise Kopie aber nicht gewünscht ist. Z.B. führt bei Klassen, die als Komponenten Pointer auf dynamisch allokierte Speicherbereiche enthalten, eine byteweise Kopie i.a. zu Fehlern (s. Copy-Konstruktor).
  Wenn man lediglich die Zulässigkeit der Zuweisung zwischen Objekten einer Klasse verhindern möchte, genügt es eine Operatorfunktion für den Zuweisungsoperator als „private“ zu deklarieren (eine Definition ist weder erforderlich noch sinnvoll).

- In vielen Fällen wird man aber einen klassenspezifischen Zuweisungsoperator mit vom Default-Operator abweichenden Verhalten benötigen.
  $\Rightarrow$ Definition einer klassenspezifischen Operatorfunktion operator=().

- Allgemeine Eigenschaften der Operatorfunktion operator=() :
  - Sie muß eine nichtstatische Member-Funktion sein
  - Sie wird nicht an abgeleitete Klassen vererbt
  - Sie kann überladen werden
    ($\Rightarrow$ verschiedene Typen des rechten Operanden)
  - Der rechte Operand wird als Parameter i.a. als Referenz auf ein konstantes Objekt übergeben
  - Der Funktionswert ist i.a. eine Referenz auf das Objekt, für den die Operatorfunktion aufgerufen wurde, d.h auf den linken Operanden (dadurch werden verkettete Zuweisungen und Tests nach Zuweisungen möglich und Probleme vermieden, die durch die Erzeugung einer temporären Kopie des Rückgabeobjektes bei Wertrückgabe entstehen können).

- Innerhalb der Zuweisungs-Operatorfunktion sollte als erstes überprüft werden, ob ein Objekt sich selbst zugewiesen wird (Überprüfung auf Identität des linken und rechten Operanden, durch Überprüfung auf Gleichheit der Adressen). In diesem Fall muß die Funktion das Objekt unverändert lassen.
  $\Rightarrow$ Vermeidung von Fehlern

  (sonst würde gegebenfalls Speicherplatz des Objekts freigegeben, dessen Daten anschließend zugewiesen werden sollen).

- Anmerkung: Zuweisung und Initialisierung sind unterschiedliche Operationen.

Überladen des Zuweisungs-Operators in C++ (2)

- Beispiel :

```
class String  {
 public:
   // ....
   String& operator=(const String&);   // Zuweisung eines String-Objekts
   String& operator=(const char *);    // Zuweisung eines char-Arrays
   // ....
 private:
   char *cp;
   int len;
};

String& String::operator=(const String& so) {
 if (this!=&so)  {                          // schützt gegen so=so
   if (len!=so.len)  {
     delete [] cp;
     len=so.len;
     if ((cp=new char[len+1])==NULL)  {           // Allokation eines neuen
       cout << "\nAllokations-Fehler\n";          // Speicherbereichs
       exit(1);
     }
   }
   strcpy(cp, so.cp);
 }
 return *this;
}

String& String::operator=(const char* s)  {
 if (len!=strlen(s)) {
   delete [] cp;
   len=strlen(s);
   if ((cp=new char[len+1])==NULL) {         // Allokation eines neuen
     cout << "\nAllokations-Fehler\n";            // Speicherbereichs
     exit(1);
   }
 }
 strcpy(cp, s);
 return *this;
}
// ....

int main()  {
 String a("Kraut ");
 String b="Blume : ";
 // ....
 a=b="Unsere Renten sind sicher!";
 // ....
 return 0;
}
```

5.4 Überladen des Indizierungs-Operators []

- Der Indizierungs-Operator ist ein binärer Operator.
  Standarmäßig ist er definiert für den Zugriff zu einer Komponente eines Arrays. Der linke Operand ist das Array (genauer: ein Pointer auf das erste Element des Arrays). Der rechte Operand muß ein arithmetischer Ausdruck sein, der den Index (laufende Nummer der gewünschten Array-Komponente) liefert.

- Für Klassen läßt er sich sinnvoll überladen
  - zur Indizierung von - statischen oder dynamischen - Arrays, die Komponenten der Klasse sind, wobei er mit zusätzlichen Funktionalitäten, wie z.B. Indexgrenzenüberprüfung oder Array-Vergrößerung versehen werden kann,
  - zur Realisierung indizierungsähnlicher Auswahl-Operationen an Objekten, die Ansammlungen (Mengen) mehrerer gleicher oder ähnlicher Elemente nicht als Arrays sondern in einer anderen Realisierungsform, z.B. als verkettete Listen, enthalten.

- Einige Eigenschaften der Operatorfunktion operator[]() :
  - Sie muß eine nichtstatische Member-Funktion sein
  - Der Parameter (rechter Operand, Index) kann von beliebigem Typ sein.
  - Sie kann überladen werden.
    Dies ermöglicht eine unterschiedliche Implementierung des Indizierungs-Operators für verschiedene Index-Typen

- Beispiel :

```
class String  {
 public:
    // ....
    char& operator[](unsigned);      // Überladen für Indizierung
    // ....
 private:
    char *cp;
    int len;
};

char& String::operator[](unsigned i)  {
 if (i>=len)  {
    cout << "\nIndexgrenzenüberschreitung !\n";
    exit(1);
 }
 return cp[i];
}

// ....
int main()
{ String b="wer";
 // ....
 b[1]='i';
 // ....
}
```

Demonstrationsprogramm zum Überladen des Indizierungs-Operators in C++

```
// Programm ASSARDEM            c:\irber\oop\bsp_cpp\k4_1
// Demonstrationsprogramm zum Überladen des Indizierungs-Operators
// Realisierung eines einfachen assoziativen Arrays
// In diesem Beispiel werden alle Wörter in der Einlesereihenfolge abgespeichert,  ist ein
   Wort
// bereits gespeichert,  wird es nicht mehr abgespeichert, sondern nur der Wortzähler erhöht.
// PS: Die bei weitem häufigste Umsetzung von assoziativen Arrays ist jedoch die Hashtabelle.
#include <iostream.h>
#include <string.h>
class Assar  {
   public :
    Assar(int=10);                    // Konstruktor
    int& operator[](const char*);     // Indizierungs-Operator
    void print_all();                      // Einfache Ausgabe aller Komponenten
   private :
    struct paar { char *wort; int zahl; } *vec;
    int max;
    int free;
    Assar(const Assar&);              // verhindert Kopieren bei Initialisierung
    Assar& operator=(const Assar&);  // verhindert Kopieren bei Zuweisung
};
Assar::Assar(int n)  {                     // Konstruktor
   max=n; free=0; vec=new paar[max];
}
int& Assar::operator[](const char *wp)  { // Indizierungs-Operator
   paar *pp=&vec[free-1];
   while ((pp>=vec) && (strcmp(wp, pp->wort)!=0)) pp--;
   if (pp<vec)  {                          // neuer String
     if (free==max) {                      // Vergrößerung des Arrays
          paar *neuvec=new paar[2*max];
          for (int i=0; i<max; i++) neuvec[i]=vec[i];
          delete[] vec;
          vec=neuvec;
          max=2*max;
       }
    pp=&vec[free++];                    // Belegung einer neuen Komponente
    pp->wort=new char[strlen(wp)+1];
      strcpy(pp->wort, wp);
      pp->zahl=0;
    }
   return pp->zahl;
}
void Assar::print_all()  {
   for (int i=0; i<free; i++)
    cout << vec[i].wort << " : " << vec[i].zahl << '\n';
}
int main(void)  {
   const int MAXL=256;
   char buffer[MAXL];
   Assar feld;
   while (cin >> buffer)       // Wortweises einlesen bis CTRL-Z
         feld[buffer]++;       // Indizierung über String
   feld.print_all();
    return 0;
   }
```

Aufruf und Ausgabe des C++-Programms ASSARDEM

```
   G:\>assardem
   der Euro wird weich
   das Versprechen der Bundesregierung der Euro werde so stark sein wie die D-Mark ist
   nicht mehr seriös vielmehr zeichnet sich deutlich ab daß die zukünftige europäische
   Gemeinschaftswährung nach allen Gesetzen der wirtschaftlichen Logik nicht so hart
   werden kann wie das Produkt der Bundesbank wenn die Währungsunion tatsächlich wie
   geplant eingeführt wird
   ^Z
der : 5
Euro : 2
wird : 2
weich : 1
das : 2
Versprechen : 1
Bundesregierung : 1
werde : 1
so : 2
stark : 1
sein : 1
wie : 3
die : 3
D-Mark : 1
ist : 1
nicht : 2
mehr : 1
seriös : 1
vielmehr : 1
zeichnet : 1
sich : 1
deutlich : 1
ab : 1
daß : 1
zukünftige : 1
europäische : 1
Gemeinschaftswährung : 1
nach : 1
allen : 1
Gesetzen : 1
wirtschaftlichen : 1
Logik : 1
hart : 1
werden : 1
kann : 1
Produkt : 1
Bundesbank : 1
wenn : 1
Währungsunion : 1
tatsächlich : 1
geplant : 1
eingeführt : 1
```

5.5 Überladen des Increment- und Decrement-Operators in C++

- **Notationsformen des Increment- bzw. Decrement-Operators**

  ◇ Die **unären Operatoren** ++ (**Increment**) und -- (**Decrement**) existieren für die Standard-Datentypen jeweils in **2 Formen** :

  ▻ **Prefix-Notation** : **++x**; Ausdruckswert : veränderter Operandenwert

  ▻ **Postfix-Notation**: **x++**; Ausdruckswert : alter Operandenwert

  ◇ In **früheren** C++-Versionen konnte beim Überladen dieser Operatoren **nicht** zwischen **Prefix**- und **Postfix**-Notation **unterschieden** werden.

  ◇ Im **ANSI/ISO-C++**-Standard ist jedoch eine **Unterscheidungsmöglichkeit** vorgesehen:

  ▻ Die **Prefix-Notation** wird - wie bei den anderen unären Operatoren - durch eine **Operatorfunktion** mit **einem Parameter** (der bei der Definition als Member-Funktion implizit als `this`-Pointer übergeben wird) realisiert.

  ▻ Die **Postfix-Notation** dagegen wird durch eine **Operatorfunktion** mit **zwei Parametern** (von denen bei der Definition als Member-Funktion der erste implizit als `this`-Pointer übergeben wird) realisiert.
  Der **zweite** Parameter ist ein **Dummy-Parameter**. Er muß vom Typ **int** sein.
  Beim Aufruf der Operatorfunktion wird für ihn i.a. der Wert 0 übergeben.

  ◇ Damit sich die überladenen Operatoren bezüglich Prefix- und Postfix-Notation wie die Standard-Operatoren verhalten – was sinnvollerweise der Fall sein sollte – , sollten sich die **beiden** jeweiligen Funktionen lediglich im **Rückgabewert** unterscheiden. Dabei ist zu berücksichtigen, dass die Standard-Increment-und Decrement-Operatoren in der **Prefix**-Notation einen ***Lvalue*** (→ Rückgabe einer Referenz) und in der **Postfix**-Notation einen ***Rvalue*** (→ Rückgabe eines Wertes) zurückgeben.

- **Beispiel** :

```
class Ratio
{ public:
    //  ...
    Ratio& operator++(void);     // Überladen Increment-Operator (Prefix)
    Ratio operator++(int);       // Überladen Increment-Operator (Postfix)
  //  ...
};

Ratio& Ratio::operator++(void)       // Prefix
{ zaehler+=nenner;
  return *this;                      // Rückgabe neuer Wert (als Referenz)
}

Ratio Ratio::operator++(int dummy)   // Postfix, dummy ist Dummy-Parameter
{ Ratio temp=*this;
  zaehler+=nenner;
  return temp;                       // Rückgabe alter Wert
}

int main(void)
{
  Ratio a(4,5), b, c;

  b=a++;       // a.operator++(0);   ==>   b ⇐ (4,5),   a ⇐ (9,5)
  c=++a;       // a.operator++();    ==>   c ⇐ (14,5),  a ⇐ (14,5)
  //  ...
```

- **Anmerkung :**

  Die **Prefix-Notation** realisiert i.a. eine **schnellere Operation** als die Postfix-Notation (zweimaliges Kopieren des Objekts) → sie sollte – wenn nur der Inc- bzw. Dec-Effekt benötigt wird – in der Regel **bevorzugt verwendet** werden.

5.6 Überladen des Funktionsaufruf-Operator in C++

- **Funktionsaufruf**

  ◇ Ein **Funktionsaufruf** hat die Form

  `expression(expression-list)`

  ◇ Er kann formal als **zweistellige** (binäre) Operation aufgefasst werden:
  - **`()`** ist der – binäre – **Funktionsaufruf-Operator**
  - **`expression`** ist der **linke Operand**.
    Er muß eine Funktion referieren oder einen Funktionspointer als Wert ergeben.
  - **`expression-list`** ist der **rechte Operand**.
    Er besteht aus der durch Kommata separierten Liste der aktuellen Parameter, die auch leer sein kann.

- **Überladen des Funktionsaufruf-Operators `()`**

  ◇ Auch der Funktionsaufruf-Operator lässt sich für Objekte selbstdefinierter Klassen überladen.
  → In diesem Fall ist der linke Operand keine Funktion (bzw Funktionspointer) sondern ein Objekt

  ◇ Die Operatorfunktion **`operator()`** muß eine **nichtstatische Memberfunktion** sein, die beliebig viele Parameter (auch gar keine) haben kann.
  Default-Werte für die Parameter sind **zulässig**.

  ◇ **Beispiel** :

```
class Gerade           // y= m*x + t
{
  public :
    Gerade(double, double=0);
    double operator()(double=0) const;
  private :
    double m_dM;       // Steigung m
    double m_dT;       // Achsenabschnitt t
};

// ------------------------------------------------------------------------
Gerade::Gerade(double m, double t)
{ m_dM=m;
  m_dT=t;
}

double Gerade::operator()(double x) const
{ return x*m_dM + m_dT;
}
// ------------------------------------------------------------------------

int main(void)
{ Gerade line(0.5, 2);
  double arg;
  cout << endl << "Achsenabschnitt : " << line() << endl ;
  while (cout << "? ", cin >> arg)
    cout << "Wert : " << line(arg) << endl;
  cout << endl;
  return 0;
}
```

  ◇ Die Operatorfunktion **`operator()`** kann für eine Klasse auch **mehrfach überladen** werden (unterschiedliche Parameterlisten !)

Funktionsaufruf-Operator in C++ (2)

- **Funktionsobjekte**

  ◇ Objekte von Klassen, für die der Funktionsaufrufoperator überladen ist, bezeichnet man als **Funktionsobjekte**.

  ◇ Es sind **Objekte**, die **wie Funktionen verwendet** werden können:
  Der Ausdruck

  `object(expression-list)`

  ist kein Aufruf der Funktion `object()`,
  sondern ein Aufruf der Memberfunktion `operator()()` für das Objekt `object`:

  `object.operator()(expression_list)`

- **Anwendung von Funktionsobjekten**

  ◇ Funktionsobjekte stellen häufig eine sinnvolle **objektorientierte Alternative zu freien Funktionen** dar.
  Insbesondere dann, wenn mehrere gleichartig strukturierte aber mit unterschiedlichen Kenngrößen (z.B. Koeffizienten von Polynomen) arbeitende Funktionen benötigt werden.
  Statt diese Kenngrößen bei jedem Funktionsaufruf zusätzlich zu den eigentlichen Funktionsparametern zu übergeben, werden sie durch einen Konstruktor in Datenkomponenten von Funktionsobjekten abgelegt.

  ◇ Das Überladen der Operatorfunktion `operator()()` ist weiterhin auch immer dann sinnvoll, wenn für eine Klasse nur **eine einzige** oder eine wichtige **überwiegend angewendete Funktionalität** existiert.
  Beispiele : - Implementierung von einfachen **Iteratoren** (→ Programmbeispiel s. Kap "Iteratoren")
  - **Applikatorklassen** als eine Möglichkeit zur Realisierung von Manipulatoren mit Parametern

  ◇ Beispiele anderer gebräuchlicher Anwendungen des überladenen Funktionsaufruf-Operators sind:
  - Einsatz als **Substring-Operator**
  - **Indizierung mehrdimensionaler Arrays** bzw. von Datenstrukturen, die als mehrdimensionale Arrays behandelt werden können

  ◇ **Beispiel** zur Indizierung eines mehrdimensionalen Arrays

```
class IntMatrix
{ public :
    IntMatrix(unsigned, unsigned);
    int& operator()(unsigned, unsigned) const;
  private :
    unsigned m_uRows;
    unsigned m_uCols;
    int* m_piMatr;
};

IntMatrix::IntMatrix(unsigned rows, unsigned cols)
{ m_uRows=rows;
  m_uCols=cols;
  m_piMatr=new int[m_uRows*m_uCols];
  // Initialisierung aller Komponenten mit 0
}

int& IntMatrix::operator()(unsigned row, unsigned col) const
{ row=row%m_uRows;
  col=col%m_uCols;
  return m_piMatr[row*m_uCols+col];
}
int main(void)
{
  IntMatrix matr(5, 10);
  matr(2,3)=25;
  // ...
}
```

5.7 Überladen des Operators -> in C++ (1)

- **Interpretation des Operators `->`**

  ◇ **Standardmäßig** wird der Operator **`->`** als **binärer Operator** verwendet.
  Er dient zum Zugriff zu Komponenten eines Objekts.
  Dabei muß der **linke Operand** ein **Pointer auf ein Objekt** und der **rechte Operand** der **Name einer Komponente** dieses Objekts sein.
  Der "**Wert**" eines derartigen Ausdrucks ist die dadurch **referierte Objekt-Komponente**.
  → **dereferenzierender Komponenten-Operator**.

  ◇ Der Operator lässt sich für Klassen so **überladen**, dass er direkt auf Objekte (statt auf Objekt-Pointer) angewendet werden kann. :
  In dieser überladenen Form stellt er einen **unären Operator** dar, der bei einem Objekt als Operanden – direkt oder indirekt – einen Objekt-Pointer zurückliefern muß.

  → Ein Ausdruck **`object->name`**
  wird interpretiert als **`(object.operator->())->name`**

  ◇ Der zurückgelieferte Pointer muß nicht das Objekt referieren, auf das der Operator angewendet wurde.
  Vielmehr wird er sich bei realen Anwendungen im allgemeinen auf ein anderes Objekt beziehen.
  Die durch `name` bezeichnete Objektkomponente muß dann eine Komponente dieses Objekts und nicht eine des Operanden-Objekts sein.
  → Der Komponentenzugriff wird an ein anderes Objekt **delegiert**.
  → **Delegations-Operator**.

- **Operatoratorfunktion `operator->`**

  ◇ Eine den Operator **`->`** überladende Funktion muß eine **nichtstatische Memberfunktion ohne Parameter** sein.

  ◇ Sie sollte einen **Pointer auf ein Objekt** als **Funktionswert** zurückgeben.

  ◇ Falls die Funktion statt eines Objekt-Pointers ein Objekt (oder eine Referenz auf ein Objekt) zurückgibt, wird die **Auswertung** eines **`->` - Ausdrucks rekursiv** fortgesetzt: Für das zurückgelieferte Objekt wird wiederum eine Operatorfunktion `->` aufgerufen. Falls für dessen Klasse keine definiert ist, endet die Auswertung fehlerhaft.

  ◇ Ein expliziter Aufruf der Operatorfunktion `operator->` muß ohne Parameter erfolgen (das entspricht **einem Operanden**). Eine Verwendung von `->` in einem Ausdruck dagegen erfordert **zwei Operanden**.

```
class Abc
{ public :
    // ...
    X* operator->();          // X sei eine andere Klasse
  private :
    // ...
};

void func(Abc p)
{
  X* px1 = p.operator->();    // ok
  X* px2 = p->;               // Fehler !
  // ...
```

- **Beispiele für Anwendungen**
  ▻ Realisierung von "***smart***" **Pointern**.
  Hierbei handelt es sich um Objekte, die wie Pointer verwendet werden können, bei jedem Komponentenzugriff aber eine zusätzliche Funktionalität realisieren. (Kapselung der eigentlichen Pointer in dem Objekt)
  ▻ **Ausdehnung der Fähigkeiten** einer vorhandenen Klasse auf eine andere Klasse, in die sie eingekapselt wird, ohne Vererbung sondern mittels **Delegation**.

Operator -> in C++ (2)

- **Beispiel zur Delegation mittels Operatorfunktion `operator->`**

```
class Worker
{ public :
    Worker(int=0);
    // ...
    void doWork();
  private :
    // ...
    int m_id;
};

#include <iostream>
using namespace std;

inline Worker::Worker(int id)
{
  m_id=id;
}

inline void Worker::doWork()
{
  cout << "\nIch, der Worker mit id=" << m_id << ", arbeite gerade !\n";
}
// ----------------------------------------------------------------------------
class Chief
{ public :
    // ...
    void hireWorker(Worker&);
    Worker* operator->();
  private :
    Worker* myEmpl;
};

inline void Chief::hireWorker(Worker& fred)
{
  myEmpl=&fred;
}

inline Worker* Chief::operator ->()
{
  return myEmpl;
}

// ----------------------------------------------------------------------------

int main(void)
{ Chief moi;
  Worker tu(1);
  moi.hireWorker(tu);
  moi->doWork();         // Delegation von doWork() an Worker-Objekt
  return 0;
}
```

Ausgabe des Programms :

```
Ich, der Worker mit id=1, arbeite gerade !
```

5.8 Konvertierungsfunktionen in C++

- **Funktionen zur Typkonvertierung :**
  - ◇ **Konstruktoren**, die mit **einem Parameter** aufgerufen werden können, können als **Funktionen zur Konvertierung** des Parameter-Typs **in den Klassen-Typ** betrachtet werden.
  - ◇ Zur **entgegengesetzten Konvertierung** - **aus dem Klassentyp** in einen anderen Typ - dienen **Konvertierungs-funktionen** (*conversion functions*).
  - ◇ Im Prinzip handelt es sich bei **einer Konvertierungs-Funktion** um eine **Operatorfunktion** zum **Überladen beider Formen der** einfachen **Typkonvertierungs-Operatoren** (Cast-Operator `(typ)` und Werterzeugungs-Operator `typ()`) sowie des Operators **`static_cast`**.

- **Eigenschaften einer Typkonvertierungsfunktion :**
  - ◇ Sie muß eine **nichtstatische Member-Funktion** sein.
  - ◇ Da die Typkonvertierungs-Operatoren **unäre Operatoren** sind, hat sie **keine expliziten Parameter**.
  - ◇ Der Typ ihres Rückgabewertes ist implizit durch den Ziel-Typ, der nach dem Schlüsselwort `operator` anzugeben ist, festgelegt. Eine **explizite Festlegung des Funktionstyps** erfolgt daher **nicht**.

    → **Allgemeine Syntax** für die **Funktionsdeklaration :**

    ```
    operator typ();        // typ ist der Ziel-Typ
    ```
  - ◇ Der Ziel-Typ **`typ`** kann ein **beliebiger Typ**, auch ein anderer Klassentyp, sein.
  - ◇ Eine Konvertierungsfunktion wird **nicht nur** bei **expliziter Anwendung** eines **Typkonvertierungs-Operators** sondern auch in Fällen **impliziter Typkonvertierung** aufgerufen.

- **Beispiel :**

```
class Ratio
{ public:
    // ...
    operator double(void);     // Konvertierungsfunktion Ratio --> double
    // ...
};
// ---------------------------------------------------------------------------

Ratio::operator double(void)
{
  return (double)zaehler/nenner; // --> ((double) zaehler)/nenner
}

// ...
/ ---------------------------------------------------------------------------

int main(void)
{
  Ratio a(3,-2), b(4,9);
  double z, u, v, w;
  // ...
  z=(double) a;          // Cast-Operator    --> Aufruf von a.operator double()
  w=double(b);           // Werterzeug.-Op   --> Aufruf von b.operator double()
  u=static_cast<double>(b); // static_cast   --> Aufruf von b.operator double()
  v=a;                   // impliz.Typ-Konv. --> Aufruf von a.operator double()
  // ...
}
```

5.9 Anmerkungen zur impliziten Typkonvertierung in C++ (1)

- **Implizite Typkonvertierungen :**

  Stimmen bei einer **Zuweisung** oder **Initialisierung** die **Typen** des **Ziel-Objekts** und des **Quell-Wertes nicht überein**, versucht der Compiler eine **automatische - implizite - Typkonvertierung** des Quell-Wertes in den Typ des Ziel-Objekts vorzunehmen.

  Anmerkung: Diese Typkonvertierung findet auch bei **Parameterübergaben** in Funktionsaufrufen und in **Ausdrücken**, die ja in Aufrufe von Operatorfunktionen umgesetzt werden, statt
  (Die Übergabe von Wertparametern stellt eine Erzeugung und Initialisierung temporärer Objekte dar)

- **Regeln für die implizite Typkonvertierung**

  ◇ Für die implizite – automatische - Typkonvertierung werden die folgenden - vereinfacht dargestellten - **Regeln** in der **angegebenen Reihenfolge** angewendet (**absteigende Priorität**):

  1. Anwendung **trivialer Typumwandlungen**
     (z.B. Array-Name → Pointer, Funktionsname → Pointer, `T` → `T&`, `T&` → `T`, `T` → `const T`, `T*` → `const T*`)
  2. Anwendung der **informationserhaltenden Standard-Typumwandlungen**
     - Integral Promotion :
       `char`, `short`, `enum`, Bitfeld → `int`,
       `unsigned char`, `unsigned short`, `unsigned` Bitfeld → `(unsigned) int`
     - Umwandlung `float` → `double`
  3. Anwendung der **übrigen Standard-Typumwandlungen**
     - Standard-Typumwandlungen von C
       (z.B. `int` → `double`, `int` → `unsigned`, usw)
     - Umwandlung von Referenzen und Pointer auf abgeleitete Klassen in Referenzen und Pointer auf Basisklassen
  4. Anwendung **benutzerdefinierter Typumwandlungen**
     - **Konstruktoren**
     - **Konvertierungsfunktionen**

  ◇ Eine Typkonvertierung kann **über mehrere Umwandlungsschritte** erfolgen.
  Existieren **mehrere Konvertierungswege**, wählt der Compiler den **kürzesten** aus.
  Existieren **zwei oder mehr gleichberechtigte Wege**, erzeugt er eine **Fehlermeldung**.

- **Implizite Anwendung benutzerdefinierter Typumwandlungen**

  Es gelten die folgenden Regeln:

  ◇ Sie werden **nur dann** versucht, wenn **keine** der **Standard-Typumwandlungen** zum **Erfolg** führt.

  ◇ An einer Typkonvertierung darf **maximal eine benutzerdefinierte Typumwandlung** beteiligt sein,
  Standard-Typumwandlungen können zusätzlich ohne Einschränkung angewendet werden.

  ◇ Es darf **nur einen Konvertierungsweg** geben, an dem **eine benutzerdefinierte Typumwandlung beteiligt** ist (unterschiedliche Weglängen spielen hierbei keine Rolle). → Konvertierung muß eindeutig sein.

  ◇ **Konstruktoren** und **Konvertierungsfunktionen** sind **gleichberechtigt**.

Anmerkungen zur impliziten Typkonvertierung in C++ (2)

- Die **automatische benutzerdefinierte Typkonvertierung** kann z.B. dazu genutzt werden, um **typgemischte Ausdrücke** unter Beteiligung von Objekten selbstdefinierter Klassen zu ermöglichen.

  Statt die jeweilige Operatorfunktion mehrfach - für die zulässigen Typkombinationen - zu überladen, benötigt man **nur** jeweils **eine Operatorfunktion** und **adäquate Konstruktoren**.
  Soll eine **"symmetrische" Typ-Mischung** möglich sein, so muß die Operatorfunktion auch hier eine **"freie"** – gegebenenfalls befreundete – **Funktion** sein.

  Beispiel :

```
class Ratio
{ public:
    // ...
    Ratio(double);                                  // alternativer Konstruktor
    // ...
    friend Ratio operator+(const Ratio&, const Ratio&);
  // ...                                                    // nicht überladen
};

// ...

int main(void)
{
  Ratio a(4,9), b, c;
  // ...
  b=3.7+a;                          // Aufruf von operator+(Ratio(3.7), a)
  c=a+4.3;                          // Aufruf von operator+(a, Ratio(4.3))
  // ...
}
```

- **Probleme**

  Funktionen zur **automatischen Typkonvertierung** können auch **Probleme** erzeugen.
  Wird im obigen Beispiel zusätzlich eine Konvertierungsfunktion

  `Ratio::operator double()`

  die ein `Ratio`-Objekt in einen `double`-Wert umwandelt, definiert, sind die **gemischten Additions-Ausdrücke nicht** mehr **eindeutig**:

  Beispielsweise kann **`3.7 + a`** dann mittels Typkonvertierung
  als **`Ratio(3.7) + a`**
  und als **`3.7 + double(a)`**
  interpretiert werden.

  → **Beide Konvertierungen** verwenden eine **benutzerdefinierte Umwandlung**, was **nicht zulässig** ist.

  => Konstruktoren und Konvertierungsfunktionen sind nur sehr wohl durchdacht zu definieren.

  Insbesondere muß darauf geachtet werden, daß **keine zyklischen Typkonvertierungen** möglich werden.
  Die ist z.B. der Fall,

  - wenn - wie im obigen Beispiel - **in einer Klasse** ein **Konstruktor** und eine **Konvertierungsfunktion** für genau **entgegengesetzte Typumwandlungen** definiert werden,
  - oder in **zwei Klassen** jeweils ein **Konstruktor** vorhanden ist, der eine **Typumwandlung aus der jeweiligen anderen Klasse** vornimmt.

Ergänzungen zur Typkonvertierung in C++

- **Weiteres Beispiel zur Mehrdeutigkeit bei automatischer Typkonvertierung**

  Die **Konvertierung** eines Objekts der **Klasse X** in ein Objekt der **Klasse T** kann erreicht werden
  - ▻ entweder durch die **Konvertierungsfunktion** : `X::operator T()` `// Member von X`
  - ▻ oder durch die **Konstruktor-Funktion** : `T::T(X)` `// Member von T`

  Wenn **beide Funktionen definiert** sind, sind **automatische Typkonvertierungen** ebenfalls **nicht mehr eindeutig** → Compiler-Fehler

  Beispiel :

```
class X
{ // ...
  operator T();            // Konvertierungsfunktion
  // ...
};

class T
{ // ...
  T(X);                   // Konstruktor
  T(const T&);            // Copy-Konstruktor
  // ...
};

void main(void)
{
  X xobj;
  T tobj=xobj;           // Konstruktor oder Konvertierungsfunktion?
   // ...
}
```

- **Vermeidung der Problematik von Mehrdeutigkeiten**

  Wegen der **Gefahr von Mehrdeutigkeiten** ist es häufig **besser**, durch **Vermeiden von Konvertierungsfunktionen** automatische Typkonvertierungen zu reduzieren.
  Trotzdem **benötigte Typkonvertierungen** lassen sich i.a. **problemlos** durch **explizite Aufrufe** entsprechend definierter **normaler Member-Funktionen** realisieren.
  Diesen Funktionen sollte man dann entsprechend **aussagekräftige Namen** geben.

  Beispiel :

```
class Ratio
{ public:
    // ...
    double asDouble(void) const; // Funktion zur expliziten
  // ...                          Typkonvertierung
 }

// ...
void main(void)
{ Ratio a(5,13);
  double z = a.asDouble();     // expliziter Aufruf der Typkonvertierung
  // ...
}
```

6 Vererbung

6.1 Allgemeines zur Vererbung und zu abgeleiteten Klassen

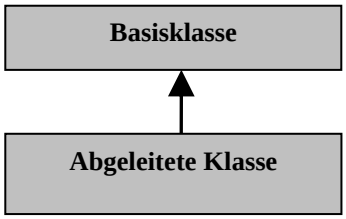
- Vererbung ist eines der Grundkonzepte der OOP.
  Sie ermöglicht es, neue Klassen aus vorhandenen Klassen abzuleiten, wobei die Eigenschaften der Ausgangsklassen übernommen und um zusätzliche Eigenschaften ergänzt werden. Diese Ergänzung kann in einem Hinzufügen neuer Komponenten - sowohl Datenkomponenten (Attribute) als auch Funktionskomponenten (Methoden) und/oder im Ändern einzelner Methoden bestehen.
  Statt eine Klasse völlig neu zu implementieren, wird auf bereits Vorhandenes zurückgegriffen und nur das was neu hinzukommt oder sich ändert als Ergänzung definiert. Durch Vererbung lassen sich also Gemeinsamkeiten zwischen Klassen erfassen und in hierarchische Beziehungen einordnen.

- Eine vorhandene Ausgangsklasse, von der andere Klassen abgeleitet werden, wird Basisklasse (base class) oder Superklasse (superclass) oder Oberklasse genannt.
  Eine von einer Basisklasse abgeleitete Klasse (derived class) wird auch als Subklasse (subclass) oder Unterklasse bezeichnet.
  Eine abgeleitete Klasse kann selbst wieder Basisklasse für weitere Ableitungen sein.
  $\Rightarrow$ mehrstufige Ableitung $\Rightarrow$ Klassenhierarchien.
  Die ürsprüngliche Basisklasse ist für mehrstufig abgeleitete Klassen eine indirekte Basisklasse.

- Einfache Vererbung (single inheritance):
  Ableitung einer Klasse von einer Basisklasse. Die abgeleitete Klasse ist eine Spezialisierung und/ oder Erweiterung der Basisklasse.
- Mehrfachvererbung (multiple inheritance):
  Ableitung einer Klasse von mehreren Basisklassen. Die abgeleitete Klasse vereint die Eigenschaften ihrer Basisklassen (und kann zusätzliche Änderungen/Erweiterungen enthalten).

- Zwischen abgeleiteter Klasse und ihrer (ihren) Basisklasse(n) besteht eine "ist ..."- ("is a ..."-) Beziehung :
  Die abgeleitete Klasse erbt die Eigenschaften ihrer Basisklasse(n). Ein Objekt der abgeleiteten Klasse besitzt daher - neben neu hinzugefügten Eigenschaften - die Eigenschaften der Basisklasse(n), d.h. es kann damit auch als ein Objekt der Basisklasse(n) betrachtet werden. Beispiel: Basisklasse Pflanze, abgeleitete Klasse Blume, jede Blume ist auch eine Pflanze.

- In Klassen-Diagrammen werden Beziehungen zwischen abgeleiteten Klassen und ihren Basisklassen i.a. durch gerichtete Pfeile, mit der Pfeilspitze an der Basisklasse, dargestellt :



- Vorteile der Vererbung :

  * Einsparung von Code

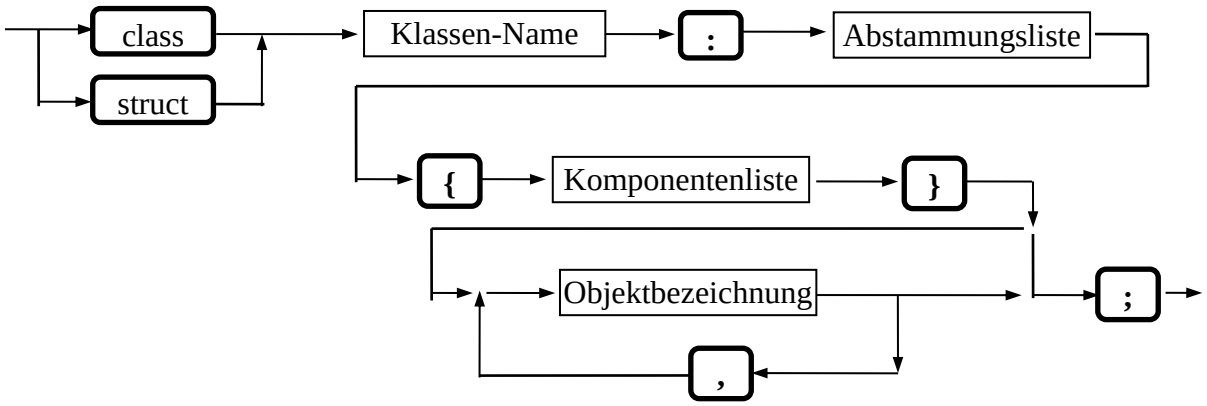
  * Konsistenter Code: gemeinsame (gleiche) Dinge stehen nur an einer einzigen Stelle und müssen nur einmal entwickelt, verändert und übersetzt werden.

  * Voraussetzung für Laufzeit-Polymorphie.

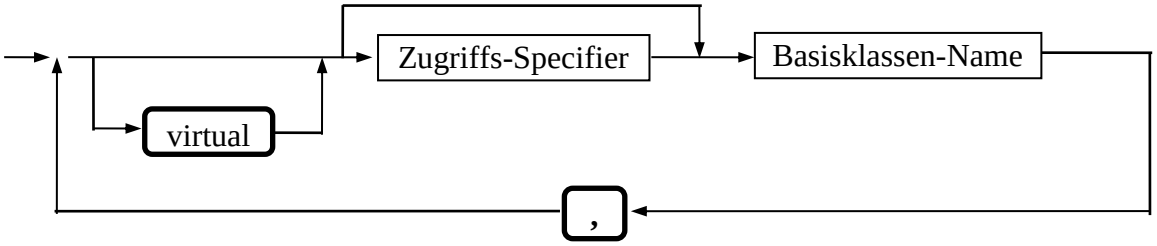
6.2 Definition von abgeleiteten Klassen in C++

- Ergänzung der Klassendefinition um eine Abstammungsliste (base-list), in der die Basisklassen aufgelistet sind. In der Komponentenliste werden nur die neu hinzukommenden Komponenten und gegebenenfalls die geänderten (neu definierten) Member-Funktionen angegeben.

- Syntax (vereinfacht)



Abstammungsliste (base-list) :



Zugriffs-Specifier (access-specifier) : private, protected, public

- Beispiel :

```
//Definition der Klasse Counter

   class Counter {
      public:
         Counter(int=0);
                     void reset();
         void preset(int);
         void count();
         void display() const;

      protected:
         int m_count;
    };
```

```
//Definition der Klasse UpDownCounter
class UpDownCounter : public Counter  {  // Klasse "UpDownCounter" abge-
  public:                                 // leitet von Klasse "Counter"
    UpDownCounter(int=0, int=1);
    void count();                        //geänderte Member-Funktion
    void  display() const;               // "
    void setdirection(int dir){m_dir=dir;} //zusätzliche neue Member-Funktion
     int getdir(){return m_dir;}
  private:
    int m_dir;       // zusätzliche neue Daten-Komponente für Zaehlrichtung
};

//Implementierung der Klasse Counter
Counter::Counter(int count):m_count(count) {}
void Counter::reset(){    m_count=0;}       //Definition der Methode reset
void Counter::count() {   m_count++; }      //Definition der Methode count
void Counter::display()const
{    cout<<"Stand:"<< m_count<<endl; }    //Definition der Methode display

//Implementierung der Klasse UpDownCounter
UpDownCounter::UpDownCounter(int count, int dir)
:Counter(count),  m_dir(dir)//Basisklassenintialisierer ,Elementinitialisierer
{ }                         //Rumpf für restliche Aktionen

void UpDownCounter::count()                //Redefinition von count
{
     switch(m_dir)
     {   case 1:    Counter::count();      //vorwärts zaehlen
                    break;
         case -1:   m_count--;             //rückwärts zählen
                    break;
         default: ;                       //Tue nichts
     }
     return;
}

void UpDownCounter::display() const   //Redfinition von display
{
      Counter::display();   //Zählerstand ausgeben mit Methode der Basisklasse
      cout << "Richtung: " << m_dir <<endl;  //Richtung ausgeben
}

//Applikation
int main()
{
     UpDownCounter updc;
     updc.count();
     updc.display();
     cout << endl;

     updc.setdir(-1);
     updc.count();
     updc.display();
     cout <<endl;
     return 0;
}
```

Definition von abgeleiteten Klassen in C++ (2)

- Eine abgeleitete Klasse besitzt zusätzlich zu den für sie neu definierten Komponenten auch alle Datenkomponenten und Member-Funktionen ihrer Basisklasse(n) mit Ausnahme der Konstruktoren, Destruktoren und Operatorfunktionen operator=(). Diese werden nicht vererbt. Der Compiler fügt für Objekte der abgeleiteten Klasse die Komponenten der Basisklasse automatisch zu den neu hinzugekommenden Komponenten hinzu. ⇒ In jedem Objekt der abgeleiteten Klasse ist ein Objekt der Basisklasse als eine Art Teil-Objekt enthalten.
  Wird ein Objekt einer abgeleiteten Klasse als Objekt einer Basisklasse betrachtet, so ist die "Sicht" auf dieses Teil-Objekt beschränkt.

  Beispiel : Objekt der Klasse UpDownCounter

	Datenkomponenten	Member-Funktionen
Teil-Objekt der Klasse Counter (geerbte Komponenten)	act_count	reset() preset() count() display()
zusätzliche Komponenten	direction	count() display() setdirection()

- Freund-Funktionen werden nicht an abgeleitete Funktionen vererbt.
  (Sie sind keine Klassen-Komponenten)

- Namenskonflikte
  In einer abgeleiteten Klasse darf man einer neu hinzugefügten Komponente den **gleichen Namen** geben, den schon eine Komponente einer - direkten oder indirekten - Basisklasse hat. In einem derartigen Fall wird der Name in der abgeleiteten Klasse neu definiert. Die Komponente gleichen Namens der Basisklasse wird aber auch geerbt. ⇒ Beide Komponenten existieren in der abgeleiteten Klasse.
  Mit dem Komponenten-Namen allein kann nur die neue Komponente erreicht werden.
  Die von der Basisklasse geerbte gleichnamige Komponente ist überdeckt.
  Sie kann nur über ihren voll-qualifizierten Namen (Komponentenname ergänzt um den Klassennamen mit dem Scope Resolution Operator) angesprochen werden.

6.3 Zugriffsrechte bei abgeleiteten Klassen in C++

- In der Abstammungsliste einer abgeleiteten Klasse kann für jede Basisklasse ein Zugriffs-Specifier angegeben werden. Dieser Vererbungs-Zugriffs-Specifier legt zusammen mit den ursprünglichen Zugriffsrechten in der Basisklasse die Zugriffsrechte zu den von der Basisklasse geerbten Komponenten fest.

  Falls für eine Basisklasse kein Vererbungs-Zugriffs-Specifier angegeben ist, gilt als Default:
  private, wenn die abgeleitete Klasse als class definiert ist,
  public, wenn die abgeleitete Klasse als struct definiert ist.

  Allgemein gilt :

  * Zu private-Komponenten der Basisklasse hat die abgeleitete Klasse (d.h. haben deren spezielle Member-Funktionen und Freund-Funktionen) grundsätzlich keinen direkten Zugriff.

Zugriffsrechte bei abgeleiteten Klassen in C++ (2)

- Zugriffsrecht "protected":

  * Dieses Zugriffsrecht ist speziell im Hinblick auf die Vererbung geschaffen worden:
    Zu protected-Komponenten einer Klasse können sowohl die eigenen Member-Funktionen (und Freund-Funktionen) als auch die Member-Funktionen (und Freund-Funktionen) der von der Klasse abgeleiteten Klassen zugreifen. Außerhalb der Klasse und der abgeleiteten Klassen sind diese Komponenten nicht zugänglich (Ausnahme: Freund-Funktionen).

  * Das Zugriffsrecht "protected" entspricht also einem auf die abgeleiteten Klassen ausgedehntem Zugriffsrecht "private".

  * Beispiel :

```
class Counter  {
  public:
    // ....
  protected:
    unsigned long act_count;
};

class UpDownCounter : public Counter  {
  public:                                      // Zugriff zu act_count
    void upcount() { act_count++; }            // nur zulässig, weil
    void downcount() { act_count--; }          // protected in Counter
    friend void show_count(UpDownCounter&);
};

// ....
#include <iostream >
using namespace std;
void count_aus(UpDownCounter& cnt)   {         // "freie" Funktion
  cout << cnt.act_count;   // unzulässig, kein Zugriff zu act_count
}

void show_count(UpDownCounter& cnt)  {         // Freund-Funktion
  cout << cnt.act_count;          // zulässig, show_count() ist Freund
}

// Anmerkung zur Funktion show_count() :
// Bei einem Parameter vom Typ Counter& wäre der Zugriff zu
// act_count unzulässig, da show_count Freund von UpDownCounter
// aber nicht von Counter ist.
```

  * Anmerkung: Wenn der Konstruktor einer Klasse protected ist, können nur abgeleitete Klassen und Freund-Funktionen Objekte dieser Klasse erzeugen.

6.4 Typkonvertierungen zwischen abgeleiteten und Basis-Klassen in C++

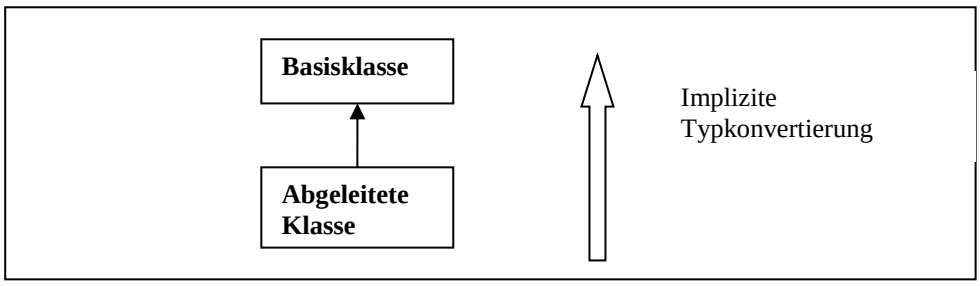
- Durch Vererbung wird - sofern public-Ableitung vorliegt - eine automatische (implizite) Typwandlung von einem Objekt einer abgeleiteteten Klasse in ein Objekt seiner Basisklasse definiert.
  Beispiel :

```
class Y { /* ¨¨¨¨ */ };
class X : public Y { /* ¨¨¨¨ */ };
void f(void) {
       X x;
       Y y;
       y=x;     // implizite Typwandlung !
}
```

- Für eine Klasse, die - über beliebig viele Stufen - von einer Basisklasse eindeutig und durchgängig public abgeleitet ist, gilt:
  Ein Pointer bzw. eine Referenz auf ein Objekt der abgeleiteten Klasse wird - wo erforderlich - implizit in einen Pointer bzw. eine Referenz auf ein Objekt der Basisklasse umgewandelt. Über den umgewandelten Zeiger (bzw. Referenz) sind natürlich nur die Komponenten der Basisklasse erreichbar.
  Beispiel :

```
class Y { public: int a; /* ¨¨¨¨ */ };
class X : public Y { public: int b; /* ¨¨¨¨ */ };
void f()  {
    X d;
    d.a=d.b=1;
    Y* p=&d;         // implizite Typkonvertierung !
    p->a++;          // auf Komponente a ist zugreifbar
    // p->b++;       // auf Komponente b ist nicht über p zugreifbar
}
```

  Die umgekehrte Typkonvertierung (Pointer auf Basisklasse in Pointer auf abgeleitete Klasse) ist nur explizit möglich. Es liegt in der Verantwortung des Programmierers, sicherzustellen, daß diese Typkonvertierung auch sinnvoll ist, d.h. der Basisklassenzeiger auch tatsächlich auf ein Objekt der abgeleiteten Klasse zeigt.



6.5 Konstruktor und Destruktor bei abgeleiteten Klassen in C++

- Konstruktoren und Destruktoren werden grundsätzlich nicht vererbt. ⇒ Sie müssen für abgeleitete Klassen - falls benötigt - neu definiert werden. Gegebenenfalls werden ein Default-Konstruktor und ein Default-Destruktor vom Compiler generiert.

- Beim Anlegen eines Objekts einer abgeleiteten Klasse werden neben dem Konstruktor dieser Klasse auch - und zwar zuerst - die Konstruktoren der Basisklassen aufgerufen.
  ⇒ Konstruktoren werden in der Reihenfolge der Ableitung aufgerufen.
  Beispiel :

```
class A { /* ¨¨¨¨ */ };
class B :public A { /* ¨¨¨¨ */ };
class C : public B { /* ¨¨¨¨ */ };
// Reihenfolge der Konstruktor-Aufrufe : A(),B(), C()
```

- Destruktoren werden in der umgekehrten Konstruktor-Reihenfolge aufgerufen :
  Zuerst für die abgeleitete Klasse, dann für die Basisklassen.

- Initialisierung von Komponenten der Basisklassen :
  * Nur durch die Konstruktoren der jeweiligen Basisklasse möglich.
  * Explizit benötigte Initialisierungswerte werden dem Konstruktor der abgeleiteten Klasse - zusammen mit den für ihn bestimmten Initialisierungswerten - als Parameter übergeben und sind von diesem an den Konstruktor der Basisklasse weiterzureichen ("chain of argument passing").
  * Die Weitergabe der Parameter erfolgt mittels der - im Funktionskopf der Konstruktor-Definition möglichen - Initialisierungsliste (s. Konstruktoren mit Initialisierungsliste).
    Analog zu Datenkomponenten mit ihren Initialisierungswerten können in dieser Liste auch Basisklassen mit ihren Initialisierungswerten aufgeführt werden.

  * **Allgemeiner Aufbau einer Konstruktor-Definition** :
    klassenname::klassenname(parameterliste) : initialisierungsliste { /* Rumpf der Konstruktor-Funktion */ }

  Allgemeiner Aufbau von initialisierungsliste :

```
basisklassenname(parameter, ...), ...,
komponentenname(parameter, ...), ...
```

  * Als Klasseneinträge in der Initialisierungsliste sind nur direkte Basisklassen zulässig. Bei mehrstufiger Ableitung müssen Initialisierungswerte gegebenenfalls stufenweise zur jeweils nächsten direkten Basisklasse weitergegeben werden. Der Auftritt eines Klassennamens in einer Initialisierungsliste führt zum Aufruf des betreffenden Konstruktors der Klasse. Falls dieser wiederum eine Initialisierungsliste mit Klassen-Einträgen besitzt wird zuerst deren jeweiliger Konstruktor aufgerufen usw.

C++ -Demo-Programm zu Konstruktor/Destruktor bei abgeleiteten Klassen

```
// ------------------------------------------------------------------------
// Programm DCKONST - Demo zu Konstr./Destruktor bei abgel. Klassen
// ------------------------------------------------------------------------
  #include <iostream >
   using namespace std;

  class Base  {
   public:
     Base(int);
     ~Base();
     void showi() const;
   private:
     int i;
  };

  class Derivat : public Base  {
  public:
     Derivat(int, int);
     ~Derivat();
     void showk() const;
  private:
     int k;
  };

  Base::Base(int m) :i(m)
  { cout << "Im Konstruktor von Base !\n"; }

  Base::~Base()      { cout << "Im Destruktor von Base !\n"; }

  void Base::showi() const { cout << "i : " << i << '\n'; }

  Derivat::Derivat(int a, int b) : Base(b), k(a)  { // Weitergabe von b an Base
       cout << "Im Konstruktor von Derivat !\n";
  }

   Derivat::~Derivat()  { cout << "Im Destruktor von Derivat !\n"; }

   void Derivat::showk()const { cout << "k : " << k << '\n'; }

   int main()  {
     Derivat obj(13, 29);
     obj.showi();
     obj.showk();
     return 0;
   }
```

Programm-Aufruf und -Ausgabe :

Im Konstruktor von Base !
Im Konstruktor von Derivat !
i : 29
k : 13
Im Destruktor von Derivat !
Im Destruktor von Base !

7 Polymorphie

7.1 Frühes und spätes Binden

- Der Aufruf einer Methode kann als Übermitteln einer Botschaft an ein konkretes Objekt betrachtet werden.

  Polymorphie (Polymorphismus) ("Ein Interface - mehrere Methoden") ermöglicht es, daß mehrere - unterschiedliche - Methoden (Member-Funktionen) den gleichen Namen haben können.
  Die Auswahl der über einen Funktionsnamen tatsächlich aufgerufenen Funktion, d.h. die Zuordnung einer Methode (Member-Funktion) zu einer Botschaft (Funktionsaufruf, Funktionsname) wird in der OOP als Binden bezeichnet.

- Frühes Binden (early binding, statisches Binden, static binding)
  Die Zuordnung erfolgt bereits zur Compilezeit. Der Compiler ermittelt die Startadresse der tatsächlich aufgerufenen Funktion. Dies ist der Normalfall in C++.
  ⇒ Realisierung einer "Compilezeit-Polymorphie" mittels überladener Funktionen.

- Spätes Binden (late binding, dynamisches Binden, dynamic binding)
  Die Zuordnung erfolgt erst zur Laufzeit. Die Startadresse der tatsächlich aufgerufenen Funktion wird nicht vom Compiler ermittelt, sondern erst zur Laufzeit in Abhängigkeit vom jeweiligen Ziel-Objekt festgelegt.
  ⇒ Realisierung einer "Laufzeit-Polymorphie".

  Benötigt wird spätes Binden, wenn durch Zeiger (oder Referenzen) auf Objekte von Basisklassen tatsächlich - häufig erst zur Laufzeit ausgewählte - Objekte von abgeleiteten Klassen referiert werden und Member-Funktionen der Basisklasse durch die abgeleitete Klasse umdefiniert worden sind. In derartigen Fällen sollen i.a. die Member-Funktionen der abgeleiteten Klasse und nicht die - gleichnamigen - der Basisklasse aufgerufen werden. Durch frühes Binden erfolgt aber eine Zuordnung zu Funktionen der Basisklasse.

  Beispiel zur Notwendigkeit späten Bindens :

```
class Auto { public: void print_gefahren() { /* ¨¨¨ */ } /* ¨¨¨ */ };
class AmphFahrz : public Auto {
    public: void print_gefahren() const { /* ¨¨¨ */ }
   // ¨¨¨
};

void main() {
  AmphFahrz amphi;
  Auto *ap;               // Pointer auf Objekte der Basisklasse
  // ¨¨¨
  ap=&amphi;              // Basisklassen-Ptr zeigt auf abgel. Klasse
  ap->print_gefahren();         // Aufruf von Auto::print_gefahren()
  // ¨¨¨                  // obwohl ap auf Objekt der Klasse AmphFahrz
}                         // zeigt (⇒ hier kein spätes Binden).
```

  Spätes Binden wird in C++ mittels virtueller Funktionen realisiert.

7.2 Virtuelle Funktionen in C++

- Allgemeines
  * Für virtuelle Funktionen findet spätes Binden statt.
  * Nur nichtstatische Member-Funktionen können virtuell sein.
  * Konstruktoren können nicht virtuell sein.
  * Destruktoren können virtuell sein.
  * Eine Klasse mit wenigstens einer virtuellen Funktion wird polymorphe Klasse bzw polymorpher Typ genannt.

- Vereinbarung virtueller Funktionen
  * Eine Member-Funktion wird zu einer virtuellen Funktion, indem man ihrer Vereinbarung innerhalb der Klassendefinition den FunktionsSpezifizierer (function specifier) virtual voranstellt.
    Beispiel:

```
class Base  {
  public:
     virtual void f1();              // Definition außerhalb
     virtual int f2(int);            // der Klassendefinition
     // ¨¨¨¨
}
```

Redefinition (Überschreiben) virtueller Funktionen

- Eine virtuelle Funktion muss in der abgeleiteten Klasse nicht neu definiert werden.
  In diesem Fall erbt die abgeleitete Klasse die virtuelle Funktion der Basisklasse.
- Typischerweise wird in der abgeleiteten Klasse jedoch eine "eigene Version" der virtuellen Funktion definiert. Diese ist dann den speziellen Fähigkeiten der abgeleiteten Klasse angepasst
- Gegenüber einer normalen Redefinition ist jedoch folgendes zu beachten:
- Die Funktion in der abgeleiteten Klasse muss im **Namen**, in der **Signatur** (Anzahl, Typ und Reihenfolge der Parameter) sowie auch im **Funktionstyp** mit der Funktion der Basisklasse übereinstimmen (Unterschied zu Überdecken).
- Diese Redefinition einer virtuellen Funktion mit identischer Signatur nennt man auch **"überschreiben"**(overriding):
  Es gilt die Regel "virtuell bleibt virtuell". D.h. die neue Version einer virtuellen Funktion ist automatisch wieder virtuell. Die Angabe von "virtual" in ihrer Vereinbarung ist nicht notwendig, aber zulässig

Redefinition mit neuer Signatur:

Eine Unterscheidung in der Signatur bewirkt kein **Überschreiben**, sondern ein **Überdecken** der Funktion der Basisklasse. Diese ist in der abgeleiteten Klasse nur über ihren vollqualifizierten Namen aufrufbar.

Eine weitere Funktionsvereinbarung in der abgeleiteten Klasse mit veränderter Signatur **überlädt** - nicht virtuell - die überschreibende virtuelle Funktion in der abgeleiteten Klasse. Eine Unterscheidung allein im Funktionstyp ist ein Fehler.

Beispiel:

```
class Base  {
   public:
      virtual void f1();              // Definition außerhalb
      virtual int f2(int);            // der Klassendefinition
      // ....
 };

 class Derivat : public Base  {
     public:
      void f1();                      // überschreibt Base::f1()
      int f2(int, char);              // überdeckt Base::f2()
      int f1(int);                    // überlädt f1()
      int f1();                       // Fehler !!!
   // ....
 };
```

* **Eine virtuelle Funktion muss für die Basisklasse entweder definiert oder als rein-virtuell (pure virtual) deklariert werden.**

* Syntax für Deklaration einer rein-virtuellen Funktion :

```
virtual typ func_name(parameter_liste) = 0;
```

Beispiel : `virtual void rotate(int) = 0;`

Eine rein virtuelle Funktion besitzt keine Implementierung. In der virtuellen Methodentabelle (VMT) der Klasse wird der Nullzeiger eingetragen. Damit wird die Syntax der Deklaration abgebildet.

Virtuelle Funktionen in C++ / Polymorphie (2)

- Anwendung virtueller Funktionen
  * Durch die Vereinbarung einer virtuellen Funktion in der Basisklasse wird das Interface der Funktion (Parameter und Funktionstyp) definiert. Die **Umdefinition** (das **Überschreiben**) in einer abgeleiteten Klasse erzeugt eine spezielle Methode, die über dieses Interface aufgerufen werden kann.

* Für eine virtuelle Funktion wird spätes Binden angewendet, wenn sie über einen **Pointer** oder eine **Referenz** auf die **Basisklasse** aufgerufen wird.
  Die Funktion bekommt polymorphes Verhalten.
  In einem derartigen Fall wird erst zur Laufzeit in Abhängigkeit von der tatsächlichen Klasse des Objekts, das durch den Pointer bzw die Referenz referiert wird, die Startadresse der aufzurufenden Funktion ermittelt, d.h. die Interpretation des Funktionsaufrufs hängt vom **Typ des referierten Objekts** ab.
  Bei nicht-virtuellen Funktionen hängt die Interpretation des Funktionsaufrufs dagegen nur vom **Typ des Pointers bzw. der Referenz** ab.

Beispiel : **Polymorphie über Basisklassenzeiger**

```
class Auto  {public: virtual void print_gefahren() const { /* ¨¨¨ */ } /* ¨¨¨ */
};
class AmphFahrz : public Auto { public: void print_gefahren() const { /* ¨¨¨ */
} /* ¨¨¨ */ };

void main()  {
    AmphFahrz amphi;
    Auto *ap;                   // Pointer auf Objekte der Basisklasse
    // ¨¨¨
    ap=&amphi;                  // Basisklassen-Ptr zeigt auf abgel. Klasse
    ap->print_gefahren();       // Aufruf von AmphFahrz::print_gefahren()
    // ¨¨¨                      // print_gefahren() ist virtuell. Aufruf
                                // über Basisklassenptr ⇒ späte Bindung
}
```

Beispiel : **Polymorphie über Referenz auf Basisklasse**

```
 class Auto  {public: virtual void print_gefahren() const { /* ¨¨¨ */ }
/* ¨¨¨ */ };
 class AmphFahrz: public Auto { public: void print_gefahren() const
{ /* ¨¨¨ */   } /* ¨¨¨ */ };

void show_gefahren(const Auto&  auto) { //Parameter ist Referenz auf
                                        //Basisklasse
      auto.print_gefahren();      //Aufruf von AmphFahrz::print_gefahren()
}

//Polymorphie über Referenz auf Basisklasse
int main()  {
    AmphFahrz amphi;
    show_gefahren(amphi);
    return 0;
}
```

7.3 Virtueller Destruktor

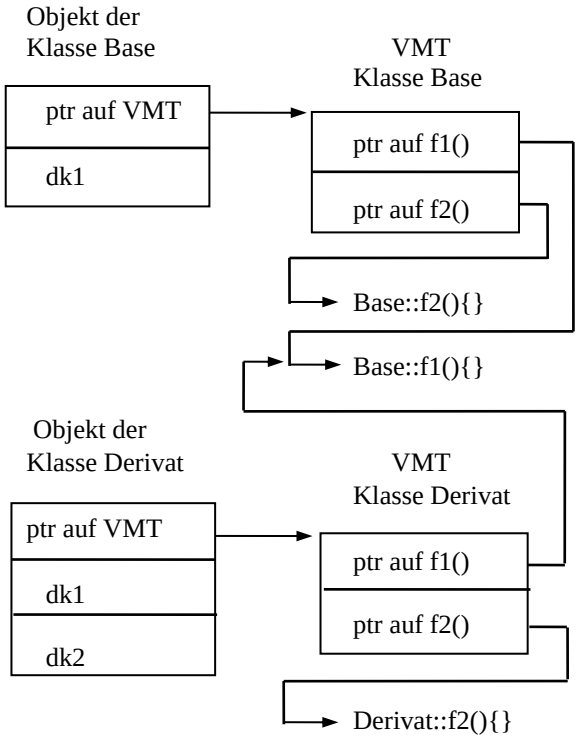
* Ein **virtueller Destruktor** wird benötigt, wenn sichergestellt werden soll, dass bei der Freigabe eines Objekts einer abgeleiteten Klasse mittels `delete` über einen Basisklassen-Pointer auch der Destruktor der abgeleiteten Klasse aufgerufen wird. Dies ist z.B. notwendig, wenn in dem Objekt der abgeleiteten Klasse zusätzlich dynamischer Speicher allokiert worden ist. Wenn für die Basisklasse ein nicht-virtueller Destruktor definiert wurde (auch der Default-Destruktor ist nicht-virtuell), wird in einem derartigen Fall nur der Basisklassen-Destruktor aufgerufen.

* Die Virtualität ihrer Member-Funktionen beeinflußt die Eignung einer Klasse für die Vererbung. Eine Klasse ist nur dann generell für die Vererbung geeignet, wenn
  - alle Member-Funktionen, die in abgeleiteten Klassen **überschrieben** werden könnten, als virtuell vereinbart sind,
  - sie einen **virtuellen Destruktor** besitzt (gegebenenfalls muß ein sonst nicht-virtueller Default-Destruktor explizit als virtuell definiert werden)

```
// v_destr.cpp
// Ohne virtuellem Destruktor würde bei delete über Basisklassenpointer
// nicht der Destruktor von Derived ausgeführt.
#include <iostream>
#include <cstring>
using namespace std;
class Base
{
  public:
   Base()
   {  cout << "Konstruktor der Klasse Basis\n"; }

   virtual ~Base()
   {  cout << "Destruktor der Klasse Basis\n"; }
};
class Derived : public Base
{
  private:
    char* name;
  public:
    Derived( const char* n)
    {
      cout << "Konstruktor der Klasse Abgeleitet\n";
      name = new char[strlen(n)+1];
      strcpy(name, n);
    }
    ~Derived()                // implizit virtuell
    {
      cout << "Destruktor der Klasse Abgeleitet\n";
      delete [] name;
    }
};

int main()
{     Base *bPtr = new Derived("DEMO");
     cout << "\nAufruf des virtuellen Destruktors!\n";
     delete bPtr;         //nur ok, falls Destruktor von Base virtuell ist
     return 0;
}
```

7.4 Virtuelle Funktionen in C++ /VMT

- Realisierung des späten Bindens bei virtuellen Funktionen
  Enthält eine Klasse virtuelle Funktionen, so wird für diese **Klasse** vom Compiler eine Virtuelle-Methoden-Tabelle (VMT) angelegt. In dieser sind die Anfangsadressen aller virtuellen Funktionen zusammengefaßt.
  Für jede **abgeleitete Klasse** wird ebenfalls eine VMT angelegt. Für nicht überschriebene Funktionen werden die entsprechenden Einträge aus der VMT der Basisklasse übernommen, für umdefinierte Funktionen werden die Basisklassen-VMT-Einträge durch die Anfangsadressen der überschreibenden Funktionen ersetzt.
  Die Einträge für eine überschreibende Funktion (in der VMT der abgeleiteten Klasse) und für die dazugehörige überschriebene Funktion (in der Basisklassen-VMT) stehen an gleicher Stelle.
  Jedes Objekt einer Klasse mit virtuellen Funktionen enthält - als zusätzliche implizite Komponente - **einen Pointer auf die VMT seiner Klasse**. Bei Objekten abgeleiteter Klassen ist dieser Pointer im Teil-Objekt der Basisklasse enthalten.
  Der Aufruf einer virtuellen Funktion über einen Basisklassen-Pointer (bzw -Referenz) erfolgt indirekt über den VMT-Pointer.
  ⇒ Der Aufruf einer virtuellen Funktion dauert etwas länger als der Aufruf einer nicht-virtuellen Funktion.
  Beispiel:




```
class Base  {
   public:
      // ¨¨¨¨
      virtual void f1() {/*¨¨¨¨*/}
      virtual void f2() {/*¨¨¨¨*/}
   private:
      int dk1;
};

class Derivat : public Base  {
{ public:
      // ¨¨¨¨
    void f2() { /*¨¨¨¨*/ } // überschrieben
   private:
   int dk2;
};

int main(void) {
   Base b;
   Derivat d;
   Base *bptr=&b;
   bptr->f1();     // Aufruf von bptr->VMT[0]() ⇒ Base::f1()
   bptr->f2();     // Aufruf von bptr->VMT[1]() ⇒ Base::f2()
   bptr=&d;
   bptr->f1();     // Aufruf von bptr->VMT[0]() ⇒ Base::f1()
   bptr->f2();     // Aufruf von bptr->VMT[1]() ⇒ Derivat::f2()
   return 0;
}
```

Virtuelle Funktionen in C++ (5)

- **Default-Parameterwerte**
  Default-Parameterwerte werden nicht zur Laufzeit sondern beim Compilieren eingesetzt.
  ⇒ Beim Aufruf einer virtuellen Funktion über einen Basisklassen-Pointer (oder -Referenz) werden immer die Default-Parameterwerte, die für die virtuelle Funktion in der Basisklasse festgelegt sind, übergeben.
  ⇒ Wenn eine überschreibende virtuelle Funktion in der abgeleiteten Klasse andere Default-Parameterwerte als die überschriebene Funktion in der Basisklasse besitzt, werden bei ihrem Aufruf über einen Basisklassen-Pointer falsche Werte übergeben.

  Beispiel:

```
class Base  {
   public:
      // ¨¨¨¨
      virtual void f1(int i=0) { /*¨¨¨¨*/ }
      // ¨¨¨¨
 };

 class Derivat : public Base  {
   public:
      // ¨¨¨¨
      void f1(int i=1) { /*¨¨¨¨*/ }
      // ¨¨¨¨
 };

 int main()  {
   Derivat d;
   Base* bptr=&d;
   // ¨¨¨¨
   bptr->f1();     // Aufruf von Derivat::f1(0) ⇒ falscher Def.-Param.
   d.f1();         // Aufruf von Derivat::f1(1) ⇒ richtiger Def.-Param.
   // ¨¨¨¨
 }
```

==> **Überschreibende und überschriebene Funktionen sollten dieselben Default-Parameterwerte besitzen.**

Demonstrationsprogramm zu virtuellen Funktionen in C++

```
// ------------------------------------------------------------------
// Programm VFCTBSP
// ------------------------------------------------------------------
  #include <iostream>
 using namespace std;
  class Num {
  public:
    Num(int i=0) { wert=i;}
    virtual void shownum() {
    cout <<"wert dezimal   : " << dec << wert << '\n'; }
  protected:
    int wert;
  };

  class HexNum : public Num  {
  public:
    HexNum(int i=0) : Num(i) { }
    void shownum()  {
    cout <<"wert sedezimal : " << hex << wert << '\n'; }
  };

  class OctNum : public Num  {
  public:
    OctNum(int i=0) : Num(i) { }
    void shownum()  {
    cout <<"wert oktal     : " << oct << wert << '\n'; }
  };

  int main()  {
    Num* baseptr[3];
    Num wd(10);
    HexNum wh(511);
    OctNum wo(63);
    baseptr[0]=&wd;
    baseptr[1]=&wh;                // implizite Typwandlung HexNum* ⇒ Num*
    baseptr[2]=&wo;                // implizite Typwandlung OctNum* ⇒ Num*
    for (int i=0; i<3; i++)
      baseptr[i]->shownum();

    baseptr[1]->Num::shownum();
    wd=wh;                         // implizite Typwandlung HexNum ⇒ Num
    wd.shownum();
   return 0;
  }
```

Aufruf und Ausgabe des Programms :

```
F:\RT\CPP\VORL>vfctbsp
wert dezimal   : 10
wert sedezimal : 1ff
wert oktal     : 77
wert dezimal   : 511
wert dezimal   : 511
F:\RT\CPP\VORL>
```

7.5 Laufzeit-Typinformation in C++

• Einführung

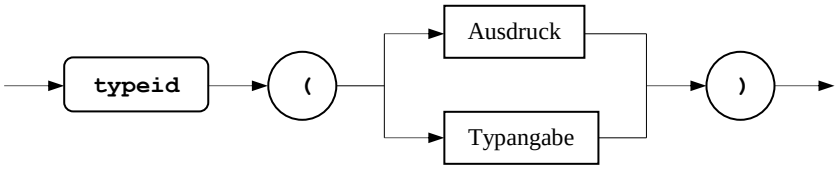
Ein **Zeiger** bzw eine **Referenz** auf ein Objekt einer **abgeleiteten Klasse** kann implizit oder explizit in einen Zeiger bzw eine Referenz auf ein Objekt einer - eindeutigen - **Basisklasse umgewandelt** werden.
Dadurch wird es möglich, mittels Basisklassen-Pointern bzw -Referenzen Objekte unterschiedlichen Typs - allerdings aus derselben Vererbungshierarchie - zu verwalten. Das bedeutet, daß der gleiche Basisklassen-Pointer (bzw -Referenz) dynamisch änderbar auf Objekte unterschiedlichen Typs zeigen kann.
Manchmal ist es wünschenswert, während der Laufzeit den **tatsächlichen Typ** des Objekts, auf das ein Basisklassen-Pointer bzw -Referenz zeigt, zu ermitteln.
→ **Laufzeit-Typinformation** (*Runtime Type Information* = **RTTI**)

• typeid-Operator

◇ **unärer** Operator

◇ ermöglicht die **Ermittlung von Typinformationen während der Laufzeit**

◇ **Syntax :**



◇ **Beispiele :** **typeid(*baseptr[1])**
typeid(int)

◇ Der **Wert** eines **typeid-Ausdrucks** ist vom **Typ const type_info&**

→ Ein typeid-Ausdruck liefert als Ergebnis eine Referenz auf ein Objekt der Klasse type_info, durch das der Typ des Operanden-Ausdrucks bzw der Operanden-Typangabe repräsentiert wird.

Die Klasse **type_info** ist Bestandteil der ANSI/ISO-C++-Standardbibliothek
. Sie ist in der Header-Datei **<typeinfo>** (bzw <typeinfo.h>) definiert.

Der **Compiler** legt **für jeden Datentyp ein Objekt dieser Klasse** an.

Wenn der **Operanden-Ausdruck** eine **Referenz** oder ein **dereferenzierter Pointer != NULL** auf Objekte einer **polymorphen Klasse** ist, ist das **Ergebnis** eine **Referenz** auf das **type_info-Objekt**, das den **tatsächlichen Typ** des aktuell referierten vollständigen Objekts referiert.
→ Ermittlung des **dynamischen** (zur Laufzeit änderbaren) **Typs**

Ist der **Operanden-Ausdruck** ein **dereferenzierter NULL-Pointer** auf Objekte einer polymorphen Klasse wird die **Exception bad_typeid** geworfen.

Für **jeden anderen Typ des Operanden-Ausdrucks** (sowie für eine **Typangabe als Operand**) ist das Ergebnis eine **Referenz** auf das **type_info-Objekt**, das diesen **(statischen) Typ** repräsentiert.

⇒ eine (dynamische) **Laufzeit-Typinformation läßt sich nur für Objekte polymorpher Klassen** ermitteln.

• Hinweis :

In **Visual-C++** muß die Unterstützung der Laufzeit-Typinformation durch Setzen eines **Compiler-Schalters** explizit **aktiviert** werden.
Menu : **Projekt → Einstellungen** → Reiter : **C++** → Kategorie : **Programmiersprache C++**→ Checkbox : **RTTI aktiv**.

Laufzeit-Typinformation in C++(2)

- **Beispiele für das Ergebnis eines `typeid`-Ausdrucks:**

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

class Fahrzeug { /* ... */ };                  // nicht-polymorphe Klasse

class LandFahrz : public Fahrzeug              // polymorphe Klasse
{ public:
    virtual void fahren(float) { /* ... */ }
    // ...
};

class Auto : public LandFahrz { /* ... */ };

class Fahrrad : public LandFahrz { /* ... */ };

class Boot : public Fahrzeug  { /* ... */ };


void main(void)
{
  Fahrzeug *pclFahr;
  LandFahrz *pclLandF;
  LandFahrz clLaFz;
  Auto a;
  LandFahrz& rLaFz=a;

  pclLandF=new Fahrrad;
  typeid(*pclLandF);    // --> type_info-Objekt von Fahrrad   (dynamisch)
  pclLandF=&a;
  typeid(*pclLandF);    // --> type_info-Objekt von Auto      (dynamisch)
  typeid(rLaFz);        // --> type_info-Objekt von Auto      (dynamisch)

  pclFahr=&clLaFz;
  typeid(*pclFahr);     // --> type_info-Objekt von Fahrzeug  (statisch)
  pclFahr=new Boot;
  typeid(*pclFahr);     // --> type_info-Objekt von Fahrzeug  (statisch)
  typeid(clLaFz);       // --> type_info-Objekt von LandFahrz (statisch)
  typeid(LandFahrz);    // --> type_info-Objekt von LandFahrz (statisch)
}
```

- **Realisierung der Ermittlung der (dynamischen) Laufzeit-Typinformation**

  Für **jede polymorphe Klasse** wird ein **Pointer** auf das die Klasse repräsentierende **`type_info`-Objekt** als **zusätzlicher Eintrag** mit in die **virtuelle Methoden-Tabelle** (VMT) aufgenommen.
  Damit ist dieses Objekt über den **VMT-Pointer** erreichbar.
  Da eine VMT nur für polymorphe Klassen existiert, ist die Ermittlung der (dynamischen) Laufzeit-Typinformation auf Objekte derartiger Klassen beschränkt.

Laufzeit-Typinformation in C++ (3)

- **Die Klasse `type_info`**
  - ◇ Objekte dieser Klasse **repräsentieren Typen**
  - ◇ Die Klasse ist in der C++-Header-Datei **`<typeinfo>`** (bzw `<typeinfo.h>`) definiert.
  - ◇ Für **jeden Datentyp** legt der Compiler **ein Objekt** dieser Klasse an.
    Dieses enthält **implementierungsabhängige Datenkomponenten** zur Speicherung des **Typnamens** sowie eines **codierten Wertes**, der es gestattet, Typen in eine **Sortierreihenfolge** anzuordnen sowie zwei Typen auf **Gleichheit** zu überprüfen.
    Die Klasse überlädt die **Operatoren `==`** und **`!=`** und definiert eine **Memberfunktion zur Ermittlung des Typnamens** sowie eine **Memberfunktion zum Vergleich des "Reihenfolgekriteriums"** zweier Typen.
  - ◇ In **ANSI/ISO-C++** ist folgende **prinzipielle Definition** der Klasse vorgesehen :

```
class type_info
{ public:
    virtual ~type_info();
    bool operator==(const type_info& rhs) const;
    bool operator!=(const type_info& rhs) const;
    bool before(const type_info& rhs) const;
    const char* name() const;
  private:
    // implementierungsabhängige Datenkomponenten zur
    // Speicherung des Typnamens und eines "Reihenfolgekriteriums"
    type_info(const type_info& rhs);
    type_info& operator=(const type_info& rhs);
};
```

  - ◇ Da der **Copy-Konstruktor** und die **Zuweisungsoperator-Funktion** dieser Klasse **`private`** sind, lassen sich `type_info`-Objekte **nicht kopieren**.

- **Beispiele zur Anwendung der Memberfunktionen der Klasse `type_info`** :

```
#include <typeinfo>
using namespace std;
{ //  ...
  LandFahrz *apclFuhrpark[ANZ];
  int iAnzVelo=0;
  //  ...
  for (int i=0; i<ANZ; i++)
    if (typeid(*apclFuhrpark[i]) == typeid(Fahrrad))
      iAnzVelo++;
  //  ...
}

{ //  ...
  LandFahrz *pclFahr;
  //  ...
  cout << typeid(*pclFahr).name();
  //  ...
}
```

7.6 Der Typkonvertierungsoperator `dynamic_cast` in C++

- **`dynamic_cast`**

  ◇ Dieser Typkonvertierungsoperator ermöglicht **sichere Typkonvertierungen innerhalb von Klassenhierarchien**; insbesondere eine **sichere Rückwandlung** eines **Pointers** (Referenz) auf **Basisklasse** in **Pointer** (Referenz) auf **abgeleitete Klasse**.
  Eine eventuelle **`const`-**Eigenschaft läßt sich mit ihm **nicht entfernen**.

  ◇ Der Ausdruck **`dynamic_cast<T>(e)`**

  bewirkt die Konvertierung des Ausdrucks **`e`** in den Typ **`T`**.

  Der **Zieltyp** **`T`** muß ein **Pointer** oder eine **Referenz** auf eine vollständig definierte Klasse bzw der Typ **`void*`** sein. Entsprechend muß der **Quellausdruck** **`e`** ein **Pointer** auf ein Klassen-Objekt oder ein **Lvalue** eines Klassentyps sein.

  ◇ **Folgende Fälle** sind **zulässig:**

  a) Der **Quellausdruck** **`e`** ist ein Pointer auf ein Objekt bzw ein Lvalue einer - von einer Basisklasse `B` **abgeleiteten - Klasse** **`D`.** Dabei muß `B` eine zugreifbare (`public`) und eindeutige Basisklasse von `D` sein.
  `T` ist ein Pointer bzw eine Referenz auf diese **Klasse** **`B`.**
  In diesem Fall ist das **Ergebnis** ein Pointer bzw eine Referenz auf das im **`D`-Objekt enthaltene Teil-Objekt der Klasse** **`B`.**
  → dieser Fall **entspricht** der **impliziten Standardkonvertierung**

  Beispiel :

```
class B { /* ... */ };

class D : public B { /* ... */ };

// ...
D clD;
B* pclB1 = dynamic_cast<B*>(&clD);
B* pclB2 = &clD;                    // pclB1 == pclB2 !
```

  b) Der **Quellausdruck** **`e`** ist ein Pointer auf ein Objekt einer **polymorphen Klasse** und **`T`** ist der Typ **`void*`.**
  In diesem Fall ist das **Ergebnis** ein Pointer auf das **vollständige** durch **`e`** tatsächlich referierte Objekt.

  c) Der **Quellausdruck** **`e`** ist ein Pointer auf ein Objekt bzw ein Lvalue einer **polymorphen Klasse** **`B`** und **`T`** ist **nicht** der Typ **`void*`.**
  In diesem Fall wird mittels einer **Laufzeit-Typprüfung** des tatsächlich referierten Objekts **geprüft**, ob der Quellausdruck **`e`** in den Zieltyp **`T`** **umgewandelt** werden kann :

  ▻ Ist **`T`** ein Pointer bzw eine Referenz auf eine **von** **`B`** **`public`** **abgeleitete Klasse** **`D`** und wird **durch** **`e`** tatsächlich ein **Objekt dieser Klasse** **`D`** oder einer **von** **`D`** **abgeleiteten Klasse** referiert, so ist das **Ergebnis** ein Pointer bzw eine Referenz auf das **`D`-(Teil-)Objekt**.

  ▻ Ist **`T`** ein Pointer bzw eine Referenz auf eine **Klasse** **`A`,** die zugreifbare und eindeutige **Basisklasse** des **durch** **`e`** **tatsächlich referierten Objekts** ist, so ist das **Ergebnis** ein Pointer bzw eine Referenz auf das **`A`-Teil-Objekt** des von **`e`** referierten Objekts.

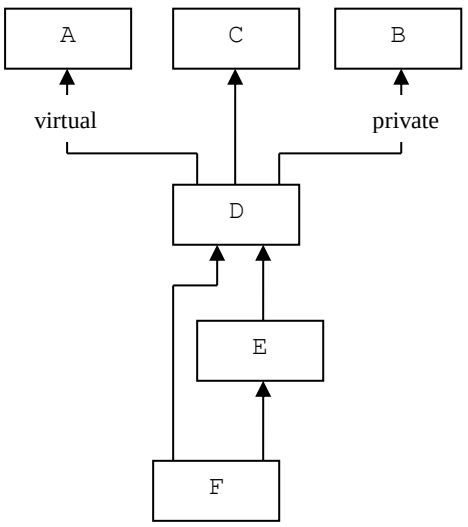
  ▻ In allen **übrigen Fällen** ist die **gewünschte Typumwandlung nicht möglich**.
  → Wenn **`T`** ein **Pointer-Typ** ist, wird als **Ergebnis** der **`NULL`-Pointer** erzeugt;
  wenn **`T`** eine **Referenz** ist, wird die **Exception** **`bad_cast`** geworfen.

Der Typkonvertierungsoperator `dynamic_cast` in C++ (2)

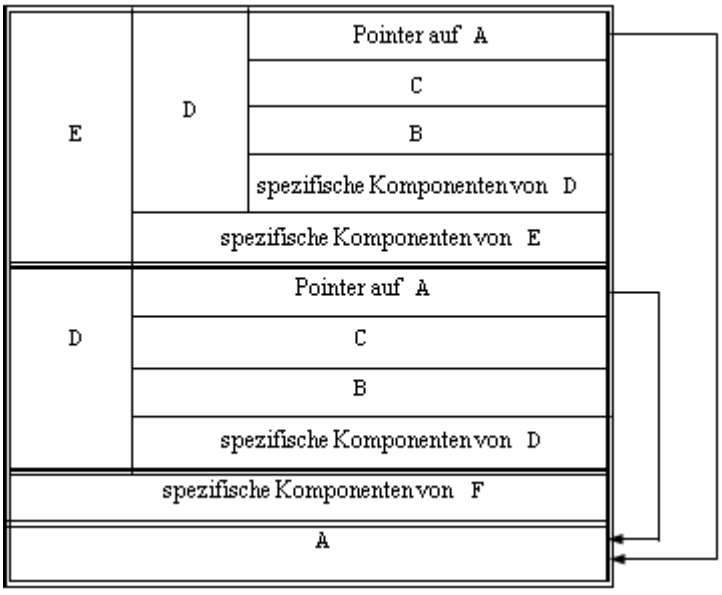
- **Beispiel für Typwandlungen bei polymorphen Klassen** :

```
class A { virtual void f(); /* ...*/ };
class B { virtual void g(); /* ...*/ };
class C { virtual void h(); /* ... */ };
class D : public virtual A, public C , private B { /* ... */ };
class E : public D { /* ... */ };
class F : public E, public D { /* ...*/ };    // in Visual-C++ 6.0 nicht zulaessig

void func(void)
{ A  a;
  D  d;
  E  e;
  F  f;
  A* ap = &a;
  B* bp = dynamic_cast<B*>(&d);    // D* → B*  Fehlschlag (private)
  D* dp = dynamic_cast<D*>(ap);    // A* → D*  Fehlschlag (ap zeigt auf A)
  C* cp = dynamic_cast<C*>(ap);    // A* → C*  Fehlschlag (ap zeigt auf A)
  ap = &d;                         // D* → A*  implizit
  dp = dynamic_cast<D*>(ap);       // A* → D*  hier o.k.(ap zeigt auf D)
  D& dr2 = dynamic_cast<D&>(*ap); // A  → D&  o.k. (ap zeigt auf D)
  cp = dynamic_cast<C*>(ap);       // A* → C*  o.k. (ap zeigt auf D, C ist auch
                                   //           Basisklasse von D)
  bp = dynamic_cast<B*>(ap);       // A* → B*  Fehlschlag (ap zeigt auf D,
  ap = &e;                         //           B ist private Basisklasse von D)
  dp = dynamic_cast<D*>(ap);       // A* → D*  o.k.(ap zeigt auf E,
                                   //           D ist public-Basisklasse von E)
  ap = &f;                         // o.k.,     nur ein A in F enthalten
                                   //           (virtuelle Basisklasse)
  dp = dynamic_cast<D*>(ap);       // A* → D*  Fehlschlag (mehrdeutig)
  E* ep1 = (E*)ap;                 // Fehler,   C-compatibler cast von
                                   //           virtueller Basis
  E* ep2 = dynamic_cast<E*>(ap);   // A* → E*  o.k.(ap zeigt auf F,
                                   //           E ist public-Basisklasse von F)
 }
```

Klassendiagramm



Objekt der Klasse `F`



7.7 Abstrakte Klassen in C++

Eine Klasse, für die wenigstens eine **rein-virtuelle Funktion** deklariert ist, wird als **abstrakte** Klasse bezeichnet.
Eine **rein-virtuelle Funktion** besitzt **keine Definition** in der Basisklasse. Für sie ist **lediglich** das **Interface** (Parameter und Funktionstyp) jedoch keine konkrete Implementierung festgelegt.
Eine **abstrakte Klasse** ist damit **unvollständig** definiert.
Von ihr lassen sich **keine konkreten Objekte anlegen**.
Sie kann **nur als Basisklasse** zur Ableitung anderer Klassen verwendet werden.

Jede von einer abstrakten Klasse abgeleitete Klasse, von der konkrete Objekte erzeugt werden sollen, muß für sämtliche geerbten rein-virtuellen Funktionen Definitionen enthalten (→ **konkrete Klasse**). Eine rein-virtuelle Funktion, die in einer abgeleiteten Klasse nicht definiert wird, bleibt rein-virtuell für diese Klasse. Die abgeleitete Klasse ist damit ebenfalls abstrakt.

Eine **abstrakte Klasse** darf **nicht als Parameter-Typ** und **nicht als Funktions-Rückgabe-Typ** verwendet werden und darf **nicht in** einer **expliziten Typ-Konvertierung** auftreten.
Pointer und Referenzen auf abstrakte Klassen sind zulässig.

Hinweis: Besitzt eine Klasse keine rein virtuelle Funktion, sie soll aber trotzdem nur als Teilobjekt einer abgeleiten Klasse instanziert werden können, so muss ihr Konstruktor in den protected Bereich geschoben werden

- Anwendung :
  Abstrakte Klassen dienen in einer Klassenhierarchie zur Zusammenfassung gemeinsamer Eigenschaften unterschiedlicher konkreter Objekte unter einem Oberbegriff und zur **Bereitstellung eines gemeinsamen Methoden-Interfaces** ohne Implementierungs-Details festzulegen. Die Implementierung der Methoden kann geändert oder ergänzt werden (neue abgeleitete Klassen), ohne daß dies Auswirkungen auf die Schnittstelle und damit auf die Anwendung der Methoden hat.
- Beispiel :

```
class Punkt { private: int x, y; /* ... */ };
class GeoObj  {                              // abstrakte Klasse
    public:
      GeoObj(const Punkt& p) : refpunkt(p) {}
      virtual void move(const Punkt&) = 0;
      virtual void draw() = 0;                  //rein virtuelle Methode
      // ...
    protected:
      Punkt refpunkt;
 };
class Linie : public GeoObj  {    // konkrete Klasse
    public:
      void move(const Punkt& p)   { /* Verschiebe Linie ... */ }
      void draw()                 { /* Zeichne Linie ... */ }
    // ...
 };
class Kreis : public GeoObj  {          // konkrete Klasse
    public:
      void move(const Punkt& p)   { /* Verschiebe Kreis ... */ }
      void draw()                 { /* Zeichne Kreis ... */ }
    // ...
 };
```

Demonstrationsprogramm zu abstrakten Klassen in C++

```
// ------------------------------------------------------------------
// Demonstrationsbeispiel zu abstrakten Klassen
// ------------------------------------------------------------------
  #include <iostream >
   using namespace std;

  class Num  {                    // abstrakte Klasse
  public:
     Num(int i=0) { wert=i;}
     virtual void shownum() const=0;      //rein virtuell, keine Implementierung

  protected:
     int wert;
  };

  class DecNum : public Num  {      // konkrete Klasse
  public:
     DecNum(int i=0) : Num(i) { }
     void shownum(void) const  {
        cout <<"wert dezimal   : " << dec << wert << '\n'; }
 };

  class HexNum : public Num  {      // konkrete Klasse
  public:
     HexNum(int i=0) : Num(i) { }
     void shownum(void) const {
        cout <<"wert sedezimal : " << hex << wert << '\n'; }
   };

  class OctNum : public Num  {      // konkrete Klasse
  public:
     OctNum(int i=0) : Num(i) { }
     void shownum(void) const {
        cout <<"wert oktal     : " << oct << wert << '\n'; }
 };

int main()  {
  Num* baseptr[3];    //Zeiger auf Bsisklassen können schon definiert werden
  DecNum wd(10);                  //Objekt abgeleiter Klasse!!!!
  HexNum wh(511);                //  "
  OctNum wo(63);                   //
  baseptr[0]=&wd;             // implizite Typwandlung DecNum* ⇒ Num*
  baseptr[1]=&wh;             // implizite Typwandlung HexNum* ⇒ Num*
  baseptr[2]=&wo;             // implizite Typwandlung OctNum* ⇒ Num*
  for (int i=0; i<3; i++)
     baseptr[i]->shownum();    //Polymorphie über Basisklassenzeiger
  }
  return 0;
```

Aufruf und Ausgabe des Programms :

wert dezimal : 10
wert sedezimal : 1ff
wert oktal : 77

8 Ausnahmebehandlung

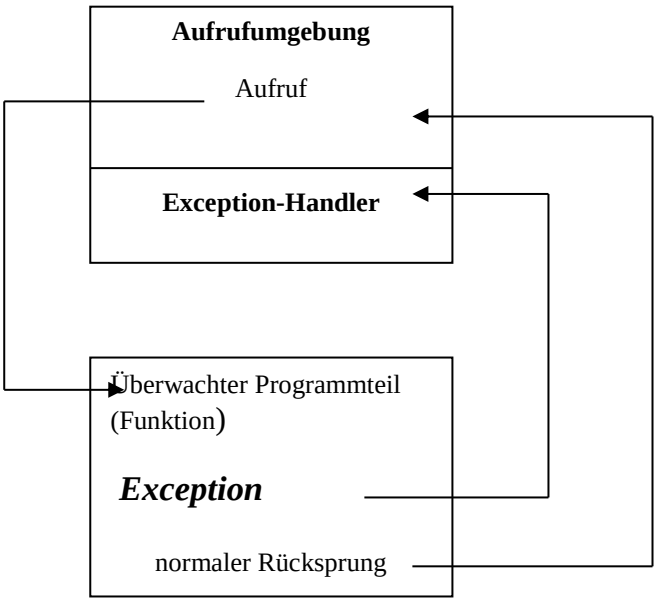
8.1 Ausnahmebehandlung in C++ - Allgemeines

- **Ausnahmesituationen**

  C++ stellt einen speziellen Mechanismus zur Behandlung von Ausnahmesituationen (Fehlerfällen, Exceptions) zur Verfügung. ⇒ Exception Handling.
  Ausnahmesituationen in diesem Sinne sind Fehler oder sonstige unerwünschte Sonderfälle (z.B. Dateizugriffsfehler, Fehler bei der dynamischen Speicherallokation, Bereichsfehler usw.), die im normalen Programmablauf nicht auftreten sollten aber auftreten können.
  Sie werden vom Programmierer festgelegt.
  → Es handelt sich nicht um einen Mechanismus zur Behandlung von externen oder internen Interrupts.

- **Grundprinzip :**

  Das in C++ implementierte Exception-Handling beruht auf der **Trennung** von der im normalen Programmablauf auftretenden **Fehlererkennung** und der **Fehlerbehandlung**:
  Der Programmteil (i. a. eine Funktion), der einen Problemfall (Fehlerfall) entdeckt, bearbeitet diesen nicht selbst, sondern "wirft" eine Exception ("throws an exception").
  In der Aufrufumgebung dieses Programmteils sollte eine Bearbeitungsroutine für diese Exception (→ Exception-Handler) sein. Der Exception Handler "fängt" die Exception und behandelt den Problemfall.



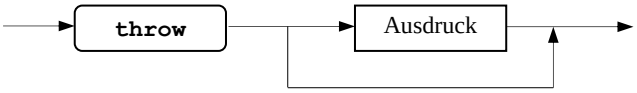
- **Exception-Klassen**

  Eine derartige Exception wird durch einen Ausdruck beschrieben, der prinzipiell einen beliebigen Wert ergeben kann. Im einfachsten Fall kann dies der Wert eines Standard-Datentyps oder ein char-Pointer (C-String) sein, meist wird es sich dabei aber um ein Klassen-Objekt handeln.
  Durch den Typ des erzeugten Werts bzw Objekts lassen sich verschiedene **Exception-Arten** unterscheiden.
  Zweckmäßigerweise definiert man für die verschiedenen Exceptions spezielle Exception-Klassen (Ausnahme-Klassen, Fehler-Klassen, → **Exception-Typ**).
  Beim Auftritt einer Exception wird dann ein Objekt der entsprechenden Klasse erzeugt und geworfen.
  ⇒ die Exception wird als Objekt behandelt.
  Wenn eine Exception-Klasse allein zur Anzeige der Exception-Art dienen soll, muß sie keine Komponenten besitzen. I.a. wird sie jedoch Komponenten haben, die der Fehlerbehandlung genauere Informationen über die Fehlerursache zur Verfügung stellen.

8.2 Werfen und Fangen von Exceptions

- **throw-Ausdruck**

  Das **"Werfen" einer Exception** erfolgt durch einen throw-Ausdruck. Dieser wird mit dem **unären throw-Operator** gebildet, dessen Präzedenz zwischen Komma-Operator und den Zuweisungsoperatoren liegt.
  Der **Typ** eines throw-Ausdrucks ist **void** → ein throw-Ausdruck ist nur in der Form einer Ausdrucksanweisung möglich.



  Der throw-Ausdruck initialisiert ein **temporäres Objekt** seines Operanden-Typs (=**Exception-Typ**) mit dem Wert seines Operanden (→ "geworfene" Exception), **sucht** den passenden **Exception-Handler**, initialisiert gegebenenfalls dessen Parameter mit diesem Wert und übergibt die Programmfortsetzung an diesen.
  → Der Handler hat die Exception "**gefangen**".
  Dabei findet eine **Stackbereinigung** ("*stack unwinding*") statt: Sämtliche auto-Objekte (und einfache auto-Variable), die seit Eintritt in den zum Exception-Handler gehörenden `try`-Block angelegt worden sind, werden entfernt.

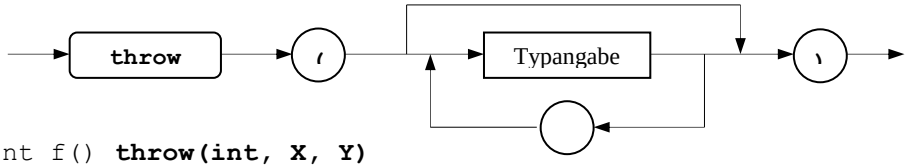
  ◇ Das vom throw-Ausdruck angelegte temporäre Objekt existiert solange, solange ein Exception-Handler für diese Exception ausgeführt wird. Erst nach Beendigung des (letzten) Exception-Handlers für diese Exception wird das Objekt zerstört.

  ◇ Die Form des throw-Ausdrucks **ohne Operanden** ist nur **innerhalb eines Exception-Handlers** (bzw. innerhalb einer von einem Exception-Handler aufgerufenen Funktion) zulässig.
  Ein derartiger throw-Ausdruck bewirkt, daß eine weitere Exception mit dem vorhandenen temporären Objekt geworfen wird, d.h. es wird versucht, die Programmfortsetzung an einen weiteren Exception-Handler zu übergeben (→ "***rethrow***" *the exception*).

  ◇ Kann durch den throw-Ausdruck kein passender Exception-Handler gefunden werden, so wird die (Standardbibliotheks-)Funktion **terminate()** aufgerufen.

- **Exception-Specification**

  ◇ Ergänzung einer **Funktionsdeklaration** bzw des Funktionskopfes einer **Funktionsdefinition** um eine **Auflistung der Exception-Typen**, die von der **Funktion** – direkt oder indirekt – **geworfen** werden können bzw dürfen.



  ◇ **Beispiele** :

```
int f() throw(int, X, Y)
{
..// Funktion darf Exceptions vom Typ int, X und Y werfen
}
```

  ◇ Eine Exception-Specification mit **leerer Typ-Liste** bedeutet, daß die entsprechende Funktion **keine Exceptions** werfen darf. Eine **ohne Exception-Specification** vereinbarte Funktion darf **beliebige Exceptions** werfen.

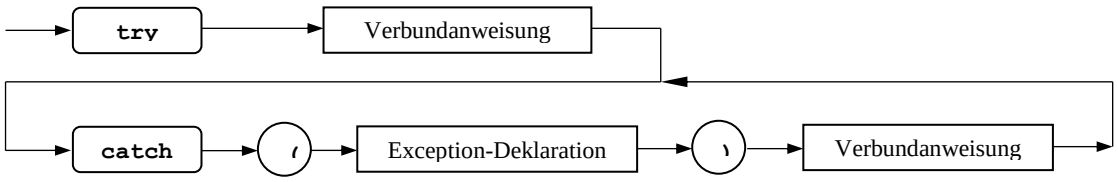
  ◇ Wenn eine Typ-Liste den Typ X enthält, so darf die betreffende Funktion neben Exceptions dieses Typs auch Exceptions aller Typen, die sich von X öffentlich und eindeutig ableiten lassen, werfen

  ◇ Wird während der Abarbeitung einer Funktion eine Exception geworfen, deren Typ nicht in der Typ-Liste enthalten ist, so wird die Standardbibliotheks-Funktion **unexpected()** aufgerufen.

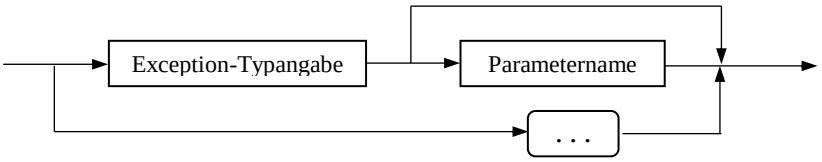
◇ **Anmerkung**: Die Exception-Specification hat bei **Visual-C++ keine Wirkung**.

- **`try`-Anweisung**

  ◇ Sie legt die Aufrufumgebung und damit den Gültigkeitsbereich einer Ausnahmebehandlung und die dazugehörigen Exception-Handler fest.
  ⇒ Sie besteht aus
  - einer **Verbundanweisung** (**`try`-Block**), die aus den Anweisungen besteht, in denen die Fehlererkennung wirksam ist (Häufig sind Aufrufe von Funktionen enthalten, in denen die eigentliche Fehlererkennung stattfindet).
  - den **Exception-Handlern** (**`catch`-Blöcken**), die den verschiedenen jeweils fangbaren Exception-Typen zugeordnet sind. Für jeden Exception-Typ ist ein eigener Handler vorzusehen.



Exception-Deklaration :



  ◇ **`try`-Anweisungen** können **geschachtelt** werden.

  ◇ Der in der Exception-Deklaration eines Handlers angegebene Typ wird als der **Typ des Handlers** bezeichnet.

- **"Fangen" von Exceptions**

  ◇ Der `throw`-Ausdruck bewirkt eine **Suche nach einem Handler**, der die geworfene Exception "fangen", d. h. bearbeiten kann. Die Suche erfolgt in der Reihenfolge der `catch`-Blöcke.
  Dem **ersten Handler**, der die Exception bearbeiten kann, wird das geworfene **Fehlerobjekt übergeben**.
  Die Übergabe erfolgt **analog zur Parameterübergabe** bei Funktionen, allerdings finden **keine impliziten Typkonvertierungen** für **Standard-Datentypen** statt.

  Ist der Typ in der Exception-Deklaration des Handlers von der Form

  ```
  T, const T, T& oder const T&
  ```

  so wird der Handler aufgerufen, wenn der Typ T
  - identisch mit dem Typ der geworfenen Exception ist oder
  - eine Basisklasse der Klasse der geworfen Exception ist oder
  - ein Basisklassenzeiger und die Exception ein Zeiger auf eine Ableitung davon ist

.

- ◇ Ein Handler, in dessen Exception-Deklaration statt eines Typs drei Punkte (`...`) angegeben ist, kann **jede beliebige Exception** fangen. Falls vorhanden, muß er der **letzte Handler** einer try-Anweisung sein.

- ◇ Wird kein passender Handler gefunden, wird die Suche in der nächsten umfassenden `try`-Anweisung – sofern vorhanden – fortgesetzt usw.
  Wird auch auf diese Weise **kein passender Handler** gefunden, wird die (Standardbibliotheks-) Funktion **`terminate()`** aufgerufen.

- ◇ **Nach** erfolgter **Abarbeitung eines Handlers** wird – sofern der Handler nicht das Programm beendet oder erneut eine Exception geworfen hat – das **Programm** mit der Anweisung, die **auf die `try`-Anweisung folgt**, zu der der Handler gehört, **fortgesetzt**. Es findet also **keine Rückkehr** zu der **Stelle**, an der die **Exception geworfen** wurde, statt.

Einfaches Demonstrationsprogramm zur Ausnahmebehandlung in C++ mit selbstdefinierter Exception-Klasse RangeError

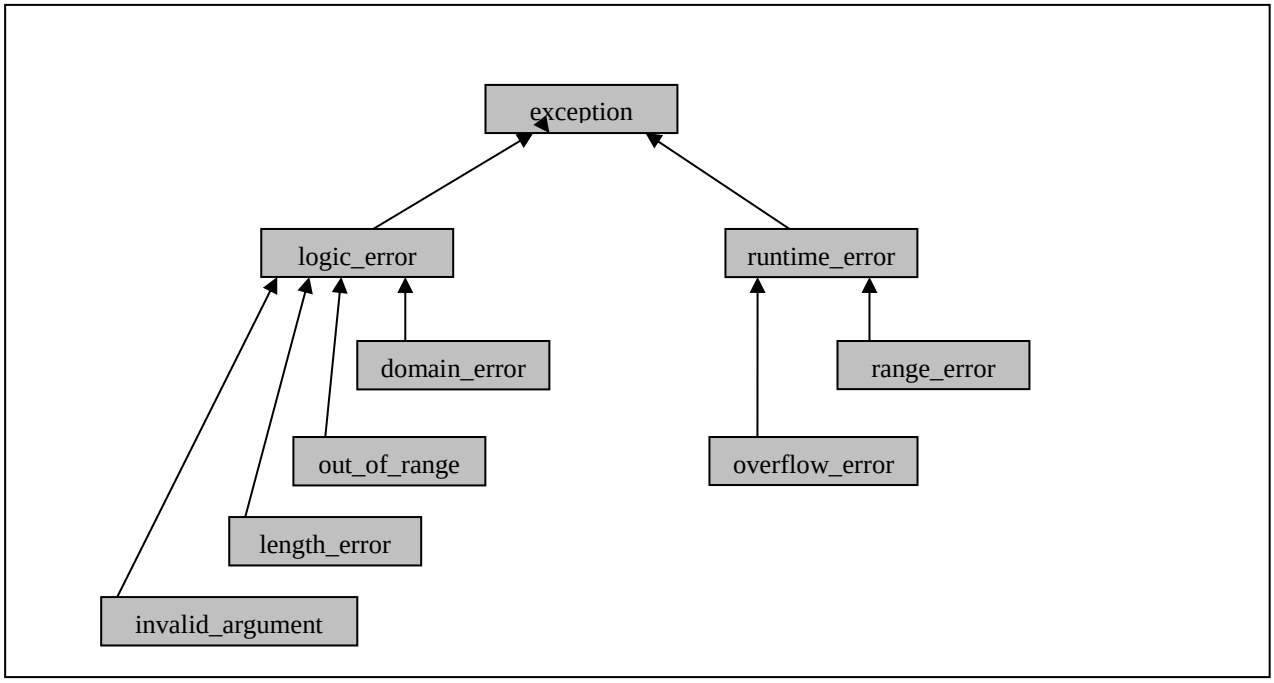
- **Demonstrationsprogramm `exhdemo`:**

```
//-----------------------------------------------------------------------------
// Programm EXHDEMO
// ----------------------------------------------------------------------------
// Einfaches Demonstrationsprogramm zum Exception Handling
// mit eigener Fehlerklasse RangeError
// ----------------------------------------------------------------------------

#include <iostream>
using namespace std;

class IntVec
{ public:
    IntVec(int);
    ~IntVec() {  delete[] ip;  }
    int& operator[](int);
    //  ...
  private:
    int *ip;
    int len;
};

class RangeError                                    // Fehlerklasse
{ public:
    RangeError(int i) : ind(i) { }
    int getInd(){ return ind; }
  private:
    int ind;                                        // fehlerhafter Index
};

IntVec::IntVec(int i)
{  ip=new int[i];  len=i;  for(i=0; i<len; i++)  ip[i]=i;  }

int& IntVec::operator[](int i)
{ if (i<0 || i>=len)
    throw RangeError(i);                            // "Werfen" der Exception
  else
    return ip[i];
};

int main(void)
{ IntVec iv(20);
  try                                               // try-Block
  { cout << "\niv[10] : " << iv[10] << '\n';
    cout <<   "iv[30] : " << iv[30] << '\n';
    cout << "Ende try-Block\n";
  }
  catch (RangeError& err)                           // catch-Block
  { cout << "Index [" << err.getInd() << "] out of range\n";
  }
  cout << "Programmende\n";
```

- **Ausgabe des Programms :**

```
iv[10] : 10
Index [30] out of range
Programmende
```

8.3 Fehlerklassen in der C++ Standard-Bibliothek



Vererbungsdiagramm der Standard-Fehlerklassen

Die obenstehenden Fehlerklassen werden von der C++ Standard-Bibliothek zur Verfügung gestellt. Das Fehlermodell der Standard-Bibliothek unterscheidet zwei Arten von Fehlern:

- Logische Fehler (Klasse logic-error)
- Laufzeitfehler (Klasse runtime_error)

Die Klasse exception stellt eine gemeinsame Basisklasse für alle Fehlerarten zur Vefügung:

```
class exception
{
        public :
        exception() throw();
        exception(const exception&) throw();
        exception& operator=(const exception&) throw();
        virtual ~exception() throw();
        virtual const char* what() const throw();
        private :
// ...
};
```

Die Methode `what()` gibt einen String zurück, der eine Meldung über die Fehlerursache enthält.

Jede andere Fehlerklasse ist entweder von der Klasse `logic_error` oder von der Klasse `run_time_error` abgeleitet

```
class logic_error : public exception {
    public:
       explicit logic_error(const string& message) throw();
       virtual const char* what() throw();
};
```

```
class runtime_error : public exception {
   public:
       explicit runtime_error(const string& message) throw();
       virtual const char* what() throw();
};
```

Von der Klasse `logic_error` ist z.B. die Klasse `out_of_range` abgeleitet.
Man sieht, dass beim Werfen eines Objektes dieser Klasse ein String übergeben werden muss!
Mit `explicit` wird aber verhindert, dass der angegebene Konstruktor zum *Konvertierungs-Konstruktor* wird.

```
class out_of_range : public logic_error {
   public:
       explicit out_of_range(const string& message) throw();
};
```

```
//-------------------------------------------------------------------------
// Programm EXHDEMO mit Verwendung der C++ Standard-Exception-Klassen
// ------------------------------------------------------------------------
#include <iostream>
#include <stdexcept>
using namespace std;

class IntVec
{ public:
    IntVec(int);
    ~IntVec() {  delete[] ip;  }
    int& operator[](int); //throw (out_of_range)
    //  ...
  private:
    int *ip;
    int len;
};

IntVec::IntVec(int i)
{  ip=new int[i];  len=i;  for(i=0; i<len; i++)  ip[i]=i;  }

int& IntVec::operator[](int i)
{     // "Werfen" der Exception;
      if (i<0 || i>=len) throw out_of_range("out_of_range");
         else
           return ip[i];
};

int main()
{
      intVec iv(20);
      try                                          // try-Block
      {  cout << "\niv[10] : " << iv[10] << '\n';
         cout <<   "iv[30] : " << iv[30] << '\n';
         cout << "Ende try-Block\n";
      }
      catch (out_of_range& err) {                  // catch-Block
           cout <"Exception:"<< err.what();
      }
      catch(const exception& err)  {
         cout << "Handler fuer alle Standard-Exceptions\n"
               << err.what() <<endl;
      }
      catch(...) {    cout <<"Unbekannte exception" << endl;  }
      cout << "Programmende  <<endl ;  return 0;
}
```

- **Ausgabe des Programms** :

```
iv[10] : 10
Exception:out_of_range
Programmende
```

Bei Verwendung der C++ Standard-Exception kann im obigen Beispiel der fehlerhafte Index nicht mit ausgegeben werden.
Abhilfe siehe „eigene Fehlerklassen von Standard-Fehlerklassen ableiten“

```
//-------------------------------------------------------------------------
// Programm EXHDEMO mit Verwendung der C++ Standard-Exception-Klassen
// und davon abgeleiteter eigener Fehlerklasse My_out_of_range
// ------------------------------------------------------------------------
#include <iostream>
#include <stdexcept>
using namespace std;

class My_out_of_range:public out_of_range
{
  public:
     //String fuer out_of_range notwendig
     My_out_of_range(const string& message, int ind)
     :out_of_range(message), mind(ind){}
     int getIndex()const {return m_ind;}

  private:
    int m_ind;
};

class IntVec
{ public:
    IntVec(int);
    ~IntVec() {  delete[] ip;  }
    int& operator[](int);
    //  ...
  private:
    int *ip;
    int len;
};


IntVec::IntVec(int i)
{  ip=new int[i];  len=i;  for(i=0; i<len; i++)  ip[i]=i;  }

int& IntVec::operator[](int i)
{ if (i<0 || i>=len)
    throw My_out_of_range("My_out_of_range",i); // "Werfen" der Exception
  else
    return ip[i];
};

int main()
{
      intVec iv(20);
      try                                              // try-Block
      {  cout << "\niv[10] : " << iv[10] << '\n';
         cout <<   "iv[30] : " << iv[30] << '\n';
         cout << "Ende try-Block\n";
      }
      catch (My_out_of_range& err) {                   // catch-Block
         cout <<err.what();
         cout << "Index [" << err.getIndex() << "]" <<endl;
      }
      catch(const exception& err)  {
            cout   << "Handler fuer alle Standard-Exceptions\n"
            << err.what() <<endl;
      }
      catch(...)   {
            cout <<"Unbekannte exception" << endl;
      }

      cout << "Programmende <<endl ;
      return 0;
}
```

- **Ausgabe des Programms** :

```
iv[10] : 10
Exception: out_of_range
Index [30] out of range
Programmende
```

Die `My_out_of_range` Exception erweitert die C++ Standard-Exception um die Möglichkeit der Indexübergabe.

9 C++ I/O-Streams

C++-Stream-I/O - Allgemeines

- Neben der nach wie vor in C++ verfügbaren Verfahren von C zur Datei-u.Geräte-Ein- und Ausgabe (stdio.h) ist in der C++-Standard-Bibliothek eine alternative Möglichkeit zur Ein- u. Ausgabe implementiert.
  Die C++-Standard-I/O-Bibliothek ist konsequent objektorientiert realisiert. Sie besteht aus mehreren - größtenteils voneinander abgeleiteten - Klassen (⇒ I/O-Klassenbibliothek), die sehr mächtige und effektive Möglichkeiten zur Geräte- und Datei-Ein- und Ausgabe zur Verfügung stellen.

- Wie in C findet auch in C++ jegliche Ein- und Ausgabe über Streams statt. Im Streams-Modell wird die Ein- und Ausgabe sämtlicher - auch noch so komplex zusammengesetzter - Daten auf Bytefolgen abgebildet. Streams werden sowohl Dateien als auch Geräten zugeordnet

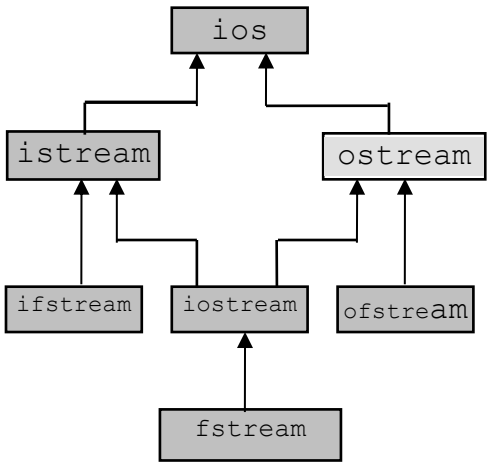
  ⇒ Datei- und Gerätezugriffe sind gleichartig.

  In der C++-Methode sind Streams Objekte, deren Eigenschaften (Zustände und Methoden) durch Klassen beschrieben werden.
  Für die Standard-Ein-/Ausgabe-Geräte sind entsprechende Objekte vordefiniert.

- Die C++-I/O-Klassenbibliothek umfaßt auch String-Stream-Klassen.
  Diese ermöglichen einen zum Datei-/Geräte-Zugriff analogen Zugriff zu Strings.

9.1 Hierarchie der wichtigsten C++-Stream-Klassen

- Anmerkung: die String-Stream-Klassen (strstreambuf, istrstream, ostrstream, strstream, strstreambuf) sind nicht aufgeführt.



Header-Dateien für C++-Stream-I/O und vordefinierte Stream-Objekte

- Header-Dateien für C++-Stream-I/O :

  <iostream>: Grundlegende Headerdatei für C++-Stream-I/O,
  Definition der grundlegenden Stream-Klassen (außer den File-Stream- und den String-Stream-Klassen),
  Deklaration der vordefinierten "Standard"-Stream-Objekte,
  Deklaration der parameterlosen Manipulatoren

  <fstream>: Definition der File-Stream-Klassen,
  Einbindung der Header-Datei <iostream>

  <strstream>: Definition der String-Stream-Klassen,
  Einbindung der Header-Datei <iostream>

  <iomanip> : Definitionen und Deklarationen für die Manipulatoren mit Parametern

  In vielen Implementierungen existieren darüberhinaus häufig noch weitere Header-Dateien, die aber von den obigen Header-Dateien eingebunden werden.

  $\Rightarrow$ Für die Anwendung der C++-I/O-Klassenbibliothek sind i.a. die o.a. Header-Dateien ausreichend.

- Die vordefinierten "Standard"-Stream-Objekte :

Stream-Objekt	Stream-Klasse	Bedeutung	Default-Gerät	Entsprechung C-I/O
cin	istream	Standard-Eingabe-Kanal	Tastatur	stdin
cout	ostream	Standard-Ausgabe-Kanal	Bildschirm	stdout
cerr	ostream	Standard-Fehlerausgabe-Kanal *)	Bildschirm	stderr
clog	ostream	Standard-Protokoll--Kanal *)	Bildschirm	

*) Unterschied : Ausgabe nach cerr ist ungepuffert, Ausgabe nach clog ist gepuffert

Die vordefinierten Stream-Objekte haben die zugehörigen Kanäle bereits geöffnet und können daher unmittelbar verwendet werden. Einzige Voraussetzung für C++-Stream-I/O:

Einbinden von <iostream >

9.2 Der Ausgabe-Operator << in C++

- Zur Ausgabe in Streams ist in der Klasse **ostream** der Links-SchiebeOperator << für alle elementaren Datentypen einschließlich void* und char* (und gegebenenfalls bool) als rechter Operand überladen.
  ⇒ Ausgabeoperator (output operator, insertion operator)

- Der linke Operand muß ein Objekt der Klasse ostream sein (Objekt, für das die Operatorfunktion aufgerufen wird).

  Beispiel :
  ```
  cout << 3.14;
  ```

  Die jeweilige Operatorfunktion gibt den Wert des rechten Operanden in den linken Operanden (Stream !) aus. Das Ausgabeformat hängt vom jeweiligen auszugebenden Wert und vom Zustand des Streams ab.

- Als Ergebnis liefert der Operator eine Referenz auf den linken Operanden, also wiederum ein ostream-Objekt.

  ⇒ Operator läßt sich in verketteten Ausdrücken anwenden.

  Beispiel :
  ```
  int wert ;
  cout << "\nErgebnis: " << wert << '\n';
  ```

- Die Anwendung des Ausgabe-Operators ist nicht auf die in der Definition der Klasse ostream aufgeführten Datentypen beschränkt:
  Durch Überladen des Operators für beliebige eigene Datentypen, können Werte dieser Typen in der gleichen Art und Weise wie elementare Datentypen-Werte ausgegeben werden.

  ⇒ Operator << ist universell einsetzbar.

  Beispielsweise ist der Ausgabeoperator in anderen Teilen der ANSI-C++Standardbibliothek zusätzlich u.a. für Strings, Bitsets und komplexe Zahlen überladen.

C++-Demonstrationsprogramm zum Überladen des Ausgabe-Operators

```
// ------------------------------------------------------------------------
// Programm AUSOPDEM
// ------------------------------------------------------------------------
// Demonstrationsbeispiel zum Überladen des Ausgabe-Operators
// ------------------------------------------------------------------------
  #include <iostream >
   using namespace std
  class Ratio  {
  public:
    Ratio(long=0, long=1);                                // Konstruktor
    Ratio operator+(const Ratio&);                        // Additionsoperator
    friend ostream& operator<<(ostream&, const Ratio&); // Ausgabeoperator
    // ....
  private:
    long zaehler;
    long nenner;
    void kuerze();
  };

  Ratio::Ratio(long z, long n)  {
    if (n<0) { zaehler=-z; nenner=-n; }
    else { zaehler=z; nenner=n; }
    kuerze();
  }

  Ratio Ratio::operator+(Ratio& y)  {
    Ratio temp;
    temp.zaehler=zaehler*y.nenner+nenner*y.zaehler;
    temp.nenner=nenner*y.nenner;
    temp.kuerze();
    return temp;
  }

  long ggt(long a, long b);         // Ermittlung des ggt in Modul ggt.cpp
    void Ratio::kuerze()  {
    long g;
    if ((g=ggt(zaehler>0?zaehler:-zaehler, nenner))>1)  {
      zaehler/=g; nenner/=g; }
  }

  ostream& operator<<(ostream& strm, Ratio rw)  {
    strm << rw.zaehler << '/' << rw.nenner;
    return strm;
  }

  int main()  {
    Ratio a(3,-2);
    Ratio b(13,6);
    cout << "\nRatio-Add : " << a << " + " << b << " = " << a+b << '\n';
   return 0;
  }
```

Aufruf und Ausgabe des Programms :

```
F:\RT\CPP\VORL>ausopdem
Ratio-Add : -3/2 + 13/6 = 2/3
```

9.3 Der Eingabe-Operator >> in C++

- Zur Eingabe aus Streams ist in der Klasse **istream** der Rechts-SchiebeOperator >> für alle elementaren Datentypen einschließlich char* (und ggf. bool) als rechter Operand überladen.
  ⇒ Eingabeoperator (input operator, extraction operator)

- Der linke Operand muß ein Objekt der Klasse istream sein (Objekt, für das die Operatorfunktion aufgerufen wird). Der rechte Operand muß eine Referenz auf eine Variable eines der elementaren Datentypen oder ein char-Pointer (Array-Name !) sein.

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

  Die jeweilige Operatorfunktion liest die nächsten Zeichen aus dem linken Operanden (Stream !) ein und interpretiert diese entsprechend dem Typ des rechten Operanden. Die eingelesenen Zeichen werden in die jeweilige interne Darstellung umgewandelt und der durch den rechten Operanden referierten Variablen zugewiesen. Das Einlesen wird bendet, wenn auf einen Whitespace-Character oder ein Zeichen, das nicht zum zu lesenden Typ paßt, gestoßen wird. (Das Zeichen verbleibt im Stream). Im Normalfall werden führende Whitespace-Character überlesen.
  ⇒ **In eine char-Variable können keine Blanks, Newlines oder Tabs eingelesen werden.** (Allerdings läßt sich das Überlesen von Whitespace-Character im Stream-Objekt abschalten.)

- Als Ergebnis liefert der Operator >> eine Referenz auf den linken Operanden, also wiederum ein istream-Objekt.
  ⇒ Auch mit dem Eingabe-Operator lassen sich verkettete Ausdrücke bilden.
  Beispiel :
  ```
  int wert ;
  char name[30];
  cin >> name  >> wert;
  ```
- Auch der Eingabe-Operator läßt sich für beliebige selbst-definierte Datentypen überladen. Damit können Werte dieser Typen in der gleichen Art und Weise wie Werte der elementaren Datentypen eingelesen werden.
  ⇒ Der Eingabe-Operator ist ebenfalls nicht nur auf die elementaren Datentypen begrenzt, sondern universell einsetzbar.
  Zu beachtende Punkte bei selbstdefiniertem Überladen :
  - der erste Parameter der Operator-Funktion muß eine Referenz auf ein Objekt der Klasse istream sein;
  - der zweite Parameter muß eine Referenz auf ein Objekt des einzulesenden Typs sein;
  - die Operatorfunktion muß auf richtiges Eingabeformat prüfen;
  - beim Auftreten des ersten Zeichens, das nicht mehr dem Eingabeformat entspricht, sollte das Einlesen beendet werden, das falsche Zeichen ist nicht einzulesen, ggf. an den Stream zurückzugeben;
  - bei Eingabefehlern müssen entsprechende Status-Flags im Eingabe-Stream gesetzt werden;
  - der Wert des einzulesenden Objekts darf nur verändert werden, wenn beim Einlesen kein Fehler aufgetreten ist.

Der Eingabe-Operator >> in C++ (2)

- Beispiel für selbstdefinierten überladenen Eingabeoperator :

```
// Einlesen von Ratio-Objekten.
// Gültige Formate : long/long, long/, long

istream& operator>>(istream& strm, Ratio& ro) {
  long z, n;
  strm >> z;
  int c;

  if ((c=strm.get())=='/')  {
    if ((c=strm.peek())!=' ' && c!='\n' && c!='\x09')
      strm >> n;
    else
      n=1;
  }
  else  {
    strm.putback(c);
    n=1;
  }

  if (n==0)
    strm.clear(ios::failbit);
  else
    if (strm) {
      ro=Ratio(z,n);
      ro.kuerze();
    }
  return strm;
}
// Anmerkung :
// Da Zugriff zur privaten Member-Funktion kuerze() erfolgt,
// muß die Operatorfunktion Freund-Funktion der Klasse Ratio sein
```

9.4 Zustand von C++-I/O-Streams

- Streams besitzen einen Zustand. Dieser hängt vom Erfolg/Mißerfolg der letzten I/O-Operation ab. Er bestimmt, ob eine weitere I/O-Operation sinnvoll und möglich ist.
  Zur Kennzeichnung des Zustands dient eine Datenkomponente (Statuswort), in der einzelne Bits (Flags) bestimmte Fehler- und Sondersituationen kennzeichnen (Fehlerflags). Die betreffenden Bits sind durch den Wertevorrat des in der Klasse **ios** (bzw. ios_base) definierten Aufzählungstyps io_state (bzw. iostate) festgelegt (Fehlerzustands-Konstanten).

- Folgende Fehlerzustands-Konstanten sind definiert :

Konstante	Bedeutung
goodbit	kein Fehler aufgetreten, kein Fehlerflag gesetzt
badbit	ein fataler I/O-Fehler ist aufgetreten, der Stream ist prinzipiell nicht mehr in Ordnung, Daten sind verloren gegangen (z.B. Datenträger ist voll)
failbit	ein weniger gravierender I/O-Fehler ist aufgetreten, die letzte I/O-Operation konnte nicht korrekt abgeschlossen werden, der Stream ist aber prinzipiell noch in Ordnung (z.B. Format-fehler bei der Eingabe)
eofbit	das Datei-Ende ist erreicht. Da das Dateiende i.a. erst dann erkannt wird, wenn versucht wird, über das Datei-Ende hinaus zu lesen, wird mit eofbit i.a. auch failbit gesetzt.

  Die genauen Werte der einzelnen Zustands-Konstanten und damit die zugehörigen Bits (Fehlerflags) im Statuswort sind implementierungsabhängig.

- Da die Fehlerzustands-Konstanten nicht global, sondern in der Klasse ios (bzw ios_base) definiert sind, müssen sie i.a. zusammen mit dem Klassennamen und dem Scope Resolution Operator verwendet werden (vollqualifizierter Name), z.B. ios::eofbit.

- Zum Ermitteln/Setzen des Stream-Zustands sind in der Klasse ios (bzw ios_base) geeignete Member-Funktionen definiert:

Methode		Aktion
int good()		liefert true (1), wenn kein Fehlerflag gesetzt ist, andernfalls false (0)
int bad()		liefert true (1), wenn ios::badbit gesetzt ist, andernfalls false (0)
int fail()		liefert true (1), wenn ios::failbit oder ios::badbit gesetzt ist, andernfalls false (0)
int eof()		liefert true (1), wenn ios::eofbit gesetzt ist, andernfalls false (0)
ios::iostate rdstate()		liefert den Stream-Zustand (Statuswort)
ios::iostate clear (ios::iostate status = 0)		- setzt den Stream-Zustand (Statuswort) gleich dem Parameter status, - bzw. setzt den Stream-Zustand auf fehlerfrei (alle Fehler-Flags = 0) bei Aufruf ohne Parameter, - liefert den neuen Stream-Zustand
void setstate (ios::iostate neu_flags)		setzt zusätzliche Fehler-Flags (bereits gesetzte bleiben gesetzt)

Zustand von C++-I/O-Streams (2)

- Überladene Operatorfunktionen
  In der Klasse **ios** (bzw. ios_base) sind die Operatoren ! (logische Negation) und (void *) (Typkonvertierung) so überladen, daß Streams in logischen Ausdrücken bezüglich ihres Zustandes ausgewertet werden können.
  Für das Stream-Objekt strm gilt:

!strm	⇒	- true (!=0), wenn strm in einem Fehler-Zustand ist, d.h. wenigstens ein Fehler-Flag gesetzt ist, - false (==0), wenn strm in keinem Fehler-Zustand ist
(void *)strm	⇒	- NULL-Pointer, wenn strm in einem Fehler-Zustand ist - != NULL-Pointer, wenn strm in keinem Fehler-Zust. ist

Auszug aus der Definition der Klasse ios (WATCOM-C++ 10.5):

```
class ios  {
  public:
    // ....
    operator void * () const;
    int operator !  () const;
    int fail() const;     // Anmerkung : liefert Fehler-Zustand
    // ....
};
inline ios::operator void * () const  {
  return( (void *) (fail()==0) ); }
inline int ios::operator ! () const  {
  return( fail() );
}
```

Da die in Kontrollstrukturen überprüften Bedingungen einen Ausdruck erfordern, der - gegebenenfalls nach impliziter Typwandlung - einen ganzzahligen Wert oder einen Zeigerwert ergibt, lassen sich hiermit z.B. einfache Überprüfungen auf den Erfolg von Stream-Eingaben vornehmen (EOF führt zum Setzen von ios::eofbit und damit zu einem Fehlerzustand.).

Beispiele:

```
        while (cin >> obj)  {     // solange Einlesen erfolgreich ist
                  //...           // obj verabeiten
        }
        if (!(cin >> x))  {       // Klammerung notwendig !
                  //...           // Einlesen war nicht erfolgreich
        }
```

Anmerkung: Ausdrücke dieser Art sind i.a. nicht zum Einlesen von Einzelzeichen (obj vom Typ char) verwendbar, da der Operator >> führende Whitespace-Character (also auch z.B. Blanks) überliest.

9.5 C++-Stream-I/O - Standardfunktionen

- In der C++-Standard-I/O-Bibliothek sind - in Ergänzung zu den I/O-Operatoren - auch Funktionen zur Ein- u. Ausgabe definiert.
  Diese I/O-Standardfunktionen sind Member-Funktionen der Klassen **ostream** bzw. **istream**.
  Sie stellen eine ergänzende bzw. alternative Möglichkeit für Stream-I/O dar. Sie ermöglichen insbesondere auch eine unformatierte rein binäre Ein- und Ausgabe.
  Alle Funktionen zum Lesen bzw. Schreiben, setzen im Fehlerfall entsprechende Fehler-Flags im Statuswort des Streams.

- Memberfunktionen der **Klasse ostream** zur Ausgabe :

`ostream& put(char c)`
- schreibt den Parameter c als nächstes Zeichen in den Ausgabe-Stream
- liefert eine Referenz auf den Ausgabe-Stream als Funktionswert
- entspricht in der Wirkung (nicht im Funktionswert) der C-Standardfunktion putchar() bzw. fputc() (putc())
  ⇒ `cout.put(c)  entspricht  cout << c`

`ostream& write(const char *buff, int anz)`
- schreibt anz Zeichen aus dem über buff referierten Speicherbereich in den Ausgabe-Stream
- liefert eine Referenz auf den Ausgabe-Stream als Funktionswert

```
⇒  cout.write(buff, anz)   entspricht
     for (int i=0; i<anz; i++) cout << buff[i];
```

`ostream& flush(void)`
- gibt den Inhalt des mit dem Ausgabe-Stream verknüpften Buffers tatsächlich aus und leert dadurch den Buffer
- liefert eine Referenz auf den Ausgabe-Stream als Funktionswert

---

`ostream& seekp(…)`
`ios::streampos tellp(void)`
- Diese Funktionen dienen zum Setzen/Ermitteln der Schreibposition, sie haben bei Geräten keine Bedeutung und werden daher im Zusammenhang mit File-I/O besprochen

---

C++-Stream-I/O - Standardfunktionen

- Memberfunktionen der **Klasse istream** zur Eingabe (1).

Anmerkungen :
* Whitespace-Character werden von den Funktionen nicht überlesen
* bei allen Funktionen, die eine Referenz auf den Eingabe-Stream als Funktionswert zurückgeben, ist der Erfolg/Mißerfolg der Leseoperation über den Stream-Zustand ermittelbar

---

`int get(void)`
- liest das nächste Zeichen aus dem Eingabe-Stream und gibt es als Funktionswert zurück
- gibt bei Erreichen des Datei-Endes EOF zurück
- entspricht der C-Standardfunktion getchar() bzw fgetc() (getc())
- Achtung : Da get() auch Whitespace-Character liest, entspricht c=cin.get() nicht cin>>c

---

`istream& get(char& c)`
- liest das nächste Zeichen aus dem Eingabe-Stream und weist es dem Parameter c zu.
- liefert eine Referenz auf den Eingabe-Stream als Funktionswert

  ⇒ typische Anwendung :

```
void stream_copy(ostream& out, istream& in) {
  char c;
  while (in.get(c)) out.put(c);
}
```

`istream& get(char *buff, int anz, char ende='\n')`
- liest maximal anz-1 Zeichen aus dem Eingabe-Stream und legt sie in dem durch buff referierten Speicherbereich ab
- beim Auftritt des Zeichens ende wird das Einlesen vorher beendet
- das Zeichen ende wird nicht gelesen, es verbleibt im Eingabe-Stream - die gelesenen Zeichen werden mit dem '\0'-Character abgeschlossen - das Zeichen '\n' ist default für Ende
- liefert eine Referenz auf den Eingabe-Stream als Funktionswert

---

`istream& getline(char *buff, int anz, char ende='\n')`
- liest maximal anz-1 Zeichen aus dem Eingabe-Stream und legt sie in dem durch buff referierten Speicherbereich ab
- beim Auftritt des Zeichens ende wird das Einlesen beendet
- das Zeichen ende wird aus dem Eingabe-Stream entfernt (Unterschied zu get(¨¨¨¨¨)), es wird jedoch nicht im Speicher abgelegt
- die gelesenen Zeichen werden mit dem '\0'-Character abgeschlossen
- das Zeichen '\n' ist default für Ende
- liefert eine Referenz auf den Eingabe-Stream als Funktionswert

---

C++-Stream-I/O - Standardfunktionen (3)

- Memberfunktionen der **Klasse istream** zur Eingabe (2) :

`istream& read(char *buff, int anz)`

- liest die nächsten anz Zeichen aus dem Eingabe-Stream und legt sie in dem durch buff referierten Speicherbereich ab
- die gelesenen Zeichen werden nicht mit dem '\0'-Character abgeschlossen
- liefert eine Referenz auf den Eingabe-Stream als Funktionswert
- das vorzeitige Erreichen des Datei-Endes wird als Fehler betrachtet und führt zum Setzen von ios::failbit neben ios::eofbit

---

`int readsome(char *buff, int anz)` NEU im ANSI/ISO-Standard

- liest die nächsten anz Zeichen aus dem Eingabe-Stream und legt sie in dem durch buff referierten Speicherbereich ab
- die gelesenen Zeichen werden nicht mit dem '\0'-Character abgeschlossen
- liefert die Anzahl gelesener Zeichen als Funktionswert
- das vorzeitige Erreichen des Datei-Endes wird nicht als Fehler betrachtet (ios::eofbit wird nicht gesetzt)

`int gcount(void) const`

- liefert die Anzahl der bei der letzten Leseoperation eingelesenen Zeichen als Funktionswert
- sinnvoll einsetzbar nach Beendigung von read(...) oder get(...) bei vorzeitigem Datei-Ende

---

`istream& ignore(int anz=1, int ende=EOF)`

- überliest maximal anz Zeichen im Eingabe-Stream
- beim Auftritt des Zeichens ende (bzw defaultmäßig bei Datei-Ende) wird das Überlesen beendet
- das Zeichen Ende wird ebenfalls überlesen
- liefert eine Referenz auf den Eingabe-Stream als Funktionswert
- Beispiel : Überlesen des Rests der aktuellen Eingabezeile `cin.ignore(INT_MAX, '\n')`

`int peek(void)`

- liefert das nächste Zeichen aus dem Eingabe-Stream als Funktionswert
- das Zeichen wird nicht ausgelesen, sondern verbleibt im Eingabe-Stream $\Rightarrow$ es kann mit der nächsten Einlese-Operation gelesen werden - liefert bei Erreichen des Datei-Endes EOF als Funktionswert

---

C++-Stream-I/O - Standardfunktionen (4)

- Memberfunktionen der **Klasse istream** zur Eingabe (3) :

`istream& putback(char c)`

- gibt das zuletzt gelesene und als Parameter übergebene Zeichen c in den Eingabe-Stream (genauer : in dessen Buffer) zurück
- kann das Zeichen nicht zurückgegeben werden (kein Platz im Buffer) oder wird versucht das falsche (nicht zuletzt gelesene) Zeichen zurückzugeben, wird ios:badbit gesetzt
- liefert eine Referenz auf den Eingabe-Stream als Funktionswert
- in Implementierungen nach dem AT&T-de-facto-Standard ist die Zurückgabe eines anderen als des gelesenen Zeichens zulässig, es erfolgt keine entsprechende Überprüfung und kein daraus resultierendes Setzen von ios::badbit

---

`istream& unget(char c)` NEU im ANSI/ISO-Standard

- gibt das als Parameter übergebene Zeichen c in den Eingabe-Stream (genauer : in dessen Buffer) zurück
- kann das Zeichen nicht zurückgegeben werden (kein Platz im Buffer) wird ios:badbit gesetzt
- es erfolgt keine Prüfung, ob das zurückzugebende Zeichen das zuletzt gelesene Zeichen ist (Unterschied zu putback())
- liefert eine Referenz auf den Eingabe-Stream als Funktionswert

---

`istream& seekg(…)`
`ios::streampos tellg(void)`

- Diese Funktionen dienen zum Setzen/Ermitteln der Leseposition, sie haben bei Geräten keine Bedeutung und werden daher im Zusammenhang mit File-I/O besprochen

---

Demonstrationsbeispiel zu C++-Stream-I/O-Standardfunktionen

```
// ------------------------------------------------------------------------
// Programm IOFUDEM - Demonstrationsprogramm zu I/O-Standardfunktionen
// Überladener Ein- und Ausgabe-Operator für selbstdef. String-Klasse
// ------------------------------------------------------------------------
 #include <iostream >
 #include <cstdlib.h>
 #include <limits.h>
 class String  {
  public:
   String(int anz=80);                    // Konstruktor
   ~String(void) { delete [] cp; }        // Destruktor
   // ¨¨¨
   friend istream& operator>>(istream&, String&);
   friend ostream& operator<<(ostream&, String&);
  private:
   char *cp;
   int max;
   int len;
 };
// ---------------------------------------------------------------------
 String::String(int lang)  {
  max=lang;
  if ((cp=new char[max+1])==NULL)  {
      cout << "\nAllokations-Fehler\n"; exit(1); }
  len=0; cp[0]='\0';
 }

 istream& operator>>(istream& istrm, String& so){     // Einlesen max. so.max
  so.len=istrm.get(so.cp, so.max+1).gcount();         // Zeichen oder bis '\n' return
  istrm.ignore(INT_MAX, '\n');                  // Überlesen Zeilen-Rest
 }

 ostream& operator<<(ostream& ostrm, String& so) {
  ostrm.write(so.cp, so.len);                   // oder:  ostrm << so.cp;
  return ostrm << '[' << so.len << ']';
// ---------------------------------------------------------------------

 int main(void)  {            // Eingabe von Zahl und String
  String sa(20);              // Eingabe Zahl darf fehlen, dann Default=1
  int izahl;
  cout << "Geben Sie [ Nummer und ] Name ein :\n";
  while (!(cin >> izahl).eof())  {
   if(!cin.good()) izahl = 1;
   cin.clear(); cin >> sa;
   cout << izahl << "   " << sa << '\n';
   }
  return 0;
 }
```

```
 Beispiel für Aufruf und Ausgabe des Programms :
 G:\>iofudem
 Geben Sie [ Nummer und ] Name ein :
 23   Wasserpumpenzangenbehaelter
 23   Wasserpumpenzangenbe[20]
 Taschenrechner
 1   Taschenrechner[14]
^Z
```

9.6 C++-Stream-I/O : Formatierung der Aus- u. Eingabe

- Die C++-Stream-I/O erlaubt eine Formatierung der Ausgabe in vielfältiger Weise. Auch das Eingabeformat kann hinsichtlich einiger Aspekte beeinflußt werden.
  Die jeweils aktuellen Eigenschaften des Ausgabe- bzw Eingabe-Formats werden in speziellen - in der Klasse **ios** definierten - Datenkomponenten des Stream-Objekts festgehalten :
  - Formatflags
  - Feldbreite
  - Genauigkeit bei Gleitpunktzahlen (Anzahl Nachpunktstellen)
  - Füllzeichen

- Die meisten Format-Eigenschaften werden durch einzelne Bits gekennzeichnet, die in einer Datenkomponente zusammengefaßt sind
  $\Rightarrow$ Formatflags
  Ähnlich wie die Fehler-Flags des Stream-Zustands sind die Formatflags durch den Wertevorrat eines in der Klasse ios (bzw ios_base) definierten Aufzählungstyps (fmt_flags bzw fmtflags) festgelegt.
  Folgende Formatflags sind definiert:

Formatflag	Bedeutung	Bitfeld
left	linksbündige Ausgabe	
right	rechtssbündige Ausgabe	adjustfield
internal	Vorz. links-, Wert rechtsbündig	
dec	Aus- bzw Eingabe dezimal von	
oct	Aus- bzw Eingabe oktal ganzen	basefield
hex	Aus- bzw Eingabe hexadezimal Zahlen	
showbase	Ausgabe mit Zahlensystem-Kennung (0 für oktal, 0x für hexadezimal)	
showpoint	immer Ausgabe des Dezimalpunktes und abschließende Nullen (bei Gleitpunktzahlen)	
showpos	explizite Ausgabe eines pos. Vorzeichens	
uppercase	Ausgabe von Großbuchstaben bei hexadezimalen Zahlen und Gleitpunktzahlen in Exponentialdarstellung	floatfield
fixed scientific	Ausgabe von Gleitpunktzahlen in Dezimalbruchdarstellung Ausgabe von Gleitpunktzahlen in Exponentialdarstellung	floatfield
skipws	Überlesen von führendenWhitespace-Char.	
unitbuf	Leeren des Puffers nach jeder Ausgabe	
boolalpha *)	Ausgabe von Bool-Werten als Text, ansonsten als 0/1	
stdio **)	Flush cout, cerr nach jeder Ausgabe	

*) im ANSI/ISO-Standard neu eingeführt (z.Zt. kaum implementiert)
**) nur im AT&T-de-facto-Standard vorhanden (nicht im ANSI/ISO-Std)

C++-Stream-I/O : Formatierung der Aus- u. Eingabe (2)

- Da die Formatflags in der Klasse ios (bzw ios_base) definiert sind, muß bei ihrer Verwendung der vollqualifizierte Name (klassenname::flagname) angegeben werden, z.B. ios::hex
- Die genauen Werte der einzelnen Formatflags und damit die ihnen zugeordneten Bits sind implementierungsabhängig.
- Zum Setzen bzw Ermitteln der Formatflags können spezielle in der Klasse ios (bzw ios_base) definierte Memberfunktionen verwendet werden.
  Das Setzen einzelner Formatflags kann auch mit sogenannten Manipulatoren erfolgen, die wie Ausgabewerte bzw Eingabeobjekte als rechte Operanden in Aus- bzw. Eingabe-Ausdrücke verwendet werden.
- Memberfunktionen der Klasse ios (bzw ios_base) zum Ermitteln bzw. Setzen der Formatflags :
  Die Funktionen verwenden den implementierungsabhängigen - in der Klasse ios (bzw ios_base) definierten - Datentyp fmtflags (meist gleich long).

```
ios::fmtflags flags(void) const
```
- liefert die Formatflags (d.h. den Wert der entsprechenden Datenkomponente) als Funktionswert

---

```
ios::fmtflags flags(ios::fmtflags bits)
```
- setzt alle Formatflags auf die im Parameter bits enthaltenen Werte (d.h setzt die entsprechende Datenkomponente gleich bits)
- die alten Formatflags werden überschrieben
- gibt die alten Formatflags als Funktionswert zurück

```
ios::fmtflags setf(ios::fmtflags onbits)
```
- setzt die im Parameter onbits gesetzten Formatflags
- die übrigen Formatflags werden nicht beeinflußt
- gibt die alten Formatflags als Funktionswert zurück

```
ios::fmtflags setf(ios::fmtflags onbits, ios::fmtflags mask)
```
- setzt die durch den 2. Parameter mask festgelegten Formatflags (üblicherweise ein Bitfeld) zurück und setzt anschließend die im 1. Parameter onbits gesetzten Formatflags.
  Typ. Anwendung : Setzen von Formatflags innerhalb eines Bitfelds unter gleichzeitigem Rücksetzen der übrigen Flags dieses Bitfelds (innerhalb eines der definierten Bitfelder darf immer nur 1 Flag gesetzt sein)
- die übrigen Formatflags (außerhalb des durch mask festgelegten Bitfelds) werden nicht beeinflußt.
- gibt die alten Formatflags als Funktionswert zurück

```
ios::fmtflags unsetf(ios::fmtflags offbits)
```
- setzt die im Parameter offbits gesetzten Formatflags zurück
- die übrigen Formatflags werden nicht beeinflußt
- gibt die alten Formatflags als Funktionswert zurück

9.7 C++-Stream-I/O : Demonstrationsprogramm zu Formatflags

```
// ------------------------------------------------------------------
// Programm FLAGDEMO
// Demonstrationsprogramm zur Wirkung der Formatflags
// ------------------------------------------------------------------
 #include <iostream>
 using namespace std;

  int main()  {
   int ivar=1023, iwert=127;
   double dvar=253.0;
   cout.setf(ios::hex, ios::basefield);
   cout << "\nFormatflags   : cout : " << cout.flags();
   cout << "  +++  cin : " << cin.flags();
   cout << "\nWert von ivar : " << ivar;
   cout.setf(ios::showbase | ios::uppercase);
   cout << "\nWert von ivar : " << ivar;
   cout.unsetf(ios::uppercase);
   cout.setf(ios::showpos);
   cout << "\nWert von ivar : " << ivar;
   cout.setf(ios::dec, ios::basefield);
   cout << "\nWert von ivar : " << ivar;
   cout << "\nWert von dvar : " << dvar;
   cout.setf(ios::showpoint);
   cout << "\nWert von dvar : " << dvar;
   cout.unsetf(ios::showpos);
   cout.setf(ios::scientific, ios::floatfield);
   cout << "\nWert von dvar : " << dvar;
   cout.setf(ios::fixed);
   cout.unsetf(ios::showpoint);
   cout << "\nWert von dvar : " << dvar;
   cin.setf(ios::hex, ios::basefield);
   cout << "\nEingabe 2 int-Werte (sedezimal) ? ";
   cin >> ivar >> iwert;
   cout << "eingeg. Werte : " << ivar << "   und   " << iwert;
   cout.setf(ios::hex, ios::basefield);
   cout << "\nFormatflags   : cout : " << cout.flags();
   cout << "  +++  cin : " << cin.flags() << '\n';
   return 0;
 }
```

// ----------

Ausgaben:

```
Formatflags   : cout : 40  +++  cin : 1
Wert von ivar : 3ff
Wert von ivar : 0X3FF
Wert von ivar : 0x3ff
Wert von ivar : +1023
Wert von dvar : +253
Wert von dvar : +253.000
Wert von dvar : 2.530000e+002
Wert von dvar : 253
Eingabe 2 int-Werte (sedezimal) ? 23  BC
eingeg. Werte : 35   und   188
Formatflags   : cout : 0x18c0  +++  cin : 0x41
Press any key to continue
```

C++-Stream-I/O : Formatierung der Aus- u. Eingabe (2)

- Weitere beeinflußbare Format-Eigenschaften sind in jeweils eigenen Datenkomponenten des Stream-Objekts abgelegt:
  Feldbreite (width),
  Genauigkeit bei Gleitpunktzahlen (precision),
  Füllzeichen (fill character).
  Zum Ermitteln/Setzen dieser Eigenschaften sind in der Klasse **ios** (bzw ios_base) geeignete Member-funktionen definiert. Ein Setzen ist auch mit Manipulatoren möglich (s.u.).

- Feldbreite
  Defaultmäßig entspricht die Ausgabe-Feldbreite genau der für die Darstellung des Ausgabewertes benötigten Zeichenzahl
  ($\Rightarrow$ Default-Feldbreite = 0).
  Sie kann jedoch - immer nur für die jeweils nächste Ausgabe - auf einen anderen Wert gesetzt werden. Benötigt die Ausgabe des Wertes tatsächlich mehr Zeichen, so wird die Feldbreite entsprechend erhöht. Die Feldbreite legt also eine minimale Ausgabe-Feldgröße fest.
  Nach jeder Ausgabe wird die Feldbreite wieder auf ihren Defaultwert 0 gesetzt.

`int width(int breite)`
- setzt die Feldbreite für die nächste Ausgabe auf den im Parameter breite spezifizierten Wert
- liefert den alten Wert der Feldbreite als Funktionswert

`int width(void) const`
- liefert den aktuellen Wert der Feldbreite als Funktionswert

---

Die eingestellte Feldbreite beeinflußt auch die Eingabe von Strings mittels des Eingabeoperators. Ist ihr Wert breite ungleich 0, so werden maximal (breite-1) Zeichen eingelesen und mit einem '\0'-Character abgeschloseen. Wenn vorher ein Whitespace-Character auftritt, werden entsprechend weniger Zeichen gelesen. Die Feldbreite legt also eine maximale Eingabefeldgröße für Strings fest.
Defaultmäßig (breite=0) werden immer alle Zeichen bis zum nächsten Whitespace-Character eingelesen.
$\Rightarrow$ Beim Einlesen von Strings mittels des Eingabe-Operators sollte sicherheitshalber immer die Feldbreite auf die Größe des den String aufnehmenden Speicherbereichs begrenzt werden.

Beispiel :

```
char name[30];
cin.width(sizeof(name));
cin >> name;
```

Da nach dem Einlesen eines Strings die Feldgröße wieder auf ihren Defaultwert 0 gesetzt wird, muß sie für jeden weiteren einzulesenden String erneut gesetzt werden.

C++-Stream-I/O : Formatierung der Aus- u. Eingabe (4)

- Genauigkeit bei Gleitpunktzahlen
  Die Genauigkeit bei Gleitpunktzahlen legt für die Ausgabe von Gleitpunktzahlen sowohl in der Dezimalbruchdarstellung als auch in der Exponentialdarstellung die maximale Anzahl der Nachpunktstellen fest - sofern die Darstellungsart über ein Formatflag explizit eingestellt wurde. Die letzte Stelle wird gegebenenfalls gerundet. Abschließende Nullen nach dem Dezimalpunkt werden nicht mit ausgegeben.
  Ist keines der Formatflags für die Darstellungsart gesetzt, so bestimmt gemäß dem ANSI/ISO-Standard die Genauigkeit die Gesamtanzahl der Stellen. Wenn mit dieser Stellenzahl die Zahl als Dezimalbruch dargestellt werden kann, wird diese Darstellungsart gewählt, andernfalls die Exponentialdarstellung. Allerdings zeigen reale Implementierungen für diesen Fall ein zum Teil abweichendes Verhalten.
  Der Defaultwert für die Genauigkeit beträgt 6.
  Er kann durch einen beliebigen anderen Wert ersetzt werden.
  Eine einmal eingestellte Genauigkeit gilt solange für alle folgenden Gleitpunktzahl-Ausgaben, bis sie explizit geändert wird.

`int precision(int npstell)`

- setzt die Genauigkeit (Anzahl der Nachpunktstellen) auf den im Parameter npstell spezifizierten Wert
- liefert den alten Wert der Genauigkeit als Funktionswert

`int precision(void) const`

- liefert den aktuellen Wert der Genauigkeit als Funktionswert

---

Beispiel :

```
// Programm PRECBSP
#include <iostream>
using namespace std;
                                          // Ausgabe :
int main()  {
   double dvar=123.45678;
   cout << '\n' << dvar << '\n';          // 123.45678
   cout.setf(ios::fixed, ios::floatfield);
   cout << dvar << '\n';                  // 123.45678
   cout.setf(ios::scientific, ios::floatfield);
   cout << dvar << '\n';                  // 1.234567e+02
   cout.precision(3);
   cout << '\n' << dvar << '\n';          // 1.234e+02
   cout.setf(ios::fixed, ios::floatfield);
   cout << dvar << '\n';                  // 123.457
   return 0;
}
```

C++-Stream-I/O : Formatierung der Aus- u. Eingabe (5)

- Füllzeichen
  Ist die Ausgabe-Feldbreite größer als die Anzahl der auszugebenden Zeichen, so wird in die nicht belegten Stellen das Füllzeichen ausgegeben.
  Defaultmäßig ist das BLANK als Füllzeichen eingestellt. Es kann durch ein beliebiges anderes Zeichen ersetzt werden. Ein einmal eingestelltes Füllzeichen gilt solange, bis es explizit durch ein anderes ersetzt wird.

`char fill (char fzeich)`

- setzt das Füllzeichen gleich dem als Parameter fzeich übergebenen Wert - liefert das alte Füllzeichen als Funktionswert

---

`char fill(void) const`

- liefert das aktuelle Füllzeichen als Funktionswert

Beispiel :

```
// Programm FILLBSP
#include <iostream>
using namespace std;
int main()  {
   int iwert=-234;
   cout << '\n';
   cout.width(10);
   cout << iwert << '\n';
   cout.fill('#');
   cout.width(10);
   cout << "Hallo" << '\n';
   cout.width(10);
   cout << iwert << '\n';
   cout.setf(ios::left, ios::adjustfield);
   cout.width(10);
   cout << iwert << '\n';
   cout.setf(ios::internal, ios::adjustfield);
   cout.width(10);
   cout << iwert << '\n';
   return (0);
}
```

```
G:\>fillbsp        (Aufruf und Ausgabe des Programms)
     -234
#####Hallo
######-234
-234######
-######234
```

9.8 C++-Stream-I/O : Manipulatoren

- Manipulatoren sind spezielle Operanden in I/O-Ausdrücken, die i.a. nicht eingelesen bzw. ausgegeben werden, sondern in ganz bestimmter Weise das jeweilige Stream-Objekt beeinflussen ("manipulieren"), z.B. Formateinstellungen vornehmen oder den Ausgabepuffer leeren.
  Da sie wie normale I/O-Operanden verwendet werden, können sie auch in zusammengesetzten (verketteten) I/O-Ausdrücken auftreten. Dies erlaubt i.a. eine wesentlich effektivere Realisierung der I/O-Formatierung als die explizite Verwendung der entsprechenden Memberfunktionen:
  Die Formateinstellungen werden innerhalb einer "Shift-Kette" vorgenommen, diese braucht also nicht unterbrochen zu werden.

- Es gibt vordefinierte Standard-Manipulatoren sowohl als parameterlose Manipulatoren als auch als Manipulatoren mit Parametern.
  Die parameterlosen Standard-Manipulatoren sind in der Header-Datei <iostream.h>, die Standard-Manipulatoren mit Parametern in der HeaderDatei <iomanip.h> deklariert.

- Die parameterlosen Manipulatoren sind durch Funktionen folgenden Typs realisiert :

```
ios&     iomani(ios&);
ostream&  omani(ostream&);
istream&  imani(istream&);
```

Beispiel : Prinzipielle Realisierung des Manipulators `endl`

```
ostream& endl(ostream& strm)  {
  strm.put('\n');     //put() ist Memberfunktion der Klasse ostream
  strm.flush();       //flush() ist ebenfalls Memberfunktion
  return strm;
}
```

Um Manipulatoren in eine "Shift-Kette" einbeziehen zu können sind die Operatoren << bzw. >> in den Klassen `ostream` bzw. `istream` auch für Pointer auf Funktionen dieses Typs überladen.
Auszug aus den Klassendefinitionen (Prinzipbeispiel):

```
class ostream {
   //.....
   ostream &operator << (ostream &(*__f)(ostream &) );
   ostream &operator << (ios     &(*__f)(ios     &) );
   //.....
}
class istream {
   //.....
   istream &operator << (istream &(*__f)(istream &) );
   istream &operator << (ios     &(*__f)(ios     &) );
   //.....
}
```

Prinzipiell sind diese Operatorfunktionen wie folgt definiert:

```
ostream & ostream ::operator << (ostream & (*__f)(ostream &)) {
   return (*__f)(*this);}
```

C++-Stream-I/O : Manipulatoren (2)

- Es ist jederzeit möglich, eigene parameterlose Manipulatoren zu erzeugen. Hierfür muß lediglich eine Funktion obiger Art - wie endl() - definiert werden.

- Parameterlose Standard-Manipulatoren (deklariert in <iostream >)

Name	Anwendung für	Wirkung	
endl	Ausgabe	Newline ausgeben und Puffer leeren	
ends	Ausgabe	NUL-Char. ausgeben und Puffer leeren	
flush	Ausgabe	Puffer leeren	
ws	Eingabe	Whitespace-Character überlesen	
dec	Ein-/ Ausgabe	dezimale Darstellung ganzer Zahlen	
hex	Ein-/ Ausgabe	hexadezimale Darstellung ganzer Zahlen	
oct	Ein-/ Ausgabe	oktale Darstellung ganzer Zahlen	
left	Ausgabe	linksbündige Ausgabe einstellen	*)
right	Ausgabe	rechtsbündige Ausgabe einstellen	*)
internal	Ausgabe	VZ linksbündig, Wert rechtsbündig	*)
fixed	Ausgabe	Dezimalbruchdarstellung einstellen	*)
scientific	Ausgabe	Exponentialdarstellung einstellen	*)
boolalpha	Ausgabe	ios::boolalpha setzen	*)
noboolalpha	Ausgabe	ios::boolalpha rücksetzen	*)
showpos	Ausgabe	ios::showpos setzen	*)
noshowpos	Ausgabe	ios::showpos rücksetzen	*)
uppercase	Ausgabe	ios::uppercase setzen	*)
nouppercase	Ausgabe	ios::uppercase rücksetzen	*)
showbase	Ausgabe	ios::showbase setzen	*)
noshowbase	Ausgabe	ios::showbase rücksetzen	*)
showpoint	Ausgabe	ios::showpoint setzen	*)
noshowpoint	Ausgabe	ios::showpoint rücksetzen	*)
skipws	Eingabe	ios::skipws setzen	*)
noskipws	Eingabe	ios::skipws rücksetzen	*)

*) im ANSI/ISO-Standard neu eingeführt (nicht überall implementiert)

Name	Anwendung für	Wirkung
setw(int w)	Ein-/Ausgabe	Setzen der Feldbreite auf w Stellen (entspricht width())
setprecision(int p)	Ausgabe	Setzen der Genauigkeit auf p Stellen (entspricht precision())
setfill(int c)	Ausgabe	Setzen des Füllzeichens auf das Zeichen c (entspricht fill())
setbase(int b)	Ausgabe	Setzen der Basis des Zahlen-systems auf b (nur für b = 8, 10, 16)
setiosflags(fmtflags f)	Ein-/Ausgabe	Setzen der in f spezifizierten Formatflags (entspricht setf())
resetiosflags(fmtflags f)	Ein-/Ausgabe	Rücksetzen der in f spezifizierten Formatflags (entspricht unsetf())

C++-Stream-I/O : Manipulatoren (3)

- Beispiel :

```
#include <conio.h>
#include <iostream.h>
#include <iomanip.h>
 using namespace std;
void main()  {
    cout << setiosflags(ios::left)       //linksbündig

         << setfill('*')                 //Füllzeichen *
         << setprecision(3);             //Genauigkeit
    cout  << setw(21) << 4.56789 << endl; //Länge des Ausgabefeldes
    cout  << setw(21) << -3123.4567  << endl;
    cout  << setprecision(5);
    cout  << setw(21) << 4.56789       << endl;
    cout  << setw(21) << -3123.4567  << endl;


    cout << resetiosflags(ios::left)     //IO-Statusflag zurücksetzen
         << setiosflags(ios::right)      //rechtsbündig
         << setfill('!');                // neues Füllzeichen !
    cout << setw(21) << 4.56789 << '\n'
         << setw(8)  << -3.1234567 << endl;

    cout << hex << setiosflags(ios::showbase) << 100 << endl;
    //Zahlenbasis
                                         // cout.setf(ios::hex, ios::basefield);
    cout << setfill('#') << setw(10) <<  "Hallo" << endl;
    cout << dec << resetiosflags(ios::right | ios::internal)
          << setiosflags(ios::left) << setw(8) << 255 << endl;
}
```

Ausgabe:

```
4.57*****************
-3.12e+003***********
4.5679***************
-3123.5**************
!!!!!!!!!!!!!!!4.5679
!-3.1235
0x64
#####Hallo
255#####
```

9.9 C++-Stream-I/O : Dateibearbeitung - Allgemeines

- Dateien werden ebenfalls über Stream-Objekte angesprochen. Hierfür sind in der Header-Datei <**fstream**> die File-Stream-Klassen definiert :

  - **ofstream** (ANSI/ISO-Standard : Klassen-Template basic_ofstream<>)
    für Dateien, die geschrieben werden sollen,
  - **ifstream** (ANSI/ISO-Standard : Klassen-Template basic_ifstream<>)
    für Dateien, die gelesen werden sollen und
  - **fstream** (nur AT&T-Standard, nicht ANSI/ISO-Standard)
    für Dateien, die gelesen und geschrieben werden sollen

  Diese Klassen sind von den Klassen ostream, istream und iostream abgeleitet. Sie stellen im wesentlichen einen Dateipuffer bereit und implementieren Funktionen zum Öffnen und Schließen von Dateien.
  Die in den Klassen ios und ostream bzw. istream definierten Memberfunktionen zur Ein- und Ausgabe (einschließlich der überladenen Operatorfunktionen << bzw, >>) und zur Formatbeeinflussung lassen sich auch auf Objekte der File-Stream-Klassen anwenden. Gleiches gilt für die Manipulatoren sowie für selbst überladene Ein-/Ausgabe-Operatoren.
  ⇒ Die Datei-Ein- u. Ausgabe kann in gleicher Weise wie die Geräte-Ein-/Ausgaben erfolgen.

- Zur Verwendung der File-Stream-Klassen ist die Header-Datei
  <fstream>
  einzubinden. Durch diese wird auch die Header-Datei <iostream > eingebunden.

- Wird dem Konstruktor einer der File-Stream-Klassen ein Dateipfad als Parameter übergeben, so öffnet dieser die dadurch referierte Datei.
  ⇒ Automatisches (implizites) Öffnen einer Datei bei der Definition eines File-Stream-Objektes.

- Der jeweilige Destruktor einer der File-Stream-Klassen schließt die mit dem File-Stream-Objekt assoziierte Datei.
  ⇒ Automatisches (implizites) Schließen einer Datei bei der Zerstörung des File-Stream-Objektes.

  Beispiel :

```
#include <fstream >
using namespace std;
int main() {
   ifstream eindat("atest.txt");       // Öffnen der Datei zum Lesen
   // ...
   return 0;
}                                      // Schließen der Datei
```

- Ein Mißerfolg beim Öffnen einer Datei führt zum Setzen von badbit im Status-Wort des Stream-Objekts.
  ⇒ Erfolgsüberprüfung nach jedem Öffnen.

C++-Stream-I/O : Demo-Programm zum impliziten Datei-Öffnen/Schließen

```
// ------------------------------------------------------------------------------------------------
// Programm IFODEMO
// ------------------------------------------------------------------------------------------------
// Demonstrationsprogramm zum impliziten Öffnen und Schließen von Dateien
// ------------------------------------------------------------------------------------------------
// Kopieren zweier Dateien, Dateipfade werden als Parameter übergeben
// ------------------------------------------------------------------------------------------------
```

```
#include <fstream>
#include <stdlib.h>
using namespace std;

#define Qdatei  "vonmir.txt"
#define Zdatei  "zudir.txt"

void error(char *s, char *s2="")  {
   cerr << endl << s << ' ' << s2 << endl;
   exit(1);
}

int main()  {
   ifstream quelle(Qdatei, ios::nocreate);     //implizites Öffnen der
                                               //Quelldatei
   if (!(quelle))
      error("Eingabe-Datei kann nicht geöffnet werden : ", Qdatei);
   else  {
      ofstream ziel(Zdatei);       // implizites Öffnen der Zieldatei
      if (!ziel)
         error("Ausgabe-Datei kann nicht geöffnet werden : ", Zdatei);
      else  {
         char c;
         while (quelle.get(c)) {
            ziel.put(c);
            cout << c;
         }
         cout << endl << "Datei " << Qdatei << " nach " << Zdatei
              << " kopiert !" << endl;
      }
   }            // implizites Schließen von Zdatei

   return 0;
}               // implizites Schließen von Qdatei
```

Stream-I/O : Dateibearbeitung - Öffnen/Schließen von Dateien (1)

- Für die File-Stream-Klassen existiert auch jeweils ein Default-Konstruktor.
  ⇒ Es lassen sich auch File-Stream-Objekte ohne initialisierte Kopplung an eine Datei erzeugen.

  Diese Kopplung kann dann durch explizites Öffnen einer Datei mit der für die Klassen ofstream und ifstream (und fstream) definierten - Member-Funktion `open()` hergestellt werden.
  Auch ein explizites Schließen einer Datei ist - bei gleichzeitiger Aufhebung der File-Stream-Objekt-Kopplung - möglich. Hierfür existiert die Member-Funktion `close()`.

- Memberfunktionen der File-Stream-Klassen zum Öffnen und Schließen von Dateien :

```
void open(const char *pfad,
          ios::openmode mode= default_ modus,
          int prot=filebuf::openprot)
```

- Öffnen der durch den Parameter pfad referierten Datei für den durch den Parameter mode festgelegten Bearbeitungsmodus.
- Kopplung des File-Stream-Objekts, für den diese Funktion aufgerufen wird, an die geöffnete Datei (⇒ "Öffnen des Streams").
- Setzen von badbit im Statuswort des Stream-Objektes, wenn das Öffnen fehlschlägt ⇒ durch Auswertung des Stream-Zustands nach dem Aufruf von open() kann der Erfolg der Öffnungsoperation überprüft werden.
- Default-Wert für den Bearbeitungs-Modus (Parameter mode) : default_modus = ios::out für ostream-Objekte (Öffnen zum Schreiben) default_modus = ios::in für istream-Objekte (Öffnen zum Lesen)
- Der 3. Parameter prot legt die Zugriffsberechtigung für eine durch das Öffnen neu erzeugte Datei fest.
  Meist wird dieser Parameter nicht explizit angegeben, sondern mit dem in der Klasse filebuf definierten Default-Wert gearbeitet. Im ANSI/ISO-Standard ist dieser Parameter nicht mehr vorgesehen.

```
void close(void)
```

- Schließen der an das File-Stream-Objekt, für das die Funktion aufgerufen wird, gekoppelten Datei.
- Aufheben der Kopplung zwischen Stream-Objekt und Datei

```
int is_open(void) const
```

- Überprüfung, ob das File-Stream-Objekt an eine Datei gekoppelt ist.
- Funktionswert = 1, wenn an Datei gekoppelt
  = 0, wenn an keine Datei gekoppelt

Stream-I/O : Demo-Programm zum expliziten Datei-Öffnen/Schließen

```
// ------------------------------------------------------------------------
// Programm EFODEMO
// ------------------------------------------------------------------------
// Demonstrationsprogramm zum expliziten Öffnen und Schließen von Dateien
// ------------------------------------------------------------------------
// Ausgabe des Inhalts mehrerer Dateien an die Standardausgabe
// Fehlermeldungen ("Datei kann nicht geoeffnet werden") werden in einer
// temporären Log-Datei gesammelt und am Ende ausgegeben
// ------------------------------------------------------------------------
#include <fstream>
#include <stdio.h>
using namespace std;
#define LOGDATEI "error.log"

int main(void)  {
     char *pArr[]= {    "Datei1.txt",
                        "Datei2.txt",
                        "Datei3.txt" };
     int arg = sizeof(pArr)/sizeof(char*);

     ofstream errlog(LOGDATEI);            // implizites Öffnen
     ifstream eindat;  // erzeugt File-Stream-Objekt ohne Datei-Öffnen
     char c;
     for(int i=0; i<arg; i++)  {
          eindat.open(pArr[i],ios::nocreate);      // explizites Öffnen
          if (eindat)  {
               while (eindat.get(c))
                    cout.put(c);
               eindat.close();                     // explizites Schließen
               }
          else
               errlog   << "\nDatei \"" << pArr[i]
                        << "\" ist nicht zu oeffnen\n";
          eindat.clear();                  // Fehler-Flags rücksetzen
     }
     errlog.close();                       // explizites Schließen
     eindat.open(LOGDATEI);                // explizites Öffnen
     while(eindat.get(c))
          cout.put(c);
     eindat.close();                       // explizites Schließen
     remove(LOGDATEI);
     return 0;
}
```

Ergebnis eines Probelaufes:

```
Text der
Datei 1
Datei 2
enthaelt diesen Text

Datei "Datei3.txt" ist nicht zu oeffnen
Press any key to continue
```

Stream-I/O : Dateibearbeitung - Öffnen/Schließen von Dateien (2)

- Der Datei-Öffnungs-Modus wird durch Modus-Flags beschrieben. Diese sind durch den Wertevorrat eines in der Klasse **ios** (bzw ios_base) definierten Aufzählungstyps (open_mode bzw. openmode) festgelegt.

  Folgende Modus-Flags sind definiert:

Modus -Flag	Bedeutung	
in	Öffnen zum Lesen (default bei ifstream)	
out	Öffnen zum Schreiben (default bei ofstream)	
app (append)	Schreiben nur am Dateiende (Positionieren und Schreiben vor dem ursprünglichen Dateiende nicht möglich)	
ate (atend)	Positionieren ans Dateiende (späteres Positionieren und Schreiben auch vorher möglich)	
trunc (truncate)	alten Dateinhalt löschen (default bei out, wenn weder app noch ate angegeben sind)	
binary	Öffnen im Binärmodus	
text	Öffnen im Textmodus (default)	im ANSI/ISO-Standard nicht definiert
nocreate	nur Öffnen einer existierenden Datei	
noreplace	nur Öffnen einer neuen Datei	

- Bei der Verwendung der Modus-Flags muß ihr vollqualifizierter Name angegeben werden (z.B. ios::app).

- Zur Angabe des Öffnungs-Modus können - soweit sinnvoll - mehrere Modus-Flags miteinander - bitweise-oder- verknüpft werden.
  Im allgemeinen wird dies auch der Fall sein, denn nur die Modus-Flags ios::in und ios::out können allein angegeben werden. Alle übrigen sind nur in Verbindung mit ios::in oder ios::out sinnvoll zu verwenden.

  Beispiele :

```
#include <fstream >
using namespace std;
int main(void) {
   ifstream org("beisp.txt", ios::in);            // ios::in ist default !
   ifstream ein("f:\double.dat", ios::in | ios::binary);
   ofstream aus("c:\div\journal.erg", ios::out | ios::app);
   // ……
   return 0;
}
```

Stream-I/O : Dateibearbeitung - Öffnen/Schließen von Dateien (3)

- Öffnen einer Datei zum Lesen und Schreiben:

  * Bei Verwendung von Objekten der von der Klasse iostream abgeleiteten Klasse **fstream** lassen sich Dateien gleichzeitig zum Lesen und Schreiben öffnen. Diese Dateien können dann tatsächlich sowohl gelesen als auch geschrieben werden (sinnvoll i.a. nur mit wahlfreier Positionierung).
    Da weder für den Konstruktor noch für die open-Funktion der Klasse fstream ein Default-Wert für den Öffnungs-Modus definiert ist, muß dieser (ios::in|ios::out) beim Öffnen explizit angegeben werden:

```
#include <fstream>
using namespace std;
int main()  {
   fstream einaus("beispiel.txt", ios::in|ios::out);
   // ¨¨¨
   return 0;
}
```

  * Eine gleichzeitig zum Lesen und Schreiben geöffnete Datei kann auch durch die Kopplung zweier Streams über einen gemeinsamen Buffer realisiert werden.

  * Anmerkung:
    Auch für Stream-Objekte der Klassen ofstream und ifstream lassen sich Dateien gleichzeitig sowohl zum Lesen als auch zum Schreiben öffnen. Allerdings sind die dann tatsächlich mit einer derartigen Datei ausführbaren Operationen durch die Klasse festgelegt (ofstream nur Schreiben, ifstream nur Lesen).

9.9.1 C++-Stream-I/O : Dateibearbeitung - Wahlfreier Zugriff (1)

- In den Stream-Klassen ostream (bzw basic_ostream) und istream (bzw basic_istream) sind Member-Funktionen zum Setzen und Ermitteln der aktuellen Datei-Bearbeitungsposition definiert (⇒ Positionierfunktionen). Diese Funktionen ermöglichen einen wahlfreien Dateizugriff.
  Sie sind jeweils getrennt für die Schreibposition und für die Leseposition vorhanden.

  Anwendbarkeit der Positionierfunktionen :
  * auf Objekte der Klasse ostream (und damit auch der Klasse ofstream) nur die Funktionen für die Schreibposition
  * auf Objekte der Klasse istream (und damit auch der Klasse ifstream) nur die Funktionen für die Leseposition
  * auf Objekte der Klasse iostream (und damit auch der Klasse fstream) beide Funktionsgruppen - sofern sie zum Lesen und Schreiben geöffnet sind

  Die Funktionen lassen sich sinnvoll nur auf Stream-Objekte, die an Dateien (und nicht an Geräte) gekoppelt sind, anwenden.

- Die Positionierfunktionen verwenden die folgenden Datentypen:

  * in der Headerdatei <iostream> außerhalb jeder Klasse definiert:
    - streampos: ganzzahliger Datentyp zur Darstellung der Bearbeitungsposition, meist gleich dem Datentyp long (typedef).
    - pos_type: Bezeichnung dieses Datentyps im ANSI/ISO-Standard.
    - streamoff : ganzzahliger Datentyp zur Darstellung des Offsets der Bearbeitungsposition (gegenüber einer Bezugsposition), meist gleich dem Datentyp long (typedef)

  * in der Klasse ios (bzw ios_base) definiert :
    - seekdir : (seek_dir), Aufzählungstyp zur Beschreibung der Bezugsposition, bestehend aus den Werten :
      ¨ beg - Dateianfang ist Bezugsposition
      ¨ cur - aktuelle Bearbeitungsposition ist Bezugspos.
      ¨ end - Dateiende ist Bezugsposition

C++-Stream-I/O : Dateibearbeitung - Wahlfreier Zugriff (2)

- Memberfunktionen zum Positionieren (Positionierfunktionen) :

```
istream& seekg(streampos pos)            // Klasse istream
ostream& seekp(streampos pos)            // Klasse ostream
```

  - Setzen der Bearbeitungsposition für Lesen (seekg()) bzw Schreiben (seekp()) auf die absolute Position pos (Bytes relativ zum Dateianfang)
  - Bei Textdateien ist nur dann eine richtige Arbeitsweise sichergestellt, wenn pos ein mittels tellg() bzw tellp() ermittelter Wert ist

```
istream& seekg(streamoff offset, ios::seekdir origin)// Klasse
                                                     // istream
ostream& seekp(streamoff offset, ios::seekdir origin)// Klasse
                                                     // ostream
```

  - Verändern der Bearbeitungsposition für Lesen (seekg()) bzw Schreiben (seekp()) um offset Bytes relativ zur Bezugsposition origin
  - seekg(bytezahl, ios::beg) entspricht seekg(bytezahl) seekp(bytezahl, ios::beg) entspricht seekp(bytezahl)
  - Bei Textdateien ist nur dann eine richtige Arbeitsweise sichergestellt, wenn origin=ios::beg und offset ein von tellg() bzw tellp() zurückgegebener Wert ist

```
streampos tellg(void)                    // Klasse istream
streampos tellp(void)                    // Klasse ostream
```

  - Ermittlung der aktuellen absoluten Bearbeitungsposition für Lesen (tellg()) bzw Schreiben (tellp())
  - Rückgabe dieser Bearbeitungsposition als Funktionswert

- Anmerkungen :
  - Ein Setzen der Bearbeitungsposition vor den Dateianfang ist ein Fehler - Ein Setzen der Leseposition hinter das aktuelle Dateiende ist ein Fehler
  - Ein Setzen der Schreibposition bei einer mittels ios::out|ios::app (append) geöffneten Datei vor das urprüngliche Dateiende ist ein Fehler
  - Ein Setzen der Schreibposition hinter das aktuelle Dateiende ist zulässig. Ein anschließendes Schreiben an dieser Position führt zu einem dazwischen liegenden Dateibereich mit undefinierten Werten
  - Ein Positionierfehler führt zum Setzen von badbit im Status-Wort des Stream-Objekts. Allerdings erkennen manche C++-Systeme einen Positionierfehler erst bei einem anschließenden Lese- bzw Schreib-Versuch

C++-Stream-I/O : Demo-Programm zum wahlfreien Dateizugriff

```
// ------------------------------------------------------------------------
// Programm SEEKDEMO
// ------------------------------------------------------------------------
// C++-Demonstrationsprogramm zum wahlfreien Dateizugriff
// ------------------------------------------------------------------------
// Darstellung und Überschreiben des Inhalts beliebiger Dateipositionen
// ------------------------------------------------------------------------
  #include <limits.h>
  #include <fstream>
  #include <iomanip>
  #include <stdlib.h>
  using namespace std;

  void change_bytes_in_file(fstream& datei)  {
       char c;
       int wert;
       streampos pos;
       int laenge;
       datei.seekg(0,ios::end);
       laenge=datei.tellg();
       cout << "\nDateilaenge : " << laenge << " Bytes\n";
       cout.setf(ios::uppercase);
       while (cout << "\nPosition ? ", cin >> dec >> pos)  {
            cin.ignore(INT_MAX, '\n');
            if (pos>=0 && pos<laenge)  {
                 datei.seekg(pos);
                 datei.get(c);
                 wert=c&0xff;
                 cout << "alt : " << hex << setfill('0') << setw(2)
             << wert << "   neu ? ";
                 if (cin.peek() != '\n')  {
                          cin >> hex >> wert;
                          c=wert;
                          datei.seekp(pos);
                          datei.put(c);
                 }
            }
            else
                 cout << "Position außerhalb der Datei !\n";
       } // end of while
}

int main()  {
  fstream mDatei("probe.dat", ios::in|ios::out|ios::binary);
  if (!mDatei)  {
    cout << "\nDatei " << "probe.dat" << " kann nicht geöffnet werden\n";
    return 1;
  } else  {
    change_bytes_in_file(mDatei);
    return 0;
  }
}
```

10 Ausgewählte Komponenten der Standardbibliothek

10.1 ANSI/ISO-C++-Standardbibliothek - Allgemeines

- **Grundsätzliches**

  ◇ Mit dem **ANSI/ISO-Standard** für C++ ist neben der eigentlichen Sprache auch eine dazugehörige **Standard-Bibliothek** genormt worden.

  ◇ Die C++-Standardbibliothek verwendet eine Reihe neuerer erst **relativ spät in die Sprache übernommener Sprachmittel** (z.B. Namespaces, Exception Handling, Templates, Datentyp `bool`, neue Typwandlungs-Operatoren).Daher ist sie u. U. noch nicht in allen vorhandenen C++-Systemen vollständig realisiert.
  Die Standardbibliothek wurde nicht von Grund auf neu entwickelt, sondern fasst zahlreiche schon vorher weitgehend unabhängig voneinander entstandene Komponenten und Ansätze zusammen, die soweit wie möglich aneinander angepasst, sowie um zahlreiche Vorschläge und Mechanismen ergänzt und erweitert wurden.

  Zusätzlich wurde auch die **ANSI-C-Standardbibliothek übernommen**.
  Diese ist somit - z. T. geringfügig angepasst - Bestandteil der ANSI-C++- Standardbibliothek.
  Die **C++-Standardbibliothek** stellt daher **kein homoges Gebilde** dar, sondern besteht aus **verschiedenen Teilen**, die weitgehend unabhängig voneinander sind, die aber doch in einigen Details miteinander verbunden sind.

  ◇ Alle in der C++-Standardbibliothek definierten **globalen Namen** - auch die in der integrierten ANSI-C-Standardbibliothek definierten - sind im **Namensbereich `std`** definiert.

- **Konventionen für Header-Dateien:**

  ◇ Header-Dateien werden in der `#include`-Anweisung **ohne Namens-Extension** angegeben, z. B.

```
#include <iostream>
```

  ◇ Auch die **Header-Dateien der C-Standardbibliothek** werden **ohne Extension** angegeben, allerdings wird ihnen zur Unterscheidung **ein c vorangestellt**, z.B.

```
#include <cstdlib>        // C-Header-Datei stdlib.h
```

  ◇ Die Angabe der Header-Dateinamen ohne Extension bedeutet nicht, dass Header-Dateien generell keine Namens-Extension mehr besitzen.Vielmehr ist die Umsetzung des in der `#include`-Anweisung verwendeten Dateinamens in den tatsächlich im jeweiligen C++-System verwendeten Dateinamen implementierungsabhängig. z.b. kann

```
#include <iostream>
```

  vom Preprozessor umgesetzt werden in

```
namespace std
{
    #include <iostream.h>
}
```

  ◇ Aus **Kompatibilitätsgründen** können für die **C-Header-Dateien** auch die **"normalen" Namen mit Extension** verwendet werden.Dies gilt in der Praxis - wenn auch nicht im Standard festgelegt - ebenfalls für C++-spezifische Header-Dateien, die bereits vor der Definition der Standardbibliothek verwendet wurden, z.B.

```
#include <iostream.h>
```

  Werden **Header-Datei-Namen mit Extension** verwendet, so befinden sich die definierten globalen Namen im **globalen Namensbereich** (nicht `std`)

ANSI/ISO-C++-Standardbibliothek – Überblick(1)

- **Bestandteile der Standardbibliothek** :

  ◇ **Sprachunterstützungs-Bibliothek (*Language support library*)**

  Funktionen und Typen, die zur Realisierung bestimmter Sprachelemente benötigt werden, wie z.B.
  Funktionen, die während der Ausführung von C++-Programmen implizit aufgerufen werden, Definition zugehöriger verwendeter Typen, Festlegung implementierungsabhängiger Eigenschaften der Standardtypen
  (u.a. Klassen-Template **`numeric_limits`**)

  ◇ **Diagnose-Bibliothek (*Diagnostics library*)**

  Komponenten zum Entdecken und Melden von Fehler-Bedingungen,
  u.a. Hierarchie von Fehlerklassen (***exception classes***), Macro **`assert`**, Fehlernummern-Macros

  ◇ **Utility-Bibliothek (*General utility library*)**

  Ergänzende Hilfsmittel, u.a. Datentypen für Wertepaare (Klassen-Template **`pair`**) und "sichere" Pointer (Klassen-Template **`auto_ptr`**), Default-Allokator-Klasse **`allocator`** (Allokator-Objekte repräsentieren Speichermodelle)

  ◇ **String-Bibliothek (*Strings library*)**

  Klassen-Template **`basic_string`** und konkrete Realisierungen (Template-Klassen) **`string`** und **`wstring`**

  ◇ ***Localization library***

  Komponenten zur portablen Unterstützung nationaler Eigenheiten
  (z.B. Zeichendarstellung und Zeichensatz, Datums-, Währungs-, Zahlformate), u.a. Klasse **`locale`**

  ◇ **STL - *Standard Template Library***

  Bibliothek zur Verwaltung und Bearbeitung von Mengen beliebigen Typs.
  Sie besteht aus 3 Teilen :

  ▻ ***Containers library***
  Container-Klassen (Objekte dieser Klassen speichern und verwalten andere Objekte)

  ▻ ***Iterators library***
  Iterator-Klassen (Iteratoren dienen zum Zugriff zu einzelnen Elementen einer Objektmenge, insbesondere zu Elementen von Containern)

  ▻ ***Algorithms library***
  "freie" Funktionen zur Realisierung von Algorithmen, mit denen Mengen und Elemente in Mengen bearbeitet werden können.

  In diesem Teil der Bibliothek sind auch Komponenten enthalten, die nicht direkt auf Objekt-Mengen bezogen sind, z. B.
  - Klassen-Template **`bitset`** zur Verwaltung und Bearbeitung von Bit-Feldern fester aber beliebig wählbarer Größe (*Containers library*)
  - "freie" Funktions-Templates zur Realisierung häufig benötigter Hilfsfunktionen, wie z. B. `max()`, `min()`, `swap()`

  (*Algorithms library*)

◇ **Numerische Bibliothek (*Numerics library*)**

Klassen-Templates für komplexe Zahlen (**`complex`**) und numerische Felder, wie z.B. Vektoren und Matrizen (**`valarray`**), Erweiterung der numerischen Funktionen der C-Standard-Bibliothek durch Überladen (freie Funktionen)

◇ **Stream-I/O-Bibliothek (*Input/Output library*)**

Stream-Klassen (einschließlich File- u. String-Streams), Standard-Stream-Objekte, Stream-Buffer-Klassen, Standard-Manipulatoren

ANSI/ISO-C++-Standardbibliothek – Überblick Header-Dateien(2)

Language support library Typen Implementierungsabhängige Eigenschaften Programm-Start u. Beendigung Dynamische Speicherverwaltung Typidentifikation Ausnahmebehandlung weitere Laufzeitunterstützung	**<cstddef>** **<limits>, <climits>, <cfloat>** **<cstdlib>** **<new>** **<typeinfo>** **<exception>** **<cstdarg>, <csetjmp>, <ctime>,** **<csignal>, <cstdlib>**
Diagnostics library Exception-Klassen Assertions Fehlernummern	**<stdexcept>** **<cassert>** **<cerrno>**
General utilities library Utility-Komponenten Funktions-Objekte Speicher Datum und Zeit	**<utility>** **<functional>** **<memory>** **<ctime>**
Strings library Zeicheneigenschaften (*char traits*) String-Klassen NUL-terminierte Zeichenfolgen	**<string>** **<string>** **<cctype>, <cwctype>, cstring>,** **<cwchar>, <cstdlib>**
Localization library Locales-Komponenten u. Kategorien C-Bibliotheks-Locales	**<locale>** **<clocale>**
Containers library Folgen (sequences) Assoziative Container Bitset	**<deque>, <list>, <queue>, <stack>,** **<vector>** **<map>, <set>** **<bitset>**
Iterators library Iteratoren	**<iterator>**
Algorithms library Diverse Algorithmen C-Bibliotheks-Algorithmen	**<algorithm>** **<cstdlib>**
Numerics library Komplexe Zahlen Numerische Felder Numerische Operationen Numerische C-Bibliothek	**<complex>** **<valarray>** **<numeric>** **<cmath>, <cstdlib>**

Input/Output library	
Forwärts-Deklarationen	**<iosfwd>**
Standard-Stream-Objekte	**<iostream>**
I/O-Stream-Basisklassen	**<ios>**
Stream-Buffer	**<streambuf>**
Formatierung und Manipulatoren	**<istream>, <ostream>, <iomanip>**
String-Streams	**<sstream>, <cstdlib>**
File-Streams	**<fstream>, <cstdio>, <cwchar>**

10.2 C++-Standardbibliothek : Standard-Exception-Klassen

- **Grundsätzliches**
  - ◇ Die ANSI-C++-Standardbibliothek stellt auch einige Exception-Klassen zur Verfügung.
    Diese bilden eine **Klassen-Hierarchie**, die von der Klasse **`exception`** abgeleitet ist.
  - ◇ Einige dieser Exception-Klassen gehören zur **Sprachunterstützungsbibliothek** (*Language Support Library*).
    Sie werden von **Sprachkonstrukten** verwendet (z.B. von `new, dynamic_cast, typeid`)
  - ◇ Die übrigen Exception-Klassen werden von der **Standardbibibliothek selbst** benutzt. Sie sind – mit einer Ausnahme – in der **Diagnosebibliothek** (*Diagnostics Library*) definiert.
    Objekte dieser Exception-Klassen können auch im **Anwender-Code** geworfen werden.
    Darüber hinaus lassen sie sich auch als **Basisklassen** für **selbstdefinierte Exception-Klassen** einsetzen.

- **Die Exception-Basisklasse `exception`**
  - ◇ Bestandteil der Sprachunterstützungsbibliothek.
  - ◇ Sie ist die Basisklasse für alle Objekte, die als Exceptions von einigen Sprachausdrücken sowie von Bestandteilen der Standardbibliothek geworfen werden können.
  - ◇ Ihre **öffentliche Schnittstelle** ist in der **Headerdatei `<exception>`** wie folgt **definiert :**

```
class exception
{
  public :
    exception() throw();
    exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
  private :
    // ...
};
```

  - ◇ Die virtuelle **Methode** `const char* what() const` dient zur Ermittlung einer Information über die **Ursache** einer Exception. Der von ihr für die Klasse `exception` zurückgelieferte C-String ist **implementierungsabhängig**.
    Häufig wird in den von `exception` abgeleiteten Bibliotheks-Klassen diese Methode jeweils – ebenfalls implementierungsabhängig – überschrieben.
    Auch in selbst definierten Exception-Klassen, die von einer der Bibliotheks-Exception-Klassen abgeleitet sind, kann sie – und muss sie gegebenenfalls – in geeigneter Weise überschrieben werden.
    Bei den Exception-Klassen, deren Instanzen in Sprachausdrücken geworfen werden können, ist die zurückgelieferte Information häufig direkt in der Methode `what()` implementiert, bei den von Komponenten der Standardbibliothek verwendeten Klassen lässt sich diese Information bei der Exception-Objekt-Erzeugung dem Konstruktor übergeben.

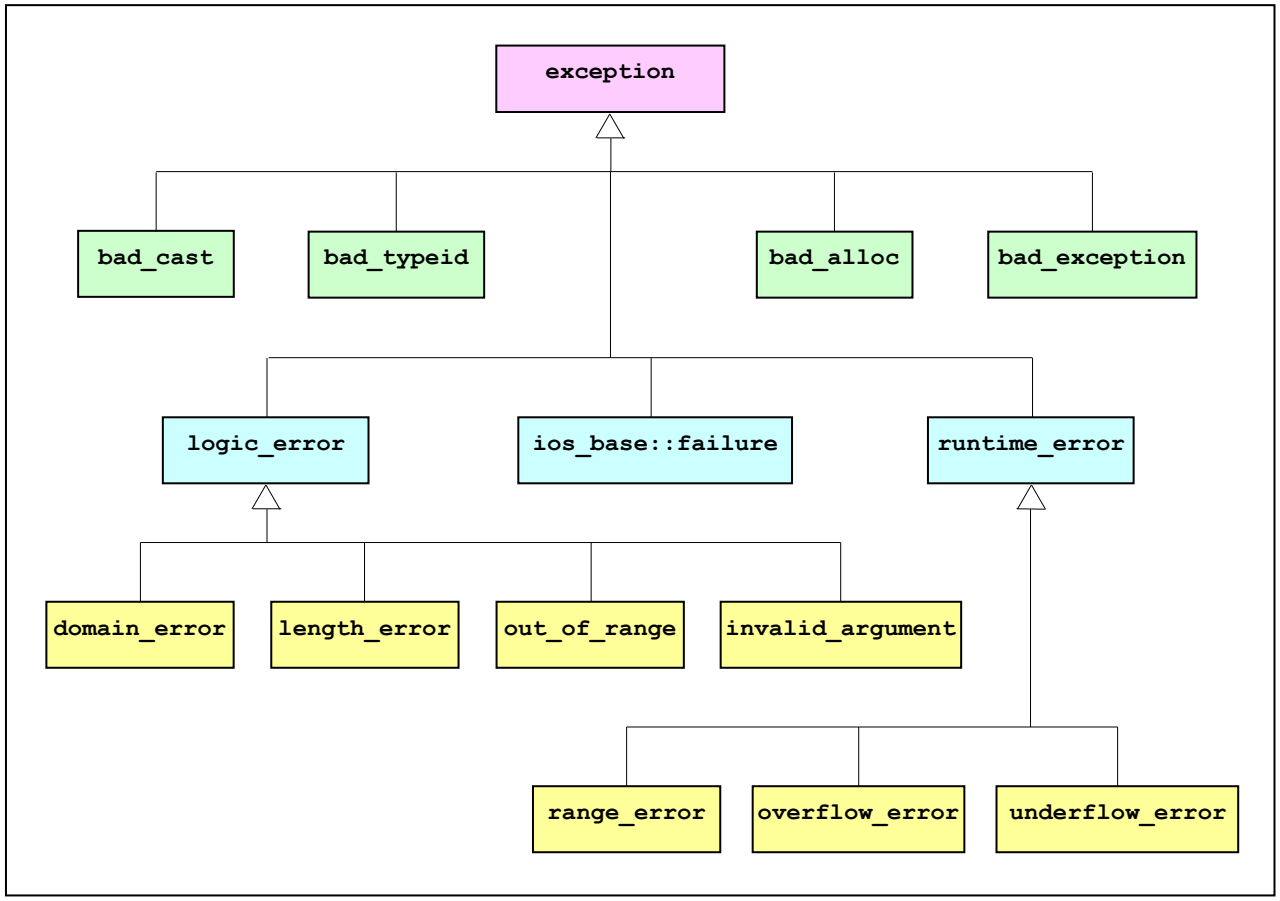
    → **portable Auswertung der Exception-Information** :

```
try
{
  // ...
}
catch (const exception& ex)
{
  cerr << endl << ex.what() << endl;
  // ...
}
```

- ◇ Die **Exception-Spezifikation `throw()`** der verschiedenen Memberfunktionen der Klasse `exception` legen fest, dass diese **selbst keine Exceptions werfen** dürfen.

ANSI/ISO-C++-Standardbibliothek : Standard-Exception-Klassen (2)

- **Hierarchie der Bibliotheks-Exception-Klassen**



- **Die Exception-Klassen zur Sprachunterstützung**

Klassenname	Headerdatei	geworfen von ... bei
bad_cast	<typeinfo>	Operator **dynamic_cast** bei ungültigem Referenz-Cast
bad_typeid	<typeinfo>	Operator **typeid** bei dereferenziertem NULL-Pointer im Operandenausdruck
bad_alloc	<new>	Operator **new** bei fehlgeschlagener Allokation (falls entsprechend implementiert)
bad_exception	<exception>	unter gewissen Umständen durch die Funktion `unexpected()`, wenn durch eine Funktion eine **nicht vorgesehene Exception** (Exception-Spec) geworfen wird

◇ Die **öffentliche Schnittstelle** für alle Exception-Klassen dieser Gruppe entspricht der Schnittstelle von `exception` :

- ▻ Konstruktor ohne Parameter
- ▻ Copy-Konstruktor
- ▻ Zuweisungsoperatorfunktion
- ▻ virtueller Destruktor
- ▻ virtuelle Memberfunktion `const char*` **`what()`** `const throw()` zur Ermittlung von Information über die Exception-Ursache

ANSI/ISO-C++-Standardbibliothek : Standard-Exception-Klassen (3)

- **Die in der Standardbibliothek eingesetzten Exception-Klassen**

  ◇ Aus dieser Gruppe sind **drei Klassen direkt** von der Klasse **`exception`** **abgeleitet**, zwei in der Diagnosebibliothek und eine in der IO-Bibliothek. Von den Klassen der Diagnosebibliothek sind weitere abgeleitete Klassen definiert.

  ◇ Klasse **`logic_error`**

  ▻ definiert in der Headerdatei **`<stdexcept>`** (Diagnosebibibliothek)

  ▻ dient als **Basisklasse** für **weitere Exceptionklassen**, die den Auftritt **logischer Fehler** charakterisieren.
  Unter logischen Fehlern werden solche Fehler verstanden, die in der Programmlogik liegen und damit prinzipiell schon **vor dem Programmstart** oder durch das Überprüfen von Funktionsparametern gefunden werden können.

  Die folgenden von dieser Klasse **abgeleiteteten Klassen** sind ebenfalls in der Headerdatei **`<stdexcept>`** definiert:

  ▻ Klasse **`domain_error`**
  Objekte dieser Klasse werden beim Auftritt von Domänenfehlern (Wertebereichsfehlern) geworfen.

  ▻ Klasse **`length_error`**
  Objekte dieser Klasse werden geworfen, wenn versucht wird, ein Objekt zu erzeugen, dessen Größe die maximal zulässige Größe überschreiten würde (z. B. durch einige Memberfunktionen der Bibliotheksklasse `string`).

  ▻ Klasse **`out_of_range`**
  Objekte dieser Klasse können geworfen werden, wenn ein Funktionsparameter außerhalb des zulässigen Bereichs liegt (vor allem bei Positionsangaben, z. B. durch einige Memberfunktionen der Bibliotheksklasse `string` zur Auswahl von Stringelementen)

  ▻ Klasse **`invalid_argument`**
  Objekte dieser Klasse können geworfen werden, wenn einer Funktion ein Parameter mit einem ungültigen Wert übergeben wird (z.B. durch den Konstruktor des Klassen-Templates `bitset<>,` wenn ihm ein Stringparameter übergeben wird, der andere Zeichen als `'0'` und `'1'` enthält)

  ◇ Klasse **`runtime_error`**

  ▻ ebenfalls definiert in der Headerdatei **`<stdexcept>`** (Diagnosebibliothek)

  ▻ dient als Basisklasse für weitere Exceptionklassen, die den Auftritt von **Laufzeitfehlern** kennzeichnen.
  Laufzeitfehler sind Fehler, die prinzipiell **erst zur Laufzeit** erkannt werden können (z.B. Divison durch `0`).

  Die folgenden von dieser Klasse **abgeleiteten Klassen** sind ebenfalls in der Headerdatei **`<stdexcept>`** definiert :

  ▻ Klasse **`range_error`**
  Objekte dieser Klasse kennzeichnen Bereichsfehler, die bei arithmetischen Berechnungen auftreten können.

  ▻ Klasse **`overflow_error`**
  Objekte dieser Klasse können geworfen werden, wenn ein arithmetischer Überlauf auftritt
  (z.B. in einer Memberfunktion des Klassen-Templates `bitset<>,` mit der ein Bitset in einen `unsigned long` Wert umgewandelt werden soll, wenn das Bitset nicht als `unsigned long` dargestellt werden kann)

  ▻ Klasse **`underflow_error`**
  Objekte dieser Klasse können geworfen werden, wenn ein arithmetischer Unterlauf auftritt

  ◇ Klasse **`ios_base::failure`**

  ▻ definiert in der Headerdatei **`<ios>`** innerhalb der Klasse `ios_base` (IO-Bibliothek)
  Die Headerdatei `<ios>` wird durch die Headerdatei `<iostream>` eingebunden.

  ▻ Objekte dieser Klasse (und davon abgeleiteter Klassen) werden zur Kennzeichnung von Fehlersituationen in der IO-Bibliothek geworfen.

  ◇ Für alle Klassen dieser Gruppe ist ein **Konstruktor** definiert, dem ein **`string`-Objekt**, das die **Fehlerursache** kennzeichnet, als Parameter zu übergeben ist.
  Beispiel : **`explicit logic_error(const string& what_arg);`**

In realen Implementierungen sind i.a. zusätzlich definiert :

- virtueller Destruktor
- virtuelle Memberfunktion `const char* what() const throw()`

Für die Klasse `ios_base::failure` sind diese Memberfunktionen auch in der ANSI-Norm vorgesehen.

Standard-Exception-Klassen - einfaches Demonstrationsprogramm (1)

- **Modul mit exception-werfender Funktion (C++-Quelldatei `testbibexcept_func.cpp`)**

```
#include <iostream>
#include <fstream>
#include <exception>
#include <stdexcept>
#include <typeinfo>
#include <new>
using namespace std;

class Base
{ public : virtual ~Base() {}; /* ... */ };

class Derived : public Base
{ public : virtual ~Derived() {}; /* ... */ };

class Der2 : public Base
{ public : virtual ~Der2() {}; /* ... */ };


void func1(int i)
{ switch(i)
  { case 0 : cout << "\ncase 0 : ";
             throw(exception());
             break;
    case 1 : cout << "\ncase 1 : ";
             throw(bad_exception());
             break;
    case 2 : cout << "\ncase 2 : ";
             { Base* bp=new Der2;
               dynamic_cast<Derived&>(*bp);    // throw(bad_cast());
             }
             break;
    case 3 : cout << "\ncase 3 : ";
             { Base* bp=NULL;
               typeid(*bp);                    // throw(bad_typeid());
             }
             break;
    case 4 : cout << "\ncase 4 : ";
             new char[0x7fffffff];            // throw(bad_alloc());
             break;
    case 5 : cout << "\ncase 5 : ";
             { ifstream istrm;
               istrm.exceptions(ios::eofbit | ios::failbit | ios::badbit);
               istrm.setstate(ios::eofbit);   // throw(ios_base::failure());
             }
             break;
    case 6 : cout << "\ncase 6 : ";
             throw(ios_base::failure("mein IO-Fehler"));
             break;
    case 7 : cout << "\ncase 7 : ";
             throw(logic_error("Exception wg. logischem Fehler"));
             break;
    case 8 : cout << "\ncase 8 : ";
             throw(runtime_error("Exception wg. Laufzeit-Fehler"));
             break;
    case 9 : cout << "\ncase 9 : ";
             throw(range_error("Exception wg. Range Error"));
             break;
    default: cout << "\ndefault: ";
```

Standard-Exception-Klassen - einfaches Demonstrationsprogramm (2)

- **Modul mit exception-fangender `main()`-Funktion (C++-Quelldatei `testbibexcept_m.cpp`)**

```
#include <iostream>
#include <exception>
#include <new.h>       // Visual-C++-spezifisch, nur fuer _set_new_handler(_PNH)
using namespace std;

#define MAX_NR 10

extern void func1(int);

int mynewhdlr(size_t)
{ throw(bad_alloc());
}


int main(void)
{ _set_new_handler(mynewhdlr);      // Visual-C++-spezifische Funktion
  for (int i=0; i<=MAX_NR; i++)
  { try
    { func1(i);
      cout << "func(" << i << ") normal beendet" << endl;
    }
    catch(const exception& e)
    { cerr << e.what() << endl;
    }
    catch(const char* cpe)
    { cerr << cpe << endl;
    }
  }
  cout << "\nWeiter gehts !\n";
```

- **Ausgabe des Programms**

```
case 0 : Unknown exception

case 1 : bad exception

case 2 : Bad dynamic_cast!

case 3 : Attempted a typeid of NULL pointer!

case 4 : bad allocation

case 5 : ios::eofbit set

case 6 : mein IO-Fehler

case 7 : Exception wg. logischem Fehler

case 8 : Exception wg. Laufzeit-Fehler
```

10.3 C++-Standardbibliothek : Klassen-Template `pair` (1)

- **Definition des Klassen-Templates `pair`**

  ◇ An einigen Stellen der Standardbibliothek werden **Wertepaare** verwendet.
  Beispielsweise verwalten die in der STL definierten Container-Klassen **`Map`** und **`Multimap`** Mengen, deren Elemente Wertepaare (Schlüssel/Wert) sind.

  ◇ Zur Unterstützung der Arbeit mit **Wertepaaren** ist in der **Utility-Bibliothek** als generischer Datentyp das – als **`struct`** realisierte – **Klassen-Template `pair`** definiert.

  ◇ Die **Definition** befindet sich in der Headerdatei **`<utility>`:**

```
template <class T1, class T2>
struct pair
{
  typedef T1 first_type;
  typedef T2 second_type;

  T1 first;                        // erster Wert des Paares
  T2 second;                       // zweiter Wert des Paares
  pair();                          // Default-Konstruktor
  pair(const T1& x, const T2& y);  // Konstruktor mit Initialisierungswerten
  template <class U, class V>
    pair(const pair<U, V>& p);     // Copy-Konstruktor
};
```

  ◇ Die **beiden Werte** eines Wertepaares sind durch zwei entsprechende **Datenkomponenten** realisiert.
  Diese sind **öffentlich** zugänglich (`public` ist Default bei `struct`), zu ihnen kann also direkt zugegriffen werden.

  ◇ Drei Konstruktoren bilden die einzigen Memberfunktionen.

- **Beschreibung der Konstruktoren**

  ◇ **`pair();`** (Default-Konstruktor)
  - ▻ Initialisierung der beiden Datenkomponenten durch Aufruf des Default-Konstruktors ihres jeweiligen Typs
  - ▻ typische Implementierung innerhalb der Klassendefinition :

```
pair() : first(T1()), second(T2()) {}
```

  ◇ **`pair(const T1& x, const T2& y);`** (Konstruktor mit Initialisierungswerten)
  - ▻ Initialisierung der beiden Datenkomponenten mit den übergebenen Parametern `x` und `y`
  - ▻ typische Implementierung innerhalb der Klassendefinition :

```
pair(const T1& x, const T2& y) : first(x), second(y) {}
```

  ◇ **`template <class U, class V> pair(const pair<U, V>& p);`** (Copy-Konstruktor)
  - ▻ Initialisierung der beiden Datenkomponenten mit den entsprechenden Datenkomponenten des Parameters `p`, wobei gegebenenfalls implizite Typkonvertierungen durchgeführt werden
  - ▻ typische Implementierung innerhalb der Klassendefinition :

```
template<class U, class V> pair(const pair<U, V> &p)
: first(p.first), second(p.second) {}
```

- **Zusätzlich definierte freie Funktionen :**

  ◇ Die Utility-Bibliothek enthält zusätzlich **7 freie Funktionen** zur Verwendung mit dem Klassen-Template `pair`. Typischerweise sind diese Funktionen in der Headerdatei `<utility>` als **`inline`-Funktionen** definiert. 6 dieser Funktionen sind **Vergleichsfunktionen,** die 7. dient zur **Erzeugung eines `pair`-Objektes**.

◇
```
template <class T1, class T2>
  inline bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y)
  { return x.first == y.first && x.second == y.second; }
```
▻ Test zweier `pair`-Objekte auf Gleichheit

▻ Zwei `pair`-Objekte sind genau dann gleich, wenn ihre beiden entsprechenden Komponenten jeweils gleich sind

---

◇
```
template <class T1, class T2>
  inline 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); }
```
▻ Test zweier `pair`-Objekte auf "kleiner als"

▻ Für den Vergleich wird primär die erste Komponente überprüft. Wenn diese für beide Objekte gleich ist, wird das Vergleichsergebnis durch Vergleich der zweiten Komponente ermittelt.

◇
```
template <class T1, class T2>
  inline bool operator!=(const pair<T1, T2>& x, const pair<T1, T2>& y)
  { return !(x == y); }
```
▻ Test zweier `pair`-Objekte auf Ungleichheit

◇
```
template <class T1, class T2>
  inline bool operator>(const pair<T1, T2>& x, const pair<T1, T2>& y)
  { return y < x; }
```
▻ Test zweier `pair`-Objekte auf "größer als"

◇
```
template <class T1, class T2>
  inline bool operator>=(const pair<T1, T2>& x, const pair<T1, T2>& y)
  { return !(x < y); }
```
▻ Test zweier `pair`-Objekte auf "größer gleich"

◇
```
template <class T1, class T2>
  inline bool operator<=(const pair<T1, T2>& x, const pair<T1, T2>& y)
  { return !(y < x); }
```
▻ Test zweier `pair`-Objekte auf "kleiner gleich"

◇
```
template <class T1, class T2>
  inline pair<T1, T2> make_pair(const T1& x, const T2& y)
  { return pair<T1, T2>(x, y); }
```
▻ Erzeugung eines `pair`-Objektes aus zwei Komponenten ohne explizite Angabe der Komponententypen

Beispiel : `return` **`make_pair(5, 3.728);`** statt `return pair<int, double>(5, 3.728);`

10.4 STL: String-Bibliothek Überblick

- **String-Bibliothek**
  - ◇ Zur ANSI-C++-Standardbibliothek gehörende Teilbibliothek für die **String-Unterstützung**.
  - ◇ Sie stellt ein **Klassen-Template** zur Verfügung, dessen Instanzen, die **String-Klassen**, eine sehr **komfortable** und **sichere** (Speicherverwaltung und Bereichskontrolle) Bearbeitung von Strings ermöglichen.
    Wesentlicher **Template-Parameter** ist der **Datentyp** zur Darstellung von **Zeichen**.
  - ◇ Strings werden also – im Unterschied zu C-Strings – als **Objekte** dargestellt und verwendet.
    Die in den Objekten enthaltene Zeichenfolge ist nicht mit einem speziellen Endzeichen (z.B. `'\0'`-Character) abgeschlossen. Vielmehr kann sie jedes Zeichen des verwendeten Zeichen-Datentyps enthalten.
    Der für die Ablage der Zeichenfolge benötigte Speicherplatz wird dem jeweiligen Bedarf entsprechend durch die Memberfunktionen des Klassen-Templates alloziert bzw freigegeben.
  - ◇ Alle für die Anwendung der String-Bibliothek notwendigen Vereinbarungen sind in der Headerdatei `<string>` enthalten. Sie liegen alle im Namensraum **`std`**.
- **Klassen-Template `basic_string`**
  - ◇ Zentrales Klassen-Template für die String-Klassen, das deren Verhalten und Fähigkeiten definiert.
  - ◇ Die aus dem Template instantiierbaren String-Klassen sind durch drei **Template-Parameter** bestimmt :
    - ▻ **`charT`** : **Datentyp** zur Darstellung der **Zeichen** (Zeichentyp)
    - ▻ **`traits`** : **Klasse** zur Beschreibung von **Eigenschaften** (Merkmalen) des **Zeichentyps** (*character traits*).
      Als **Default** ist hierfür die Instanz des ebenfalls mit dem Zeichen-Datentyp `charT` parameterisierten Klassen-Templates `char_traits`, das ist **`char_traits<charT>`**, festgelegt.
      parameterisiert ist.
      Das Klassen-Template `char_traits` ist auch in der Header-Datei `<string>` definiert.
    - ▻ **`Allocator`** : **Allokator-Klasse**, die das Speichermodell beschreibt, das die zu verwendende dynamische Speicherverwaltung festlegt.
      Als **Default** ist die Instanz des ebenfalls mit dem Zeichen-Datentyp `charT` parametrisierten Klassen-Templates `allocator`, das ist **`allocator<charT>`**, vorgesehen.
      Das Klassen-Template `allocator` bestimmt die im System eingesetzten Standard-Allokator-Klassen, die `new` und `delete` zur Speicherallokation verwenden. Seine Definition befindet sich in der Headerdatei `<memory>`.
  - ◇ **Definition** :

```
template <class charT, class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
  class basic_string { /* ... */ };
```

- **String-Klassen `string` und `wstring`**
  - ◇ Standardmäßig sind in `<string>` zwei String-Klassen als **Instanzen** von `basic_string` vordefiniert.
  - ◇ Klasse **`string`** für den Datentyp **`char`** ("normale" Ein-Byte-Zeichen) unter Verwendung der Defaults für die anderen Parameter :

```
typedef basic_string<char> string;
```

  - ◇ Klasse **`wstring`** für den Datentyp **`wchar_t`** (Mehr-Byte-Zeichen, z.B. Unicode-Zeichen) unter Verwendung der Defaults für die anderen Parameter :

```
typedef basic_string<wchar_t> wstring;
```

STL: String-Bibliothek Überblick (2)

- **Überblick über die Operationen mit Strings**
  - ◇ Die String-Bibliothek von C++ erlaubt es, dass mit Strings – im Gegensatz zu C-Strings – wie mit elementaren Datentypen gearbeitet werden kann.
    De facto steht mit der Bibliotheks-Klasse `string` (bzw. `wstring` bzw. jeder anderen Klassen-Instanz von `basic_string<>`) ein quasi-fundamentaler Datentyp zur Verfügung.
  - ◇ Das Klassen-Template **`basic_string<>`** definiert
    - ▻ grundlegende Fähigkeiten zum **Anlegen** (Initialisieren) und **Kopieren** von Strings
    - ▻ zahlreiche Methoden zum **Manipulieren** und **Bearbeiten** von Strings, wie Konkatenieren, Einfügen, Löschen, Suchen, Ersetzen, Vergleichen und Teilstringbildung.
      Die meisten dieser Methoden sind mehrfach so überladen, dass sich die entsprechenden Operationen – wo sinnvoll – auch im Zusammenhang mit C-Strings und Einzelzeichen durchführen lassen.
      Für einige der Bearbeitungsoperationen existieren neben Memberfunktionen mit einem geeigneten Namen auch entsprechende – z. T. freie – Operatorfunktionen (s. unten).
    - ▻ Methoden zum **Zugriff** zu den **einzelnen Zeichen** eines Strings (String-Elementen)
    - ▻ Methoden zum **Umwandeln** von Strings in **C-Strings**
    - ▻ Methoden zum Bereitstellen von **String-Iteratoren**
    - ▻ Methoden zur **Ermittlung** und **Änderung** der **Stringlänge** und **Stringkapazität**.
  - ◇ Weiterhin sind in der Stringbibliothek zahlreiche **freie Operatorfunktionen** für Strings definiert.
    Diese ermöglichen
    - ▻ die Stream-**Ein-** und **Ausgabe** von Strings
    - ▻ die **Konkatenation** von Strings, C-Strings und Einzelzeichen an Strings
      sowie die Konkatenation von Strings an Strings, C-Strings und Einzelzeichen
    - ▻ den **Vergleich** von Strings mit Strings, Strings mit C-Strings und C-Strings mit Strings
  - ◇ In vielen – aber nicht allen – Funktionen, in denen die Möglichkeit zu einem Speicherzugriffsfehler besteht, können **Exceptions** geworfen werden :
    - ▻ Exception der Klasse **`out_of_range`** beim Zugriff zu einer unzulässigen String-Position
    - ▻ Exception der Klasse **`length_error`** beim Versuch einen String über die maximal mögliche Länge hinaus zu verlängern

    Beide Exception-Klassen sind in der Standard-Bibliothek definiert. Sie sind von der Exception-Klasse `logic_error` abgeleitet. Diese wiederum ist von der Exception-Basis-Klasse `exception` abgeleitet.

- **Anmerkungen zur Beschreibung der String-Funktionen**
  - ◇ Zur Vereinfachung und Erhöhung der Übersichtlichkeit werden die einzelnen String-Funktionen im Folgenden nicht in der originalen Template-Form sondern in einer für die – am häufigsten verwendete – **Klassen-Instanz `string` spezialisierten Darstellung** angegeben:
    - ▻ statt dem Klassen-Template `basic_string` wird `string` eingesetzt
    - ▻ statt dem Template-Parameter `charT` wird der Zeichen-Datentyp `char` eingesetzt
  - ◇ **Beispiele**:
    - ▻ statt: `basic_string& append(const charT* s);` (Memberfunktion)
      wird formuliert: **`string& append(const char* s);`**
    - ▻ statt:
      ```
      template<class charT, class traits, class Allocator>
      basic_istream<charT, traits>&
      operator>>(basic_istream<charT, traits>& is,
      basic_string<charT, traits, Allocator>& str);
      ```
      wird formuliert: **`istream& operator>>(istream& is, string& str);`**

STL: Klasse `string` (1)

- **Datentypen als Klassenelemente**
  - ◇ Innerhalb der Klasse `string` sind mehrere Datentypen – meist indirekt – definiert. Es sind `public`-Komponenten . Sie stehen damit zur Verwendung außerhalb der Klasse `string` zur Verfügung.
  - ◇ Einer dieser Datentypen ist: **`size_type`**
    Es handelt sich um einen vorzeichenlosen ganzzahligen Datentyp, der zur Angabe von String-Positionen, Anzahl-, Längen- und Größenangaben verwendet wird. Dieser wird durch die **Allokator-Klasse** festgelegt. Für ISO-konforme Container muss dieser dem Typ **`std::size_t`** entsprechen.

- **`public`-Datenkomponente der Klasse `string`**
  - ◇ **`static const size_type npos = -1;`**
    Definition einer Konstanten, die – je nach Anwendung – als (**Default-)Parameterwert** für "**alle Zeichen**" ("bis zum Stringende") oder zur Kennzeichnung einer **ungültigen Position** verwendet wird.
    Da der Typ der Konstante (**`string::size_type`**) als vorzeichenloser ganzzahliger Datentyp definiert ist, entspricht der Wert **`-1`** dem **größtmöglichen Wert** dieses Typs.

Achtung: Außerhalb der Qualifikation ist für **`size_type`** und **`npos`** der Klassennamen `string::` mitanzugeben!●

Konstruktoren und Destruktor der Klasse `string`

`string();`	Erzeugung eines Leer-Strings (Länge `0`)
`string(const string& str, size_type pos=0, size_type n= npos);`	Erzeugung eines Strings, der mit `n` Zeichen ab der Position `pos` des Strings `str` initialisiert wird. Für die Default-Parameterwerte: Copy-Konstruktor
`string(const char* s);`	Erzeugung eines Strings, der mit dem C-String `s` initialisiert wird
`string(const char* s, size_type n);`	Erzeugung eines Strings, der mit den ersten `n` (höchstens aber allen) Zeichen des C-Strings `s` initialisiert wird
`string(size_type n, char c);`	Erzeugung eines Strings der Länge `n.` Alle Komponenten werden mit dem Zeichen `c` initialisiert
`~string();`	Destruktor

- **Memberfunktionen zur Ermittlung der Länge und Kapazität**

`size_type size() const;`	Ermittlung der aktuellen Stringlänge (Anzahl der Zeichen im String)
`size_type length() const;`	Ermittlung der aktuellen Stringlänge (Anzahl der Zeichen im String)
`size_type capacity() const;`	Ermittlung der maximal möglichen Stringlänge ohne Neuallokation
`size_type max_size() const;`	Ermittlung der maximalen Größe, die ein String haben kann
`bool empty() const;`	Ermittlung, ob String leer ist, wenn ja Rückgabe von `true`, sonst `false`

ANSI/ISO-C++-Standardbibliothek : Klasse `string` (2)

- **Memberfunktionen zur Änderung der Länge und Kapazität**

`void resize(size_type n, char c);`	Falls `n<=max_size()` : Änderung der Stringlänge auf den Wert `n` bei Stringverlängerung : Auffüllen mit dem Zeichen `c` Falls `n>max_size()` : Werfen der Exception **`length_error`**
`void resize(size_type n);`	Falls `n<=max_size()` : Änderung der Stringlänge auf den Wert `n` bei Stringverlängerung : die zusätzlichen Zeichen bleiben undefiniert Falls `n>max_size()` : Werfen der Exception **`length_error`**
`void clear();`	Löschen aller Zeichen im String, neue Stringlänge = `0`
`void reserve(size_type res=0);`	Falls `res>capacity()` : Vergrößerung der Kapazität Falls `res<=max_size()` : neue Kapazität `>= res` Falls `res>max_size()` : Werfen der Exception **`length_error`** Falls `res<=capacity()` : keine Wirkung

- **Memberfunktionen zur Zuweisung**

Rückgabewert für alle Zuweisungsfunktionen : **Referenz** auf das **aktuelle** String-Objekt (`*this`)	
`string& operator=(const string& str);`	Zuweisung des Strings `str`
`string& operator=(const char* s);`	Zuweisung des C-Strings `s`
`string& operator=(char c);`	Zuweisung des Einzelzeichens `c` (neue Stringlänge `= 1`)
`string& assign(const string& str);`	Zuweisung des Strings `str`
`string& assign(const string& str` **`size_type pos, size_type n);`**	Falls `pos<str.size()` : Zuweisung von `n` (höchstens aber allen) Zeichen ab der Position `pos` aus dem String `str` Falls `pos==str.size()` : keine Wirkung Falls `pos>str.size()` : Werfen der Exception **`out_of_range`**
`string& assign(const char* s);`	Zuweisung des C-Strings `s`
`string& assign(const char* s, size_type n);`	Zuweisung der ersten `n` (höchstens aber alle) Zeichen des C-Strings `s`
`string& assign(size_type n, char c);`	Zuweisung eines Strings der Länge `n,` bei dem alle Zeichen gleich dem Zeichen `c` sind

ANSI/ISO-C++-Standardbibliothek : Klasse `string` (3)

- **Memberfunktionen zum Zugriff zu Einzel-Zeichen im String (String-Elementen)**

`const char& operator[](size_type pos) const;`
 	
`char&  operator[](size_type pos);`
	Falls `pos<size()` : Rückgabe des Zeichens (bzw Referenz auf das Zeichen) an der Position (Index) `pos`
	Falls `pos>=size()` : undefiniert

`const char& at(size_type pos) const;`
 	
`char&  at(size_type pos);`
	Falls `pos<size()` : Rückgabe des Zeichens (bzw Referenz auf das Zeichen) an der Position (Index) `pos`
	Falls `pos>=size()` : Werfen der Exception **`out_of_range`**

- **Memberfunktionen zur Konvertierung von Strings in `char`-Arrays bzw C-Strings**

`size_type copy(char* s, size_type n,` `  size_type pos = 0) const;`	Kopieren von `n` Zeichen ab der Position `pos` (höchstens jedoch bis zum Stringende) in das durch `s` referierte `char`-Array. Es wird kein `'\0'`-Character angehängt Rückgabewert : Anzahl kopierter Zeichen
`const char* data() const;`	Rückgabe eines Pointers auf ein `char`-Array, dessen Elemente mit den Zeichen des aktuellen Strings übereinstimmen. Das `char`-Array ist nicht mit dem `'\0'`-Character abgeschlossen
`const char* c_str() const;`	Rückgabe eines Pointers auf einen C-String (Abschluss mit `'\0'`-Character), dessen Zeichen mit den Zeichen des aktuellen Strings übereinstimmen.

- **Memberfunktionen zum Vergleich von Strings bzw. Teilstrings**

Rückgabewert für **alle Vergleichsfunktionen** : `<0`, wenn akt. (Teil-) String `<` `str` (bzw `s`) `0`, wenn akt. (Teil-) String `==` `str` (bzw `s`) `>0`, wenn akt. (Teil-) String `>` `str` (bzw `s`)	
`int compare(const string& str) const;`	Vergleich aktueller String mit dem String `str`
`int compare(size_type pos, size_type n,` `  const string& str) const;`	Vergleich `n` Zeichen des aktuellen Strings ab Position `pos` mit dem String `str`

`int compare(size_type pos1, size_type n1,`
`  const string& str,`
`  size_type pos2, size_type n2) const;`
	Vergleich `n1` Zeichen des aktuellen Strings ab Position `pos1` mit `n2` Zeichen des Strings `str` ab Position `pos2`

`int compare(const char* s) const;`	Vergleich aktueller String mit dem C-String `s`

`int compare(size_type pos1, size_type n1,`
`  const char* s,`
`  size_type n2=npos) const;`
	Vergleich `n1` Zeichen des aktuellen Strings ab Position `pos1` mit `n2` Zeichen (default : bis String-Ende) des C-Strings `s`

ANSI/ISO-C++-Standardbibliothek : Klasse `string` (4)

- **Memberfunktionen zum Anhängen (Konkatenation)**

Für **alle Funktionen zum Anhängen** gilt : **Rückgabewert** ist **Referenz** auf das **aktuelle** String-Objekt (`*this`) Falls die neue Länge des Strings den maximal möglichen Wert für die Stringlänge übersteigen würde, wird die Exception **`length_error`** geworfen	
`string& operator+=(const string& str);`	Anhängen des Strings `str` an den aktuellen String
`string& operator+=(const char* s);`	Anhängen des C-Strings `s` an den aktuellen String
`string& operator+=(char c);`	Anhängen des Zeichens `c` an den aktuellen String
`string& append(const string& str);`	Anhängen des Strings `str` an den aktuellen String
`string& append(const string& str,` **`size_type n,`** **`size_type pos);`**	Falls `pos<=str.size()` : Anhängen von `n` Zeichen des Strings `str` ab der Position `pos` (aber maximal bis zum Stringende) an den aktuellen String. Falls `pos>str.size()` : Werfen der Exception **`out_of_range`**
`string& append(const char* s);`	Anhängen des C-Strings `s` an den aktuellen String
`string& append(const char* s,` **`size_type n);`**	Anhängen der ersten `n` (höchstens aber allen) Zeichen des C-Strings `s` an den aktuellen String
`string& append(size_type n, char c);`	Anhängen von `n`-mal das Zeichen `c` an den aktuellen String

- **Memberfunktionen zum Einfügen**

Für **alle Einfügefunktionen** gilt : **Rückgabewert** ist **Referenz** auf das **aktuelle** String-Objekt (`*this`) Falls eine Positionsangabe größer als die jeweilige Stringlänge ist, wird die Exception **`out_of_range`** geworfen Falls die neue Länge des Strings den maximal möglichen Wert für die Stringlänge (`max_size()`) übersteigen würde, wird die Exception **`length_error`** geworfen	
`string& insert(size_type pos,` **`const string& str);`**	Einfügen des Strings `str` in den aktuellen String ab der Position `pos`
`string& insert(size_type pos1,` **`const string& str,`** **`size_type pos2, size_type n);`**	Einfügen von maximal `n` Zeichen ab der Position `pos2` aus dem String `str` in den aktuellen String ab der Position `pos1`
`string& insert(size_type pos, const char* s);`	Einfügen des C-Strings `s` in den aktuellen String ab der Position `pos`
`string& insert(size_type pos, const char* s,` **`size_type n);`**	Einfügen der ersten `n` (maximal allen) Zeichen des C-Strings `s` in den aktuellen String ab Pos. `pos`
`string& insert(size_type pos,` **`size_type n, char c);`**	Einfügen `von n`-mal das Zeichen `c` in den aktuellen String ab der Position `pos`

ANSI/ISO-C++-Standardbibliothek : Klasse `string` (5)

- **Memberfunktionen zum Ersetzen und Löschen**

`void swap(string& str);`	Vertauschen des Inhalts des aktuellen Strings mit dem Inhalt des Strings `str`
Für **alle** folgenden **Funktionen** gilt : **Rückgabewert** ist **Referenz** auf das **aktuelle** String-Objekt (`*this`) Falls eine Positionsangabe größer als die jeweilige Stringlänge ist, wird die Exception **`out_of_range`** geworfen Für die **Ersetzungs**-Funktionen gilt : Die Anzahl der ersetzenden Zeichen kann kleiner, größer oder gleich der Anzahl der ersetzten Zeichen sein Falls die neue Länge des Strings den maximal möglichen Wert für die Stringlänge (`max_size()`) übersteigen würde, wird die Exception **`length_error`** geworfen	
`string& replace(size_type pos, size_type n, const string& str);`	Ersetzen von max. `n` Zeichen des aktuellen Strings ab der Position `pos` durch alle Zeichen des Strings `str`
`string& replace(size_type pos1,size_type n1, const string& str, size_type pos2, size_type n2);`	Ersetzen von max. `n1` Zeichen des aktuellen Strings ab der Position `pos1` durch max. `n2` Zeichen des Strings `str` ab der Position `pos2`
`string& replace(size_type pos, size_type n, const char* s);`	Ersetzen von max. `n` Zeichen des aktuellen Strings ab der Position `pos` durch alle Zeichen des C-Strings `s`
`string& replace(size_type pos, size_type n1, const char* s, size_type n2);`	Ersetzen von max. `n1` Zeichen des aktuellen Strings ab der Position `pos` durch max. `n2` Zeichen des `C`-Strings `s`
`string& replace(size_type pos, size_type n1, size_type n2, char c);`	Ersetzen von max. `n1` Zeichen des aktuellen Strings ab der Position `pos` durch `n2`-mal das Zeichen `c`
`string& erase(size_type pos = 0, size_type n = npos);`	Löschen von max. `n` Zeichen des aktuellen Strings ab der Position `pos`. Für Default-Parameterwerte : Löschen aller Zeichen

- **Memberfunktion zur Ermittlung eines Teilstrings**

`string substr(size_type pos = 0, size_type n = npos);`	Bildung und Rückgabe eines neuen Strings (`string`-Objekt), der aus max. `n` Zeichen des aktuellen Strings ab der Position `pos` besteht. Für Default-Parameterwerte : Rückgabe einer Kopie des aktuellen Strings. Falls `pos>size()` ist, Werfen der Exception **`out_of_range`**

ANSI/ISO-C++-Standardbibliothek: Klasse `string` (6)

- **Memberfunktionen zum Suchen**
  - ◇ Es existieren insgesamt **24 Memberfunktionen** zum Suchen in Strings
  - ◇ Es kann nach **Teilstrings**, **C-Teilstrings**, **Einzelzeichen** oder **Zeichen aus einer Zeichenmenge** gesucht werden. Es kann auch nach dem Auftritt von Zeichen gesucht werden, die in einer **Zeichenmenge nicht enthalten** sind
  - ◇ Es ist eine Suche nach dem **ersten** oder dem **letzten Auftritt** des Suchparameters möglich
  - ◇ Alle Suchfunktionen sind `const`-Funktionen, d. h. sie ändern das aktuelle Objekt nicht
  - ◇ Alle Suchfunktionen geben eine Wert vom Typ `string::size_type` zurück.
    Bei **Such-Erfolg** handelt es sich um die **Position** (Index) des (Beginns des) gefundenen Auftritts.
    Bei **Misserfolg** (Suchparameter ist im aktuellen Objekt nicht vorhanden) wird **`npos`** als Kenzeichnung einer **ungültigen Position** zurückgegeben.
  - ◇ Es existieren die folgenden Funktionsgruppen :

▻ **`size_type find(...) const`**	Suchen nach **erstem** Auftritt
▻ **`size_type rfind(...) const`**	Suchen nach **letztem** Auftritt
▻ **`size_type find_first_of(...) const`**	Suchen nach **erstem** Auftritt eines Zeichens aus einer **Zeichenmenge**
▻ **`size_type find_last_of(...) const`**	Suchen nach **letztem** Auftritt eines Zeichens aus einer **Zeichenmenge**
▻ **`size_type find_first_not_of(...) const`**	Suchen nach **erstem** Auftritt eines Zeichens, das in einer **Zeichenmenge nicht** enthalten ist
▻ **`size_type find_last_not_of(...) const`**	Suchen nach **letztem** Auftritt eines Zeichens, das in einer **Zeichenmenge nicht** enthalten ist

  - ◇ **Einige ausgewählte Suchfunktionen** :

`size_type find(const string& str, size_type pos = 0) const;`	Suchen nach erstem Auftritt des Strings `str` ab der Position `pos` im aktuellen String
`size_type find(const char* s, size_type pos = 0) const;`	Suchen nach erstem Auftritt des C-Strings `s` ab der Position `pos` im aktuellen String
`size_type find(char c, size_type pos = 0) const;`	Suchen nach erstem Auftritt des Zeichens `c` ab der Position `pos` im aktuellen String
`size_type find(const char* s, size_type pos, size_type n) const;`	Suchen nach erstem Auftritt der ersten `n` Zeichen des C-Strings `s` ab der Position `pos` im aktuellen String
`size_type rfind(const string& str, size_type pos = npos) const;`	Suchen nach letztem Auftritt des Strings `str` ab der Position `pos` im aktuellen String
`size_type find_first_of(const string& str, size_type pos = 0) const;`	Suchen nach erstem Auftritt eines der im String `str` enthaltenen Zeichen ab der Position `pos` im aktuellen String
`size_type find_last_not_of(const string& str, size_type pos = npos) const;`	Suchen nach letztem Auftritt eines der im String `str` nicht enthaltenen Zeichen ab der Position `pos` im aktuellen String

ANSI/ISO-C++-Standardbibliothek : Freie Stringbearbeitungsfunktionen (1)

- **Freie Operatorfunktionen zur String-Konkatenation**

Für **alle Funktionen** gilt : Sie erzeugen einen **neuen String** (`string`-Objekt), den sie als **Funktionswert** zurückgeben In ihrem Verhalten werden sie auf den Aufruf der Memberfunktion `append()` zurückgeführt Das bedeutet, dass die Exception **`length_error`** geworfen wird, falls die Länge des neu zu erzeugenden Strings den maximal möglichen Wert für die Stringlänge übersteigt.	
`string operator+(const string& str1, const string& str2);`	Erzeugung eines neuen Strings, der aus der Konkatenation des Strings `str1` mit dem String `str2` besteht
`string operator+(const char* s1, const string& str2);`	Erzeugung eines neuen Strings, der aus der Konkatenation des umgewandelten C-Strings `s1` mit dem String `str2` besteht
`string operator+(const string& str1, const char* s2);`	Erzeugung eines neuen Strings, der aus der Konkatenation des Strings `str1` mit dem umgewandelten C-String `s2` besteht
`string operator+(char c1, const string& str2);`	Erzeugung eines neuen Strings, der aus der Konkatenation des umgewandelten Zeichens `c1` mit dem String `str2` besteht
`string operator+(const string& str1, char c2);`	Erzeugung eines neuen Strings, der aus der Konkatenation des Strings `str1` mit dem umgewandelten Zeichen `c2` besteht

- **Freie Operatorfunktionen zum String-Vergleich**

Für die **folgenden Funktionen** gilt : Das Symbol ***op*** steht für einen der Vergleichsoperatoren **`== != < <= > >=`** **Rückgabewert** : `true`, falls der Vergleich erfüllt ist, `false`, falls der Vergleich nicht erfüllt ist	
`bool operator op(const string& str1, const string& str2);`	Vergleich des Strings `str1` mit dem String `str2`
`bool operator op(const char* s1, const string& str2);`	Vergleich des C-Strings `s1` mit dem String `str2`
`bool operator op(const string& str1, const char* s2);`	Vergleich des Strings `str1` mit dem C-String `s2`

- **Freie Funktion zum Vertauschen zweier Strings**

`void swap(string& str1, string& str2);`	Vertauschen des Inhalts Strings `str1` mit dem Inhalt des Strings `str2` Wirkungsgleich mit : **`str1.swap(str2);`**

ANSI/ISO-C++-Standardbibliothek : Freie Stringbearbeitungsfunktionen (2)

- **Freie Operatorfunktionen zur Stream-Ein-/Ausgabe**

`istream& operator>>(istream& is, string& str);`	Einlesen der nächsten Zeichen aus dem durch `is` referierten Eingabestream und Ablage dieser im String `str`. Falls eine Feldbreite mit `is.width(...)` explizit gesetzt worden ist, werden maximal `is.width()` Zeichen eingelesen, andernfalls werden maximal `str.max_size()` Zeichen eingelesen. Das Einlesen wird früher beendet, wenn das nächste einzulesende Zeichen ein *White-Space-Character* ist oder das Dateiende erreicht ist Rückgabewert : `is`
`ostream& operator<<(ostream& os, string& str);`	Ausgabe des Strings `str` in den durch `os` referierten Ausgabestream Rückgabewert : `os`

- **Freie Funktionen zum Einlesen von Textzeilen aus Eingabe-Streams**

`istream& getline(istream& is, string& str, char delim);`	Einlesen einer mit dem Zeichen `delim` abgeschlossenen Zeichenfolge aus dem durch `is` referierten Eingabestream und Ablage dieser im String `str`. Das Zeichen `delim` wird aus dem Stream entfernt aber nicht mit im String abgelegt. Das Einlesen wird vor Auftritt des Zeichens `delim` beendet, wenn bereits `str.max_size()` Zeichen gelesen wurden oder das Dateiende erreicht ist
`istream& getline(istream& is, string& str);`	Einlesen einer mit dem Zeichen `'\n'` abgeschlossenen Zeichenfolge aus dem durch `is` referierten Eingabestream und Ablage dieser im String `str`. Das Zeichen `'\n'` wird aus dem Stream entfernt aber nicht mit im String abgelegt. Das Einlesen wird vor Auftritt des Zeichens `'\n'` beendet, wenn bereits `str.max_size()` Zeichen gelesen wurden oder das Dateiende erreicht ist Der Aufruf dieser Funktion entspricht dem Aufruf von `getline(is, str, '\n');`

Demo-Programm 1 zur ANSI/ISO-C++-Standardbibliotheks-Klasse `string` (1)

- **C++-Quelldatei `stringdem_1.cpp` (`main()`-Funktion des Programms `stringdem1`)**

```
// C++-Quelldatei stringdem1_m.cpp
// Demonstrationsprogramm stringdem1 zur Bibliotheks-Klasse string

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

void stringInfo(ostream& out, const string& s)
{ out << s;
  out << "\nsize     : " << s.size();
  out << "\ncapacity : " << s.capacity();
  out << "\nmax_size : " << s.max_size();
}

int main()
{
  cout << "\nWert von npos : " << hex << string::npos << dec << endl;

  string s1;         // Leer-String
  cout << "\ns1 Leerstring";
  stringInfo(cout, s1);
  cout << endl;

  char* cp="Hallo !";
  string s2(cp);
  cout << "\ns2 aus C-String : ";
  stringInfo(cout, s2);
  cout << endl;

  string s3(s2);
  cout << "\ns3 mit Copy-Konstruktor : ";
  stringInfo(cout, s3);
  cout << endl;

  string s4(3, 'Z');
  cout << "\ns4 aus Einzelzeichen : ";
  stringInfo(cout, s4);
  cout << endl;

  string::size_type nLen=3;
  s2.resize(nLen);
  cout << "\ns2 nach resize(" << nLen << ") : ";
  stringInfo(cout, s2);
  cout << endl;

  nLen=10;
  s2.resize(nLen, 'o');
  cout << "\ns2 nach resize(" << nLen << ") mit Fuellzeichen : ";
  stringInfo(cout, s2);
  cout << endl;

  nLen=80;
  s2.reserve(nLen);
  cout << "\ns2 nach reserve(" << nLen << ") : ";
  stringInfo(cout, s2);
  cout << endl;
  return 0;
}
```

Demo-Programm 1 zur ANSI/ISO-C++-Standardbibliotheks-Klasse `string` (2)

- **Ausgabe des Programms `stringdem1`**

```
Wert von npos : ffffffff

s1 Leerstring
size     : 0
capacity : 0
max_size : 4294967293

s2 aus C-String : Hallo !
size     : 7
capacity : 31
max_size : 4294967293

s3 mit Copy-Konstruktor : Hallo !
size     : 7
capacity : 31
max_size : 4294967293

s4 aus Einzelzeichen : ZZZ
size     : 3
capacity : 31
max_size : 4294967293

s2 nach resize(3) : Hal
size     : 3
capacity : 31
max_size : 4294967293

s2 nach resize(10) mit Fuellzeichen : Haloooooo
size     : 10
capacity : 31
max_size : 4294967293

s2 nach reserve(80) : Haloooooo
size     : 10
capacity : 95
max_size : 4294967293
```

Demo-Programm 2 zur ANSI/ISO-C++-Standardbibliotheks-Klasse `string`

- **C++-Quelldatei `stringdem2_m.cpp` (`main()`-Funktion des Programms `stringdem2`)**

```
// C++-Quelldatei stringdem2_m.cpp
// Demonstrationsprogramm stringdem2 zur Bibliotheks-Klasse string

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

int main()
{ string s1;
  string s2("Blaer bricht das Voelkerrecht");
  string s3, s4;
  try
  { s1="Busch";
    s4=s1;
    s1.append(" Monkey");
    cout << endl << s1.c_str() << endl;    // s1.c_str() statt s1 nur zu Demo-Zweck
    s3.assign(s2, 5, string::npos);
    cout << endl << s3 << endl;
    s4+=s3;
    cout << endl << s4 << endl;
    cout << endl << endl;
    for (unsigned i=6; i<20; i++)
      cout << hex << setw(2) << setfill('0') << (s1[i]&0xff) << ' ';
    cout << dec << setfill(' ') << endl;
    for (i=6; i<20; i++)
      cout << hex << setw(2) << setfill('0') << (s1.at(i)&0xff) << ' ';
    cout << dec << setfill(' ') << endl;
  }
  catch (const exception& e)
  { cerr << "\nexception : " << e.what() << endl;
  }
  try
  { cout << "\nreserve(" << hex << string::npos << dec << ") : ";
    s4.reserve(string::npos);
  }
  catch (const exception& e)
  { cerr << "\nexception : " << e.what() << endl;
  }
```

- **Ausgabe des Programms `stringdem2`**

```
Busch Monkey

 bricht das Voelkerrecht

Busch bricht das Voelkerrecht


4d 6f 6e 6b 65 79 00 00 00 00 00 00 00 00
4d 6f 6e 6b 65 79
exception : invalid string position

reserve(ffffffff) :
```

Demo-Programm 3 zur ANSI/ISO-C++-Standardbibliotheks-Klasse `string` (1)

- **Quelldatei `stringdem_3.cpp` (`main()`-Funktion des Programms `stringdem3`)**

```
// C++-Quelldatei stringdem3_m.cpp
// Demonstrationsprogramm stringdem3 zur Bibliotheks-Klasse string
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char** argv)
{
  int iRet=0;
  string dname;
  if (argc>=2)
    dname=argv[1];
  else
  { cout << "\nName der Textdatei ? ";
    cin >> dname;
  }

  ifstream idat(dname.c_str());
  if (!idat)
  {
    cout << "\nDatei \"" << dname << "\" kann nicht geoeffnet werden !\n";
    iRet=1;
  }
  else
  { string text;
    string zeile;
    string grzeil;

    while (getline(idat, zeile))
    {
      if (zeile>grzeil)
        grzeil=zeile;
      text+=zeile;
      text+='\n';
    }
    cout << "\nInhalt der eingelesenen Textdatei :\n";
    cout << endl << text << endl;
    cout << "Gesamte Textlaenge : " << text.length() << endl;
    cout << "\n\"Groesste\" Zeile :\n";
    cout << grzeil << endl;

    string wort;
    cout << "\nSuchen nach ? ";
    cin >> wort;
    string::size_type pos=text.find(wort);
    // Haeufig trifft man auch folgendes an: //   size_t pos=text.find(wort);
    while(pos!=string::npos)
    {
      cout << "enthalten ab Pos. : " << pos << endl;
      text.insert(pos, 1, '*');
      pos+=1+wort.length();
      text.insert(pos++, 1, '*');
      pos=text.find(wort, pos+1);
    }
    cout << "\nMarkierte Textdatei :\n";
    cout << endl << text << endl;
  }
  return iRet;
}
```

- **Ausgabe des Programms `stringdem3` (Beispiel-Aufruf : `stringdem3 hausverbot.txt`)**

```
Inhalt der eingelesenen Textdatei :

Hausverbot fuer Bush und Blair in Geburtskirche
US-Praesident George Bush und der britische Premierminister Tony Blair haben
lebenslanges Hausverbot fuer die Geburtskirche in Bethlehem erhalten.
Dies betonte der palaestinensische Archimandrit Attalah Hanna vom
griechisch-orthodoxen Patriarchat in Jerusalem bei "Islam-online".
Auch US-Verteidigungsminister Donald Rumsfeld und der britische
Aussenminister Jack Straw duerften das Gotteshaus niemals mehr betreten.
Gleichzeitig sprach der Archimandrit, ranghoechstes Mitglied des
griechisch-orthodoxen Patriarchats, auch von einer "Exkommunikation".
Bush und Blair haetten sich selbst aus der Kirchengemeinschaft
ausgeschlossen, weil sie die Warnungen aller christlichen Kirchen vor
einem Irak-Krieg missachtet haetten, sagte der Geistliche.
2. April 2003 (SZ)

Gesamte Textlaenge : 817

"Groesste" Zeile :
lebenslanges Hausverbot fuer die Geburtskirche in Bethlehem erhalten.

Suchen nach ? und
enthalten ab Pos. : 21
enthalten ab Pos. : 76
enthalten ab Pos. : 379
enthalten ab Pos. : 615

Markierte Textdatei :

Hausverbot fuer Bush *und* Blair in Geburtskirche
US-Praesident George Bush *und* der britische Premierminister Tony Blair haben
lebenslanges Hausverbot fuer die Geburtskirche in Bethlehem erhalten.
Dies betonte der palaestinensische Archimandrit Attalah Hanna vom
griechisch-orthodoxen Patriarchat in Jerusalem bei "Islam-online".
Auch US-Verteidigungsminister Donald Rumsfeld *und* der britische
Aussenminister Jack Straw duerften das Gotteshaus niemals mehr betreten.
Gleichzeitig sprach der Archimandrit, ranghoechstes Mitglied des
griechisch-orthodoxen Patriarchats, auch von einer "Exkommunikation".
Bush *und* Blair haetten sich selbst aus der Kirchengemeinschaft
ausgeschlossen, weil sie die Warnungen aller christlichen Kirchen vor
einem Irak-Krieg missachtet haetten, sagte der Geistliche.
2. April 2003 (SZ)
```

10.5 Iteratoren in C++

- **Eigenschaften** :
  - ◇ Ein **Iterator** ist ein **Hilfsobjekt**, das einen **Mechanismus** zum **Durchlaufen von Objekten**, die Ansammlungen von Daten (i.a. andere Objekte) enthalten und verwalten (Container-Objekte), zur Verfügung stellt.
  - ◇ Im Wesentlichen implementieren Iteratoren die folgenden **fundamentalen Funktionalitäten**:
    - ▻ **Zugriff** zu dem gerade **aktuellen Element** (durch aktuelle Position festgelegt)
    - ▻ **Fortschreiten** zum **nächsten Element** (Setzen der aktuellen Position auf das nächste Element)
    - ▻ Erkennen des Erreichens des **letzten Elements**
  - ◇ Diese Funktionalitäten implizieren, dass die einzelnen Elemente in einer **Reihenfolge** angeordnet sind.
    Damit unterstützen Iteratoren ein **abstraktes Datenmodell für Container**, bei dem deren Inhalt als **Sequenz von Elementen** (Objekten) aufgefasst wird.
    Eine **Sequenz** kann definiert werden, als eine **Abstraktion** von **"etwas"**, das man mit der Operation **"nächstes_Element"** von Anfang bis Ende durchlaufen kann.
  - ◇ Prinzipiell kann der Mechanismus zum Navigieren in Container-Objekten auch in diesen selbst angesiedelt sein.
    Dann existiert pro Container-Objekt aber immer nur eine aktuelle Position. In vielen Anwendungen ist es aber notwendig zu mehreren Elementen (und damit zu verschiedenen aktuellen Positionen) gleichzeitig zuzugreifen (z.B. Durchlaufen einer Menge in zwei geschachtelten Schleifen zum Vergleich eines Elements mit allen anderen Elementen der **Menge**). Dies lässt sich mittels **mehrerer Iterator-Objekte** für das **gleiche Container-Objekt** relativ leicht realisieren.
    → **Auslagerung** des Navigations- und Zugriffsmechanismus in eine **eigene Iterator-Klasse** ist **wesentlich flexibler.**
  - ◇ In einer **allgemeineren Betrachtungsweise** lassen sich **Iteratoren** als **abstrahierende Verallgemeinerung** von **Pointern** (die jeweils auf ein **Element** eines **Arrays zeigen**), auffassen:
    So wie man mittels eines Pointers ein Array durchlaufen und zu den einzelnen Elementen schreibend und lesend zugreifen kann, kann man mit einem Iterator einen Container durchlaufen und zu den einzelnen in ihm gespeicherten Elementen zugreifen.
    → Ein **Iterator** kann als eine **Abstraktion** eines **Zeigers** auf ein **Element** einer **Sequenz** betrachtet werden.
  - ◇ **Sequenzen** können **unterschiedlich** strukturiert und organisiert sein. → **unterschiedliche Container-Arten**.
    Zu jeder Container-Art gehört sinnvollerweise eine eigene **passende Iterator-Art**.
    In einer konsistenten Implementierung (z.B. in einer Bibliothek) lassen sich Iteratoren – unabhängig von ihrer jeweiligen Art – aber immer **gleichartig verwenden**.
    Sie stellen damit ein **einheitliches Konzept** zum Positionieren und Durchlaufen von Container-Objekten zur Verfügung.
- **Implementierung** (1) :
  - ◇ Zur Ermittlung der jeweiligen aktuellen Position und zum Zugriff zu dem dadurch festgelegten Element benötigt ein Iterator-Objekt **Zugang** zu den **Interna** des **Container-Objekts**.
    Dieser kann realisiert werden
    - ▻ durch eine Deklaration der **Iteratorklasse** als **Freundklasse** der **Containerklasse** oder
    - ▻ durch die Bereitstellung einer geeigneten **öffentlichen Zugriffsschnittstelle** durch die **Containerklasse**
  - ◇ Häufig wird eine Iteratorklasse als **innere Klasse** der von ihr bearbeiteten Containerklasse definiert.
    Dadurch wird die Zugehörigkeit der Iteratorklasse zur Containerklasse betont.
  - ◇ Ein Iterator-Objekt muß wenigsten **zwei Datenkomponenten** besitzen:
    - ▻ Pointer oder Referenz auf das **Container-Objekt**
    - ▻ **aktuelle Position**
  - ◇ Im einfachsten Fall ist ein Iterator-Objekt ein **Funktionsobjekt**, d. h. als wesentliche Memberfunktion ist der Funktionsaufruf-Operator überladen.

Bei der Erzeugung eines Iterator-Objekts wird die aktuelle Position auf den Anfang (erstes Element) gesetzt. Jede Verwendung des Objekts in Form eines Funktionsaufrufs liefert einen Pointer auf das aktuelle Element und setzt die aktuelle Position auf das nächste Element. Liegt die aktuelle Position nach dem letzten Element wird der `NULL`-Pointer zurückgeliefert (Ende der Element-Sequenz ist erreicht!)

- **Beispiel : Realisierung einer einfachen Iteratorklasse zu einem assoziativen Array** (oop\bsp_cpp\iteratoren)

```
#define MAXLEN  10
class Assar                         // Assoziatives Array,
{ public :                          // Implementierung s.Kapitel Operatorüberladung
    Assar(int =MAXLEN);             // Konstruktor
    int& operator[](const char*);   // Indizierungs-Operator
    friend class AssarIter;         // Iteratorklasse
  private :
    struct paar { char *wort; int zahl; } *vec;
    int max;
    int free;
    Assar(const Assar&);            // verhindert Kopieren bei Initialis.
    Assar& operator=(const Assar&); // verhindert Kopieren bei Zuweisung
};
// ----------------------------------------------------------------------------
class AssarIter                      // Iteratorklasse
{ public :
    AssarIter(const Assar&);
    const char* operator()(void);
  private :
    const Assar* m_pclFeld;         // Array, auf dem Iterator arbeitet
    int m_iAct;                     // aktueller Array-Index
    AssarIter(const AssarIter&);            // verhindert Kopieren bei Initialis.
    AssarIter& operator=(const AssarIter&); // verhindert Kopieren bei Zuweisung
};

// ----------------------------------------------------------------------------
#include <cstdlib>
AssarIter::AssarIter(const Assar& fld) : m_pclFeld(&fld), m_iAct(0) {}

const  char* AssarIter::operator()()
{ char* hp;
  if (m_iAct<m_pclFeld->free)
  { hp=m_pclFeld->vec[m_iAct].wort;
    m_iAct++;
  }
  else
  { hp=NULL;
    m_iAct=0;
  }
  return hp;
}
// ----------------------------------------------------------------------------
#include <iostream>
using namespace std;

void einlesen(Assar& feld)
{ const int MAXL=256;
  char buffer[MAXL];
  while (cin >> buffer) feld[buffer]++;        // Indizierung über String
}

int main()
{ Assar feld;
  einlesen(feld);
  AssarIter next(feld);                        // Iteratorobjekt
  const char* p;
  while(p=next())                              //ruft Funktionsoperator auf
    cout << p << " : " << feld[p] << '\n';
  return 0;
}
```

Iteratoren in C++ (3)

- **Implementierung** (2) :

  ◇ Allgemeiner und flexibler einsetzbare Iteratoren **trennen**
  - den **Zugriff** zum jeweils **aktuellen Element**
  - vom **Fortschalten** der **aktuellen Position** auf das nächste Element

  durch die Bereitstellung getrennter geeigneter Memberfunktionen.
  Zusätzlich stellen sie häufig auch eine Memberfunktion zum Vergleich zweier Iteratoren bzw. der von ihnen aktuell referierten Elemente zur Verfügung.

  ◇ Noch universeller und eleganter lassen sich Iteratoren einsetzen, die die **Analogie zum Pointer implementieren**.
  Hierfür verfügen sie über **geeignete Operatorfunktionen**, i.a. wenigstens über :
  - ▻ `operator++()`,
  - ▻ `operator--()` (falls auch Rückwärts-Iteration möglich sein soll),
  - ▻ `operator*()`,
  - ▻ `operator==()`.

  Direktzugriffs-Iteratoren überladen darüber hinaus auch
  - ▻ den Additionsoperator
  - ▻ und den Subtraktionsoperator,
  - ▻ gegebenenfalls auch den Indexoperator
  - ▻ sowie die übrigen Vergleichsoperatoren.

  Derartige Iteratoren lassen sich wie Pointer, gegebenenfalls einschließlich "Pointer"-Arithmetik, verwenden.

  ◇ **Containerklassen** sind häufig als **Klassen-Templates** definiert.
  Der Typ (die Klasse) der im Container abgelegten Elemente ist in diesem Fall Template-Parameter.
  Eine für die Container-Klasse zuständige **Iteratorklasse** ist dann ebenfalls von dem **Template-Parameter abhängig**.
  Das bedeutet, dass sie bei einer **Definition außerhalb** der **Containerklasse ebenfalls als Klassen-Template** definiert werden muß.
  Erfolgt ihre Definition dagegen als **innere Klasse** der **Containerklasse** ist **kein Klassen-Template erforderlich**.
  Die Iteratorklasse kann alle Namen – also auch die formalen Typ-Parameter-Namen – des umschließenden Container-klassen-Templates direkt verwenden.

  ◇ Bei einer **pointer-analogen Implementierung** einer Iteratorklasse, existieren in der **zugehörigen Containerklasse** häufig **zwei Memberfunktionen**, die jeweils ein **Iterator-Objekt** als Funktionswert **erzeugen** :
  - ▻ Die eine dieser Funktionen (häufig **`begin()`** genannt) liefert ein Iteratorobjekt, das auf das **erste** enthaltene **Element** zeigt.
  - ▻ Die andere Funktion (häufig **`end()`** genannt) liefert ein Iteratorobjekt, das auf ein Element zeigt, das **formal unmittelbar hinter dem letzten** enthaltenen **Element** angeordnet ist, tatsächlich aber gar nicht existiert oder ein Dummy-Element ist. Es dient damit zur Kennzeichnung des **Sequenzendes**.

- **Anmerkung zur ANSI-C++-Standardbibliothek**

  ◇ Die **Standard Template Library (STL)**, ein wesentlicher Bestandteil der Standardbibliothek, definiert mehrere **Iteratorarten** (Iteratorkategorien) und **Iteratorklassen** für verschiedene Container-Klassen sowie zur Anwendung auf Stream- und Streambuffer-Klassen

Iteratoren in C++ (4 - 1)

- **Beispiel : Realisierung einer Iteratorklasse mit pointer-analoger Implementierung**
  - ◇ Iteratorklasse zu einem Klassen-Template für Listen. Der Typ der Listenelemente ist Template-Parameter.
  - ◇ Iteratorklasse ist als innere Klasse des Listen-Klassen-Templates definiert
  - ◇ **Definition des Listen-Klassen-Templates `List<T>` sowie der Iteratorklasse `ListIter`** (enthalten in Headerdatei `list.h`)

→ Demonstrationsprogramm **`listiter`**

```
//list.h
template <class T>
class List
{
  private :
    struct ListEl { ListEl* next; ListEl* prev; T data; };
  public :
    List();
    ~List();
    void insert(const T&);

    // ---------------------------------------------------------------------
    class ListIter                    // eingebettete Iteratorklasse
    {
      public :
        ListIter(List& lst)           { pLst=&lst; pAct=pLst->first; }
        T& operator*()                { return pAct->data; }
        ListIter& operator++()        { pAct=pAct->next; return *this; }
        bool operator==(ListIter& it) { return pAct==it.pAct; }
        friend class List<T>;
      private :
        ListIter& setPos(ListEl* pa)  { pAct=pa; return *this; }
        List*    pLst;
        ListEl*  pAct;
    };
  // -----------------------------------------------------------------------
    friend class ListIter;
    ListIter begin() { return ListIter(*this); }
    ListIter end()   { return ListIter(*this).setPos(last); }
  private :
    ListEl* first;
    ListEl* last;
};

template <class T>                // Konstruktor von List<T>
List<T>::List()
{ ListEl* dummy=new ListEl;       // formales Dummy-Element
  dummy->next=dummy->prev=NULL;   // gehört nicht zur eigentlichen Liste
  first=last=dummy;               // dient zur Kennzeichnung des Listenendes
}                                 // Bei Templates ist iterator-Ende hinter dem
                                  // letzten Nutzelement, somit dumm->prev
                                  //  möglich, was bei Null-Pointer als Endekenn-
                                  //  nicht möglich wäre
template <class T>
List<T>::~List()
{ while(first!=NULL){ /* Zerstörung aller Listenelemente */
      ListEl* help= first->next;
      delete first; first=help;
   }
   first=last=NULL;
}
```

Iteratoren in C++ (4 - 2)

```
//Fortsetzung
template <class T>
void List<T>::insert (const T& dat)
{ /* Einfügen eines neuen Listenelementes am Ende (vor das Dummy-Element) */
   ListEl* neu=new ListEl;
   neu->data=dat;
   if(first==last) //Liste leer
   {
          first=neu;  //List-Anfang/Ende setzen
          last->prev =neu;
          neu->prev=NULL;
          neu->next =last;

   }else {    //Verkettung (hinten anhängen)
          ListEl* help = last->prev;
          last->prev= neu;
          help->next = neu;
          neu->prev =help;
          neu->next= last;
   }
}
#endif
```

◇ **Beispiel für Anwendercode** (enthalten in Datei `listiter_m.cpp`)

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

int main(void)
{ List<int> ilst;

  for (int i=0; i<10; i++)
    ilst.insert(i);

  List<int>::ListIter it1=ilst.begin();
  List<int>::ListIter ite=ilst.end();

  cout << "\nInhalt int-Liste :\n";
  while (!(it1==ite))
  {
    cout << *it1 << "   ";
    ++it1;
  }
  cout << endl;

  List<double> dlst;

  for (i=1; i<=5; i++)
    dlst.insert(i*1.95583);

  List<double>::ListIter dit1=dlst.begin();
  List<double>::ListIter dite=dlst.end();

  cout << "\nInhalt double-Liste:\n";
  while (!(dit1==dite))
  {
    cout << *dit1 << endl;
    ++dit1;
  }
  cout << endl;
```

```
Inhalt int-Liste :
0   1   2   3   4   5   6   7   8   9

Inhalt double-Liste :
1.95583
3.91166
5.86749
7.82332
9.77915
```

10.6 STL Funktionsobjekte

- **Allgemeines**
  - ◇ **Funktionsobjekte** sind Objekte, für die der Funktionsaufruf-Operator **`operator()`** definiert ist.
    (siehe Kapitel Operatorüberladung)
    Derartige Objekte lassen sich **wie Funktionen verwenden**.
    Gegenüber normalen Funktionen können sie einen inneren Zustand (Datenkomponenten) besitzen.
    Dadurch lassen sie sich flexibler und effizienter als diese verwenden.
  - ◇ In der **Utility-Bibliothek** sind zahlreiche Klassen für Funktionsobjekte, die diverse Standard-Operationen implementieren, definiert.
    Um eine von den Operanden-Typen der jeweiligen Operation unabhängige Formulierung zu ermöglichen, sind sie als **Klassen-Templates** (mittels `struct`) implementiert.
    In erster Linie sind sie für die Verwendung mit den **Algorithmen der STL** vorgesehen.
    Sie lassen sich aber auch unabhängig davon einsetzen.
  - ◇ U.a. stellt die Bibliothek für sämtliche in der Sprache enthaltenen **arithmetischen**, **Vergleichs-** und **logischen Operatoren** (Standard-Operationen) entsprechende Funktionsobjekt-Klassen zur Verfügung.
  - ◇ Ihre Definitionen (und Implementierungen) sind in der **Headerdatei `<functional>`** enthalten.

- **Basisklassen für Funktionsobjekte**
  - ◇ Es ist je eine Basisklasse für "**einstellige Funktionsobjekte**" und für "**zweistellige Funktionsobjekte**" definiert.
    Für "einstellige Funktionsobjekte" wird die Operatorfunktion `operator()` mit einem Parameter aufgerufen, für "zweistellige Funktionsobjekte" analog mit zwei Parametern.
  - ◇ Durch die Basisklassen werden im wesentlichen nur standardisierte **Namen** für die **Parameter-** und **Rückgabe-Typen** definiert.
    Diese Typnamen werden für die Definition der – ebenfalls in der Utility-Bibliothek implementierten – **Binder-Klassen** benötigt.
  - ◇ **Basisklasse für "einstellige Funktionsobjekte"**

```
template <class Arg, class Result>
  struct unary_function
  {
    typedef Arg    argument_type;
    typedef Result result_type;
  }
```

  - ◇ **Basisklasse für "zweistellige Funktionsobjekte"**

```
template <class Arg1, class Arg2, class Result>
  struct binary_function
  {
    typedef Arg1   first_argument_type;
    typedef Arg2   second_argument_type;
    typedef Result result_type;
  }
```

ANSI/ISO-C++-Standardbibliothek : Funktionsobjekte (2)

- **Funktionsobjekt-Klassen für arithmetische Operationen**

  ◇ **Prinzipielle Definition** der Klassen-Templates :

```
template <class T>
  struct plus : binary_function<T, T, T>  //class plus: public binary…
  {
    T operator()(const T& x, const T& y) const
    { return (x + y);
    }
  }
```

Anmerkung: Da es keine zu kapselnden Datenkomponenten gibt, kann man struct verwenden und sich 2x "public" sparen. binary_function<erster_parametertyp, zweiter_parametertyp, ergebnistyp> ist die Basisklasse für binäre Funktionsobjekt-Klassen (in der nur typedefs enthalten sind). Man leitet von binary_function ab, wenn man Funktionsadapter (siehe später) auf die Funktionsobjekt-Klasse anwenden will.

◇ **Überblick**

Funktionsobjekt-Klasse	Parameter von `operator()`	Ergebnis von `operator()`
plus<T>	(const T& **x**, const T& **y**)	**x + y**
minus<T>	(const T& **x**, const T& **y**)	**x – y**
multiplies<T>	(const T& **x**, const T& **y**)	**x * y**
divides<T>	(const T& **x**, const T& **y**)	**x / y**
modulus<T>	(const T& **x**, const T& **y**)	**x % y**
negate<T>	(const T& **x**)	**- x**

- **Funktionsobjekt-Klassen für Vergleichs- und logische Operationen**

  ◇ Die Operatorfunktion `operator()` dieser Klassen liefern einen Funktionswert vom Typ `bool`. Funktionsobjekte mit dieser Eigenschaft werden **Prädikate** genannt.

  ◇ **Prinzipielle Definition** der Klassen-Templates :

```
template <class T>
  struct equal_to : binary_function<T, T, bool>
  {
    bool operator()(const T& x, const T& y) const
    { return (x == y);
    }
  }
```

◇ **Überblick**

Funktionsobjekt-Klasse	Parameter von `operator()`	Ergebnis von `operator()`
equal_to<T>	(const T& **x**, const T& **y**)	**x == y**
not_equal_to<T>	(const T& **x**, const T& **y**)	**x != y**
greater<T>	(const T& **x**, const T& **y**)	**x > y**
less<T>	(const T& **x**, const T& **y**)	**x < y**
greater_equal<T>	(const T& **x**, const T& **y**)	**x >= y**
less_equal<T>	(const T& **x**, const T& **y**)	**x <= y**
logical_and<T>	(const T& **x**, const T& **y**)	**x && y**
logical_or<T>	(const T& **x**, const T& **y**)	**x || y**
logical_not<T>	(const T& **x**)	**! x**

ANSI/ISO-C++-Standardbibliothek : Funktionsobjekte (3)

- **Einfaches Demo-Programm**

```
// C++-Quelldatei bibfodem1_m.cpp
// Modul mit main()-Funktion fuer das Programm bibfuncobjdem1
// Einfaches Demo-Programm zu Funktionsobjekten der Standard-Bibliothek

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


void testfunc();

int main(void)
{
  plus<int> iadd;       // Definition des Funktionsobjekts iadd
  int i1, i2, i3, i4;
  i1=5;
  i2=6;
  i3=10;
  i4=iadd(i1, i2);      // Aufruf operator() für Objekt iadd

  bool gleich;
  gleich=equal_to<int>()(i4, i3); // Erzeugung eines namenlosen
                                  // temporären Funktionsobjektes
                                  // Aufruf operator() fuer dieses Objekt
  cout << boolalpha;
  cout << "\nUngleichheit von (" << i1 << "+" << i2 << ") und ";
  cout << i3 << " : " << logical_not<bool>()(gleich) << endl;
  testfunc();
  return 0;
}

template<class T, class Func>
T binop(const T& a, const T& b, Func f)
{ return f(a,b);}

double quadsum(double a, double b)
{ return (a*a + b*b); }

void testfunc()
{
  cout << "\nAddition   : "
       << binop(3.5, 2.7, plus<double>());     // Uebergabe Funktionsobjekt
  cout << "\nDivision   : "
       << binop(37, 7, divides<int>());        // Uebergabe Funktionsobjekt
  cout << "\nQuad-Summe : "
       << binop(3.2, 4.3, quadsum) << endl;    // Uebergabe Funktionspointer
```

Ausgabe des Programms :

```
Ungleichheit von (5+6) und 10 : true

Addition   : 6.2
Division   : 5
Quad-Summe : 28.73
```

ANSI/ISO-C++-Standardbibliothek : Funktionsobjekte (4)

- **Funktions-Adapter**

  ◇ In der Utility-Bibliothek sind – ebenfalls in der Headerdatei **`<functional>`** – auch so genannte **Funktions-Adapter** definiert.
  Dies sind Klassen (genauer: Klassen-Templates), die es auf einfache Art und Weise ermöglichen, neue Funktions-Objekte durch Modifikation anderer Funktionsobjekte bzw. durch Anpassung von Funktions-Pointern zu erzeugen.

  ◇ Es gibt vier **Gruppen von Funktions-Adaptern**:

  ▻ **Binder** (*binders*)
  Sie erzeugen aus einem zweistelligen Funktionsobjekt ein einstelliges Funktionsobjekt, indem ein Argument der Operatorfunktion `operator()` an einen bestimmten festen Wert gebunden wird.

  ▻ **Memberfunktionsadapter** (*adapters for pointers to members*)
  Sie ermöglichen es, dass eine Memberfunktion wie ein Funktionsobjekt als Argument eines Algorithmus verwendet werden kann.

  ▻ **Funktionszeigeradapter** (*adapters for pointers to functions*)
  Sie ermöglichen es, dass eine freie (oder statische Member-) Funktion wie ein Funktionsobjekt als Argument eines Algorithmus verwendet werden kann.

  ▻ **Negierer** (*negators*)
  Sie invertieren ein Prädikat.

  ◇ Funktions-Adapter sind auch miteinander **kombinierbar**.

  ◇ **Funktions-Adapter** sind als **Klassen-Templates** definiert.
  Template-Parameter ist die Klasse des umzuwandelnden Funktionsobjektes.
  Dem Konstruktor wird – gegebenenfalls zusammen mit weiteren Parametern – das umzuwandelnde Funktions-Objekt als Parameter übergeben. Er speichert dieses in einer Datenkomponente.
  Die Operator-Funktion `operator()` führt die modifizierte Operation unter Verwendung des gespeicherten Funktionsobjekts aus.
  Zu jedem Adapter gehört eine geeignete **Umwandlungsfunktion** (definiert als Funktions-Template), die aus einem als Argument übergebenen Funktionsobjekt ein Funktionsobjekt vom Adapter-Typ erzeugt.
  Die jeweilige Umwandlungsfunktion stellt die **Anwendungsschnittstelle** eines Funktions-Adapters dar.

- **Überblick über die Funktions-Adapter**

Funktionsdapter-Klasse	Umwandlungsfunktion	Gruppe
`binder2nd`	`bind2nd()`	Binder
`binder1st`	`bind1st()`	
`mem_fun_t`	`mem_fun()`	Memberfunktionsadapter
`const_mem_fun_t`	`mem_fun()`	
`mem_fun1_t`	`mem_fun()`	
`const_mem_fun1_t`	`mem_fun()`	
`mem_fun_ref_t`	`mem_fun_ref()`	
`const_mem_fun_ref_t`	`mem_fun_ref()`	
`mem_fun1_ref_t`	`mem_fun_ref()`	
`const_mem_fun1_ref_t`	`mem_fun_ref()`	
`pointer_to_unary_function`	`ptr_fun()`	Funktionszeigeradapter
`pointer_to_binary_function`	`ptr_fun()`	
`unary_negate`	`not1()`	Negierer
`binary_negate`	`not2()`	

Funktionsobjekte (5)

- **Beispiel für einen Binder : Umwandlungsfunktion `bind2nd` (Adapter `binder2nd`)**

  ◇ Erzeugt **aus** einem **zweistelligen Funktionsobjekt** und einem Wert `y` ein **einstelliges Funktionsobjekt**, in dem das zweite Argument (des zweistelligen Funktionsobjekts) an den Wert `y` gebunden wird

  ◇ **Definition des Klassen-Templates `binder2nd`**

```
template<class Operation>
  class binder2nd
    : public unary_function<typename Operation::first_argument_type,
                            typename Operation::result_type>
{ protected :
    Operation op;
    typename Operation::second_argument_type value;
  public :
    binder2nd(const Operation& binop,
              const typename Operation::second_argument_type& y)
      : op(binop), value(y) { }
    typename Operation::result_type  operator()
      (const typename Operation::first_argument_type& x) const
    { return op(x, value); }
};
```

  ◇ **Definition der Umwandlungsfunktion `bind2nd`**

```
template<class Operation, class T>
  binder2nd<Operation> bind2nd(const Operation& binop, const T& y)
  { return
      binder2nd<Operation>(binop, typename Operation::second_argument_type(y));
  }
```

  ◇ **Beispiel :**

  `less<int>()`
  ist ein – namenloses – **zweistelliges Funktionsobjekt** (expliziter Konstruktoraufruf) für den Vergleich zweier `int`-Werte.
  Seiner Operatorfunktion `operator()(a, b)` müssen die beiden zu vergleichenden Werte `a und b` als Parameter übergeben werden..

  mittels
  `bind2nd(less<int>(), 9)`
  wird hieraus ein **einstelliges Funktionsobjekt**, das einen `int`-Wert mit dem Wert `9` vergleicht.
  Der Operatorfunktion `operator()(a)` dieses Objekts ist nur der Wert `a` als einziger Parameter zu übergeben.

10.7 STL von C++ : Überblick

- **Einführung**
  - ◇ Die **Standard-Template-Library (STL)** ist ein wesentlicher Bestandteil der ANSI-C++-Standard-Bibliothek.
    Sie stellt ein relativ mächtiges und effizientes Werkzeug zur Verarbeitung von **Datenmengen** unterschiedlichster Typen zur Verfügung
  - ◇ Es handelt sich um eine **generische Bibliothek** auf der Basis von **C++-Templates.**
    Nicht der Typ der zu bearbeitenden Daten steht im Vordergrund, sondern die Art ihrer Verwaltung, der Zugriff zu ihnen und die auf sie anwendbaren Bearbeitungsoperationen.
  - ◇ Die STL besteht im wesentlichen aus **drei** aufeinander abgestimmten **Teilen** :
    - ▻ ***Containers Library*** : Definition von Container-Klassen.
      Container-Objekte speichern andere Objekte (→ Datenmengen)
    - ▻ ***Iterator Library*** : Definition von Iterator-Klassen.
      Iterator-Objekte ermöglichen den Zugriff zu den in Container-Objekten gespeicherten Objekten (den Elementen einer Menge)
    - ▻ ***Algorithm Library*** : Definition von Funktionen zur Bearbeitung von Datenmengen und Elementen von Datenmengen

- **Grundkonzept**
  - ◇ Den Kern der STL bilden **7 Container-Klassen** :
    `vector, list, deque, set, multiset, map` und **`multimap.`**
    Alle Container-Klassen sind als **Klassen-Templates** definiert. Dabei ist der Typ der in einem Container zu speichernden Objekte (Elemente) Template-Parameter (→ generische Programmierung).
    Ein bestimmter Container kann also immer nur Elemente eines bestimmten Typs speichern.
    Die verschiedenen Container-Klassen spiegeln die unterschiedlichen Möglichkeiten zur Realisierung und Verwaltung einer Datenmenge wieder. Dementsprechend besitzen sie jeweils spezifische Vor- und Nachteile.
  - ◇ Die für die verschiedenen Container-Klassen **implementierten Schnittstellen** sind so gehalten, dass wo immer möglich und sinnvoll **gleichnamige Memberfunktionen** (mit **gleichartiger Funktionalität**) existieren. So stehen u. a. für alle Container-Klassen gleichnamige Funktionen zum Ermitteln der Iteratorgrenzwerte, sowie die Vergleichsoperatoren und der Zuweisungsoperator zur Verfügung.
    → Die verschiedenen Containerklassen lassen sich teilweise **gleichartig verwenden**.
  - ◇ Zusätzlich zu den obigen fundamentalen Container-Klassen sind drei **Container-Adapter**-Klassen definiert :
    `stack, queue` und **`priority_queue.`**
    Diese Container-Adapter passen die Container-Klassen `vector, deque` und `list` an spezielle Anforderungen an.
  - ◇ Zu allen Container-Klassen (außer den Container-Adaptern) sind jeweils zugehörige **Iterator-Klassen** definiert.
    Iteratoren erlauben einen **zeigerähnlichen Zugriff** zu den **Elementen eines Containers**. Dabei bieten sie unabhängig von der jeweiligen Container-Klasse die **gleiche Schnittstelle** für den Element-Zugriff. Zu den in einem Container gespeicherten Elementen kann damit gleichartig – unabhängig von der internen Implementierung – zugegriffen werden. Da die Iteratoren containerklassen-spezifisch implementiert sind, berücksichtigen sie die im Container eingesetzte Datenorganisation. Es gibt verschiedene **Iterator-Kategorien**, die – zusätzlich zu einer für alle Iteratoren gleichartigen Grundfunktionalität – unterschiedliche Operationen zur Verfügung stellen.
    Nicht alle Iterator-Kategorien sind für alle Container-Klassen anwendbar.
  - ◇ Zahlreiche **Algorithmen** zur Bearbeitung von Mengen als Ganzes bzw. Mengenelementen (z.B. Traversieren, Suchen, Finden, Sortieren, Kopieren usw.) sind durch **freie Funktions-Templates** implementiert. Sie arbeiten mit **Iteratoren** (Template-Parameter, Funktions-Parameter). Viele dieser Funktionen können auf alle Container-Arten angewendet werden, andere nur auf solche, die eine spezielle Iterator-Kategorie anbieten.
    Häufig können einer Algorithmen-Funktion **Hilfsfunktionen als Parameter** (Funktions-Pointer oder Funktions-Objekte) übergeben werden, mit denen eine Anpassung des Algorithmus an spezielle Bedürfnisse erreicht wird.

Standard-Template-Library (STL) von C++ : Überblick (2)

- **Anmerkungen zu Container-Elementen**
  - ◇ Alle Container der STL besitzen eine **Wert-Semantik**.
    Das bedeutet, dass die in einen Container aufgenommenen Elemente als **Kopie** und nicht als Referenz (Pointer) **abgelegt** werden und auch **Kopien** der enthaltenen Elemente **zurückgeliefert** werden.
    Dies bedeutet, dass Container **keine Referenzen** enthalten können!
    Vorteile : Probleme, wie Verweise auf nicht mehr existierende Elemente, können nicht auftreten
    Nachteile : Das Kopieren der Elemente dauert i.a. länger als das Kopieren ihrer Adressen
    Ein Element kann nicht gleichzeitig von mehreren Containern änderbar verwaltet werden.
  - ◇ Die prinzipiell mögliche Verwendung von **Pointern als Container-Elemente** sollte i.a. **vermieden** werden.
    Hierdurch lässt sich zwar indirekt eine Referenz-Semantik implementieren, die aber Probleme beinhaltet:
    - So kann es vorkommen, dass als Elemente enthaltene Pointer auf Objekte zeigen, die gar nicht mehr existieren.
    - Außerdem arbeiten Vergleiche von Zeigern mit Adressen und nicht mit den durch die Zeiger referierten Objekten (Adress-Vergleich statt Werte-Vergleich).
  - ◇ Die Container der STL können prinzipiell **Elemente beliebigen Typs** verwalten.
    Allerdings müssen diese Typen die für das **Kopieren notwenigen Eigenschaften** implementieren :
    - ▻ Es muss ein **Copy-Konstruktor** öffentlich verfügbar sein (ein selbstdefinierter, wenn der Default-Copy-Konstruktor nicht richtig arbeiten würde, andernfalls reicht dieser aus). Er sollte ein gutes Zeitverhalten besitzen.
    - ▻ Es muss ein **Zuweisungsoperator** öffentlich verfügbar sein (ein selbstdefinierter, wenn der Default-Zuweisungs-operator nicht richtig arbeiten würde, andernfalls reicht dieser aus).
    - ▻ Der **Destruktor** muss öffentlich verfügbar sein, da ein Element beim Entfernen aus dem Container zerstört werden muss.
  - ◇ Darüber hinaus müssen die Element-Typen gegebenenfalls **weitere Anforderungen** erfüllen:
    - ▻ Für die Anwendung einiger Memberfunktionen bestimmter Container-Klassen muss der **Default-Konstruktor** definiert sein.
    - ▻ Für Suchfunktionen muss der **Vergleichsoperator** für **Gleichheit** (**`==`**) definiert sein.
    - ▻ Für Sortierfunktionen muss der **Vergleichsoperator** für "**kleiner als**" (**`<`**) definiert sein.

- **Fehlerbehandlung in der STL**
  - ◇ In der STL finden praktisch **keinerlei Überprüfungen** auf **Fehler** wie
    - Überschreitung von Bereichsgrenzen,
    - Verwendung von falschen, fehlerhaften oder ungültigen Iteratoren,
    - Zugriff zu nicht vorhandenen Elementen

    statt. Beim Auftritt derartiger Fehler ist das Verhalten der entsprechenden Bibliotheks-Komponente und damit des Programms undefiniert.
  - ◇ Lediglich **zwei Memberfunktionen** der Container-Klasse **`vector`** (**`at()`** und **`reserve()`**) sowie **eine Memberfunktion** der Container-Klasse **`deque`** (**`at()`**) können eine **Exception auslösen**.
  - ◇ Darüber hinaus können gegebenenfalls lediglich die von anderen – durch die STL benutzten – Bibliothekskomponenten oder sonstigen aufgerufenen Funktionen erzeugten Exceptions auftreten (z.B. `bad_alloc` bei Allokationsfehlern).
  - ◇ Die STL behandelt (fängt) diese Exceptions aber nicht. Tritt eine Exception auf, ist der Zustand aller beteiligten STL-Objekte undefiniert. Wird z. B. beim Einfügen eines Elements in einen Container eine Exception generiert, so gerät der Container in einen undefinierten Zustand, der seine weitere sinnvolle Verwendung verhindert.
  - ◇ Der Grund für die nicht vorhandene Fehlerüberprüfung liegt in dem **Grundgedanken**, die STL mit **optimalem Zeit-verhalten** zu implementieren. Fehlerüberprüfungen kosten aber Zeit.

10.8 (STL) von C++ : Container

- **Container-Arten**

  ◇ **Sequentielle Container**

  Sie speichern die Elemente als **geordnete Mengen**, in denen jedes Element eine bestimmte **Position** besitzt, die durch den **Zeitpunkt** und **Ort** des **Einfügens** festgelegt ist.
  Die enthaltenen Elemente sind damit **linear angeordnet** und können über ihre jeweilige Position angesprochen werden.
  Typischerweise erfolgt die Speicherung der Elemente in dynamischen **Arrays** bzw. **Listen**.

  ◇ **Assoziative Container**

  Sie speichern die Elemente als **sortierte Mengen**, in denen die **Position** eines Elementes durch ein **Sortierkriterium** bestimmt ist. → sehr gutes Zeitverhalten bei Suchen und Finden.
  Ein neues Element wird entsprechend dem Sortierkriterium **automatisch sortiert** eingefügt.
  Der Zugriff zu den Elementen erfolgt **assoziativ** über **Suchschlüssel** (*keys*).
  Typischerweise erfolgt die Speicherung der Elemente in einem balancierten **Binärbaum**.

- **Sequentielle Container**

  ◇ Die Klassen-Templates für die sequentiellen Container besitzen **2 Template-Parameter** :
  `<T, Allocator = allocator<T> >`

  ▻ **`T`** ist der **Datentyp** der zu **verwaltenden Objekte** (Elemente).

  ▻ **`Allocator`** ist eine **Allokator-Klasse**, die das Speichermodell für die zu verwendende dynamische Speicherverwaltung definiert. Als **Default** ist die Standard-Allokator-Klasse **`allocator<T>`**, die `new` und `delete` zur Speicherallokation verwendet, festgelegt.

  ◇ **Vektor :**

```
template <class T, class Allocator = allocator<T> > class vector;
```

  Ein Vektor (-Container) verwaltet die Elemente in einem dynamischen Array. Er ermöglicht einen direkten wahlfreien Zugriff zu den einzelnen Elementen. Hierfür existieren der Indexoperator und Direktzugriffs-Iteratoren. Das Anhängen und Löschen von Elementen am Ende des Arrays erfolgt optimal schnell.
  Ein Einfügen oder Löschen von Elementen mitten im Array ist dagegen zeitaufwändig (Verschieben von Elementen !).
  Vektoren sind daher bevorzugt einzusetzen, wenn Einfüge- und Löschoperationen vor allem am Ende stattfinden.

  ◇ **Deque :**

```
template <class T, class Allocator = allocator<T> > class deque;
```

  Deque = ***double ended queue***
  Ein Deque (-Container) verwaltet die Elemente in einem nach beiden Seiten offenen (verlängerbaren) dynamischen Array. Damit ist das Einfügen und Löschen von Elementen nicht nur am Ende sondern auch am Anfang optimal schnell, aber etwas langsamer als bei einem Vektor. Auch hier ist das Einfügen oder Löschen in der Mitte zeitaufwändig.
  Deques gestatten ebenfalls einen direkten wahlfreien Zugriff zu den einzelnen Elementen über einen Index bzw. mittels eines Direktzugriffs-Iterators.
  Sie sollten dann gewählt werden, wenn Einfüge- u./o. Löschoperationen häufig sowohl am Ende als auch am Anfang stattfinden.

  ◇ **Liste :**

```
template <class T, class Allocator = allocator<T> > class list;
```

  Ein Listen-Container (eine Liste) verwaltet seine Elemente in einer doppelt verketteten Liste.
  Ein direkter wahlfreier Zugriff zu den einzelnen Elementen ist nicht möglich, Indexoperator und Direktzugriffs-Iteratoren sind daher nicht implementiert. Das Einfügen und Löschen von Elementen erfolgt an allen Positionen gleich schnell. Listen sind immer dann bevorzugt einzusetzen, wenn viele Einfüge- u/o Löschoperationen insbesondere in der Mitte der gespeicherten Sequenz stattfinden.

Standard-Template-Library (STL) von C++ : Container (2)

- **Für alle Container-Klassen definierte Memberfunktionen**

  ▻ In der folgenden Zusammenstellung steht **`Container`** für eine der Container-Klassen,
  z.B. bei einem **Vektor** für : **`vector<T, Allocator>`**
  z.B. bei einer **Map** für : **`map<Key, T, Compare, Allocator>`**

  ▻ **`size_type`** ist ein implementierungsabhängiger innerhalb der jeweiligen Containerklasse definierter vorzeichenloser Ganzzahl-Typ. Außerhalb der Qualifizierung muss der **vollqualifizierende Typname** verwendet werden.
  Er wird über die Allokator-Klasse definiert und entspricht bei ISO-konformen Containern dem Typ **`std::size_t`**.

  ▻ Die mit ✡ markierten Funktionen sind für **Container-Adapter nicht** (explizit) definiert

Default-Konstruktor	Konstruktor für Default-Initialisierung des Containers
Konstruktor(en) mit Parametern	Konstruktor(en) für unterschiedliche Initialisierungsmethoden
✡ **Copy-Konstruktor**	Initialisierung eines Containers mit Kopie eines existierenden Containers
`size_type` **`size()`** `const;`	Anzahl der aktuell im Container enthaltenen Elemente (Container-Größe)
✡ `size_type` **`max_size()`** `const;`	maximale Anzahl der Elemente, die ein Container aufnehmen kann
`bool` **`empty()`** `const;`	`true`, wenn Container leer, sonst `false`
✡ `void` **`swap`**`(Container& b);`	vertauscht den Inhalt des aktuellen Containers mit dem des Containers b
✡ `void` **`clear`**`();`	entfernt alle Elemente aus dem Container
✡ `Container&` **`operator=`**`(const Container& b);`	Zuweisung des Inhalts von Container `b` an akt. Container
`iterator` **`insert`**`(iterator pos, const value_type& val);`	fügt ein neues Element mit dem Wert `val` an der Position `pos` ein. `value_type` ist ein container-spezifischer Typ
✡ `iterator` **`erase`**`(iterator pos);` ✡ `iterator` **`erase`**`(iterator fi, iterator la);` **Anm.** : bei assoziativen Containern Rückgabetyp : `void`	löscht das Element an der Iterator-Pos. `pos` löscht alle Elemente zwischen `fi` (einschliesslich) und `la` (ausschließlich) liefert Iterator, der auf das folgende Element zeigt
✡ `iterator` **`begin`**`();` ✡ `const_iterator` **`begin`**`() const;`	liefert Iterator , der auf das erste Element zeigt
✡ `iterator` **`end`**`();` ✡ `const_iterator` **`end`**`() const;`	liefert Iterator, der auf die Position unmittelbar nach dem letzen Element zeigt
✡ `reverse_iterator` **`rbegin`**`();` ✡ `const_reverse_iterator` **`rbegin`**`() const;`	liefert einen Reverse-Iterator, der auf die Position unmittelbar nach dem letzten Element zeigt
✡ `reverse_iterator` **`rend`**`();` ✡ `const_reverse_iterator` **`rend`**`() const;`	liefert einen Reverse-Iterator, der auf das erste Element zeigt

- **Für alle Container-Klassen definierte freie Funktionen**
  - ◇ Hierbei handelt es sich – bis auf eine Ausnahme – um Operatorfunktionen für die **Vergleichsoperatoren.**
  - ◇ **Anmerkungen :**
    - ▻ Die als **Funktions-Templates** definierten Funktionen werden in der folgenden Zusammenstellung in **vereinfachter Darstellung** angegeben.
      Dabei steht **`Container`** für eine der Container-Klassen (s. "Allen Containern gemeinsame Memberfunktionen")
      Statt z. B. die **nur** für **Vektoren** geltende Darstellung anzugeben

```
template <class T, class Allocator>
  bool operator==(const vector<T, Allocator>& x,
                  const vector<T, Allocator>& y);
```

      wird die **allgemeingültige Formulierung** verwendet:

```
bool operator==(const Container& x, const Container& y);
```

    - ▻ **Alle** hier aufgelisteten Funktionen sind **nicht** für den Container-Adapter **`priority_queue`** definiert.
    - ▻ Die mit ✡ markierte Funktion ist für **Container-Adapter nicht** (explizit) definiert

`bool operator==(const Container& x, const Container& y);`	`true` wenn `x==y`, sonst `false` Zwei Container derselben Klasse sind gleich, wenn sie die gleichen Elemente in der gleichen Reihenfolge besitzen. Für den Vergleich wird für alle Elemente der Reihe nach jeweils `operator==()` aufgerufen.
`bool operator!=(const Container& x, const Container& y);`	`true` wenn `x!=y`, sonst `false`
`bool operator<(const Container& x, const Container& y);`	`true` wenn `x<y`, sonst `false`
`bool operator<=(const Container& x, const Container& y);`	`true` wenn `x<=y`, sonst `false`
`bool operator> (const Container& x, const Container& y);`	`true` wenn `x>y`, sonst `false`
`bool operator>=(const Container& x, const Container& y);`	`true` wenn `x>=y`, sonst `false`
`void swap(Container& x, Container& y);`	vertauscht den Inhalt des Containers `x` mit dem Inhalt des Containers `y`

10.9 Klassen-Template `vector`

- **Eigenschaften**
  - ◇ Implementierung von **Vektoren**. Ein **Vektor** (-Container) verwaltet die Elemente in einem **dynamischen Array**. Die Elemente besitzen eine **definierte Reihenfolge** ("*ordered collection*")
  - ◇ Zu den einzelnen Elementen kann **direkt wahlfrei zugegriffen** werden.
    Hierfür existieren der **Indexoperator** und **Direktzugriffs-Iteratoren**.
  - ◇ Das **Anhängen** und **Löschen** von Elementen **am Ende** des Arrays erfolgt **optimal schnell**.
    Ein Einfügen oder Löschen von Elementen **mitten im Array** ist dagegen **zeitaufwändig** (Verschieben von Elementen!).
    → Vektoren sind daher **bevorzugt einzusetzen**, wenn **Einfüge- und Löschoperationen** vor allem **am Ende** stattfinden.
  - ◇ Die **Definition** des Klassen-Templates `vector<>` befindet sich in der **Headerdatei `<vector>`**

```
template <class T, class Allocator = allocator<T> >
  class vector
  { // ...
  };
```

- **Konstruktoren**

`explicit vector(const Allocator& = Allocator());`	Erzeugung eines Vektors der Länge 0
`explicit vector(size_type n, const T& val = T(), const Allocator& = Allocator());`	Erzeugung eines Vektors der Länge n, Initialisierung der Elemente mit dem Wert `val`
`template <class InputIterator> vector(InputIterator first, InputIterator last, const Allocator& = Allocator());`	Erzeugung eines Vektors, dessen Elemente mit den zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elementen eines anderen Containers initialisiert werden
`vector(const vector<T, Allocator>& x);`	Copy-Konstruktor

- **Zusätzliche spezifische Memberfunktionen (Auswahl)**
  - ◇ **Methoden zum Ermitteln und Ändern der Kapazität sowie Änderung der aktuellen Größe**

Kapazität = maximale Anzahl der Elemente, die der Vektor ohne Neuallokation enthalten kann.
= Größe des aktuell allozierten Arrays

`size_type capacity() const;`	Rückgabe der aktuellen Kapazität des Vektors
`void reserve(size_type n);`	Vergrößerung der Kapazität auf einen Wert `>=n`, wenn `n >` akt. Kapazität Keine Wirkung, wenn `n<=` aktuelle Kapazität Falls `n > max_size()` ist, wird Exception `length_error` geworfen **Achtung** : Alle **Referenzen**, **Pointer** und **Iteratoren** auf Elemente des Vektors werden nach einer Kapazitätsvergrößerung (Neu-Allokation !) **ungültig**
`void resize(size_type sz, T c= T());`	Veränderung der akt. Größe des Vektors auf den Wert `sz` (Vergrößerung oder Verkleinerung) bei Vergrößerung : Initialisierung der neuen Elemente mit dem Wert `c`

Standard-Template-Library (STL) von C++ : Klassen-Template `vector` (2)

- **Zusätzliche spezifische Memberfunktionen (Auswahl, Forts.)**

  ◇ **Methoden zum Elementzugriff**

`T& operator[](size_type n);` `const T& operator[](size_type n) const;`	Rückgabe des Elements an der Position mit dem Index `n` Falls `n >= size()` ist das Verhalten undefiniert
`T& at(size_type n);` `const T& at(size_type n) const;`	Rückgabe des Elements an der Position mit dem Index `n` Falls `n >= size()` wird Exception `out_of_range` geworfen
`T& front();` `const T& front() const;`	Rückgabe des ersten Elements (Index `0`)
`T& back();` `const T& back() const;`	Rückgabe des letzten Elements (Index `size()-1`)

  ◇ **Methoden zum Einfügen und Löschen von Elementen**

  Achtung : Durch das Einfügen mittels **`insert()`** werden alle **Referenzen**, **Pointer** und **Iteratoren** auf Elemente ab der Einfügeposition **ungültig**. Bei erforderlicher Neu-Allokation gilt das auch für alle übrigen Positionen.

`void push_back(const T& x);`	Einfügen des Objekts (Kopie) `x` als neues letztes Element
`void pop_back();`	Löschen des letzten Elements
`iterator insert(iterator pos, const T& x);`	Einfügen des Objekts `x` als neues Element an der Position `pos` (das bisherige Element an dieser Position und alle folgenden Elemente werden um eine Position weitergeschoben) Rückgabe : die Einfügeposition `pos`
`void insert(iterator pos,` `size_type n, const T& x);`	Einfügen von `n` Kopien des Objekts `x` als neue Elemente ab der Position `pos` (das bisherige Element an dieser Position und alle folgenden Elemente werden um `n` Positionen weitergeschoben)

`template <class InputIterator>`
`void insert(iterator pos,`
`InputIterator first,`
`InputIterator last);`
	Einfügen von Kopien der zwischen den Positionen `first` (einschl.) und `last` (ausschl.) eines anderen Containers befindlichen Elemente ab der Position `pos.`
	(das bisherige Element an dieser Position und alle folgenden Elemente werden um entsprechend viele Positionen weitergeschoben)

- **Einfaches Demonstrationsprogramm `vectordem`**

```
// C++-Quelldatei vectordem_m.cpp --> Programm vectordem
// Einfaches Demo-Programm zum Klassen-Template vector<> der STL

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

template <class T>
  void showSizeData(const vector<T>& vec, ostream& out)
  { out << "size()     : " << vec.size() << endl;
    out << "max_size() : " << vec.max_size() << endl;
    out << "capacity() : " << vec.capacity() << endl;
  }

template <class T>
  void showContent(const vector<T>& vec, ostream& out)
  { for (int i=0; i<vec.size(); i++)
      out << vec[i] << ' ';
    out << endl;
  }

int main(void)
{
  vector<string> satz;
  cout << "\nleerer string-Vector :\n";
  showSizeData(satz, cout);
  vector<string>::size_type nsz = 5;
  satz.reserve(nsz);
  cout << "\nnach reserve(" << nsz << ") :\n";
  showSizeData(satz, cout);
  satz.push_back("Achtung,");
  satz.push_back("dies");
  satz.push_back("ist");
  satz.push_back("ein");
  satz.push_back("Satz");
  satz.push_back("als");
  satz.push_back("Beispiel");
  satz.push_back("!");
  cout << "\nstring-Vector enthaelt jetzt " << satz.size() << " Elemente :\n";
  showSizeData(satz, cout);
  cout << "\nInhalt :\n";
  showContent(satz, cout);
  swap(satz[1], satz[2]);
  string hilf=satz[satz.size()-2];
  satz.insert(satz.begin()+4, hilf);
  satz.erase(satz.end()-1);
  satz.pop_back();
  satz.back()="?";
  satz.insert(satz.begin()+5, "fuer");
  satz.insert(satz.begin()+6, "einen");
  satz.insert(satz.begin()+7, "wirklich");
  satz.insert(satz.begin()+8, "guten");
  cout << "\nstring-Vector nach Manipulation :\n";
  showSizeData(satz, cout);
  cout << "\nInhalt :\n";
  showContent(satz, cout);
  return 0;
```

STL von C++ : Klassen-Template `vector`

- **Ausgabe des Demonstrationsprogramms `vectordem`**

```
leerer string-Vector :
size()     : 0
max_size() : 268435455
capacity() : 0

nach reserve(5) :
size()     : 0
max_size() : 268435455
capacity() : 5

string-Vector enthaelt jetzt 8 Elemente :
size()     : 8
max_size() : 268435455
capacity() : 10

Inhalt :
Achtung, dies ist ein Satz als Beispiel !

string-Vector nach Manipulation :
size()     : 11
max_size() : 268435455
capacity() : 20

Inhalt :
Achtung, ist dies ein Beispiel fuer einen wirklich guten Satz ?
```

10.10 STL von C++ : Klassen-Template list

- **Eigenschaften**
  - ◇ Implementierung von **Listen-Containern** (Listen)
    Ein Listen-Container verwaltet seine Elemente in einer **doppelt verketteten Liste**.
  - ◇ Die Elemente besitzen eine **definierte Reihenfolge**, ein direkter **wahlfreier Zugriff** zu einzelnen Elementen ist jedoch **nicht möglich**. Indexoperator und Direktzugriffs-Iteratoren sind daher nicht implementiert.
  - ◇ Durch das **Einfügen** oder **Löschen** von Elementen werden **Verweise** auf andere Elemente **nicht ungültig**
  - ◇ Das **Einfügen** und **Löschen** von Elementen erfolgt an **allen Positionen gleich schnell**.
    Listen sind daher immer dann bevorzugt einzusetzen, wenn viele Einfüge- u/o Löschoperationen insbesondere in der Mitte der gespeicherten Sequenz stattfinden.
  - ◇ Die **Definition** des Klassen-Templates `list<>` befindet sich in der **Headerdatei `<list>`**

```
template <class T, class Allocator = allocator<T> >
  class list
  { // ...
  };
```

- **Konstruktoren**

`explicit list(const Allocator& = Allocator());`	Erzeugung einer Liste der Länge 0
`explicit list(size_type n, const T& val = T(), const Allocator& = Allocator());`	Erzeugung einer Liste der Länge n, Initialisierung der Elemente mit dem Wert `val`
`template <class InputIterator> list(InputIterator first, InputIterator last, const Allocator& = Allocator());`	Erzeugung einer Liste, deren Elemente mit den zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elementen eines anderen Containers initialisiert werden
`list(const list<T, Allocator>& x);`	Copy-Konstruktor

- **Zusätzliche spezifische Memberfunktionen (Auswahl)**
  - ◇ **Methode zur Änderung der aktuellen Größe**

`void resize(size_type sz, T c= T());`	Veränderung der akt. Größe der Liste auf den Wert `sz` (Vergrößerung oder Verkleinerung) bei Vergrößerung : Initialisierung der neuen Elemente mit dem Wert `c`

  - ◇ **Methoden zum Elementzugriff**

`T& front();` `const T& front() const;`	Rückgabe des ersten Elements
`T& back();` `const T& back() const;`	Rückgabe des letzten Elements

STL von C++ : Klassen-Template `list` (2)

- **Zusätzliche spezifische Memberfunktionen (Auswahl, Forts.)**
  - ◇ **Methoden zum Einfügen und Löschen von Elementen**

    Achtung: Durch das Einfügen oder Löschen werden **Referenzen**, **Pointer** und **Iteratoren** auf Elemente der Liste **nicht ungültig** (außer Verweise auf die gelöschten Elemente)

`void push_back(const T& x);`	Einfügen des Objekts (Kopie) `x` als neues letztes Element
`void push_front(const T& x);`	Einfügen des Objekts (Kopie) `x` als neues erstes Element
`void pop_back();`	Löschen des letzten Elements
`void pop_front();`	Löschen des ersten Elements
`iterator insert(iterator pos, const T& x);`	Einfügen des Objekts `x` als neues Element an der Position `pos` (das bisherige Element an dieser Position und alle folgenden Elemente werden um eine Position weitergeschoben) Rückgabe : die Einfügeposition `pos`
`void insert(iterator pos,` `  size_type n, const T& x);`	Einfügen von `n` Kopien des Objekts `x` als neue Elemente ab der Position `pos` (das bisherige Element an dieser Position und alle folgenden Elemente werden um `n` Positionen weitergeschoben)

`template <class InputIterator>`
`  void insert(iterator position,`
`  InputIterator first,`
`  InputIterator last);`
	Einfügen von Kopien der zwischen den Positionen `first` (einschl.) und `last` (ausschl.) eines anderen Containers befindlichen Elemente ab der Position `pos`.
	(das bisherige Element an dieser Position und alle folgenden Elemente werden um entsprechend viele Positionen weitergeschoben)

`void remove(const T& val);`	Löschen aller Elemente, die den Wert `val` besitzen
`template <class Predicate>` `  void remove_if(Predicate pred);`	Löschen aller Elemente, für die das als Parameter übergebene Funktionsobjekt `pred` den Wert `true` liefert
`void unique();`	Löschen aller Elemente jeder Gruppe aufeinanderfolgender gleicher Elemente außer dem jeweils ersten.
`template <class BinaryPredicate>` `  void unique(BinaryPredicate binpred);`	Löschen aller direkten Nachfolger-Elemente eines Elements, für die das als Parameter übergebene Funktionsobjekt `binpred` den Wert `true` liefert. `binpred` muss ein zweistelliges Funktionsobjekt sein, das auf jedes Element und sein jeweiliges direktes Nachfolger-Element angewendet wird.

Standard-Template-Library (STL) von C++: Klassen-Template `list` (3)

- **Zusätzliche spezifische Memberfunktionen (Auswahl, weitere Forts.)**
  - ◇ **Weitere Methoden zur Modifikation von Listen**

`void` **`splice`**`(iterator pos, list<T, Allocator>& x);`	Einfügen aller Elemente der Liste `x` in die aktuelle Liste vor die Position `pos`. Aus der Liste `x` werden alle Elemente entfernt
`void` **`splice`**`(iterator pos, list<T, Allocator>& x, iterator i);`	Einfügen des durch den Iterator `i` referierten Elements der Liste `x` in die aktuelle Liste vor die Position `pos`. Das Element wird aus der Liste `x` entfernt Die Liste `x` darf mit der akt. Liste identisch sein.
`void` **`splice`**`(iterator pos, list<T, Allocator>& x, iterator first, iterator last);`	Einfügen der zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elemente der Liste `x` in die aktuelle Liste vor die Position `pos`. Die eingefügten Elemente werden aus der Liste `x` entfernt. Die Liste `x` darf mit der akt. Liste identisch sein. Falls in diesem Fall `pos` zwischen `first` und `last` liegt, ist das Verhalten undefiniert
`void` **`merge`**`(list<T, Allocator>& x);`	Einfügen aller Elemente der Liste `x` in die akt. Liste unter Beibehaltung einer Sortierreihenfolge. Beide Listen müssen nach dieser Reihenfolge sortiert sein. Aus der Liste `x` werden alle Elemente entfernt
`template <class Compare>` `  void` **`merge`**`(list<T, Allocator>& x, Compare comp);`	Einfügen aller Elemente der Liste `x` in die akt. Liste unter Beibehaltung der durch den Parameter `comp` bestimmten Sortierreihenfolge. Beide Listen müssen nach dieser Reihenfolge sortiert sein. Aus der Liste `x` werden alle Elemente entfernt
`void` **`sort`**`();`	Sortieren der akt. Liste unter Verwendung von `operator<()`. Diese Operatorfunktion muss für den Typ `T` definiert sein
`template <class Compare>` `  void` **`sort`**`(Compare comp);`	Sortieren der akt. Liste unter Verwendung des Funktionsobjekts `comp`
`void` **`reverse`**`();`	Invertierung der Reihenfolge der Listenelemente

STL von C++ : Klassen-Template `list` (4)

- **Einfaches Demonstrationsprogramm `listdem`**

```
// C++-Quelldatei listdem_m.cpp --> Programm listdem
// Einfaches Demo-Programm zum Klassen-Template list<> der STL
#include <list>
#include <iostream>
#include <cstdlib>
using namespace std;

template <class T>
  void showSizeData(const list<T>& lst, ostream& out)
  { out << "size()     : " << lst.size() << endl;
    out << "max_size() : " << lst.max_size() << endl;
  }

template <class T>
  void showContent(list<T>& lst, ostream& out)
  { list<T>::iterator it=lst.begin();
    if (it==lst.end()) out << "leer";
    else
      for (; it!=lst.end(); ++it)
        out << *it << "  ";
    out << endl;
  }

int main(void)
{
  list<int> il1, il2, il3;
  cout << "\nleere int-Liste :\n"; showSizeData(il1, cout);
  for (int i=0; i<10; i++)
  { il1.push_back(rand()%11+1);
    il2.push_front(i+1);
    il3.push_back(rand()%12);
  }
  cout << "\nInhalt Liste 1 :\n"; showContent(il1, cout);
  cout << "\nInhalt Liste 2 :\n"; showContent(il2, cout);
  cout << "\nInhalt Liste 3 :\n"; showContent(il3, cout);
  il1.splice(il1.begin(), il3);
  cout << "\nInhalt Liste 1 nach il1.splice(il1.begin(), il3) :\n";
  showContent(il1, cout);
  cout << "\nInhalt Liste 3 nach il1.splice(il1.begin(), il3) :\n";
  showContent(il3, cout);
  il1.sort();
  il2.reverse();
  cout << "\nInhalt Liste 1 nach sort() :\n";
  showContent(il1, cout);
  cout << "\nInhalt Liste 2 nach reverse() :\n";
  showContent(il2, cout);
  il1.merge(il2);
  cout << "\nInhalt Liste 1 nach il1.merge(il2) :\n";
  showContent(il1, cout);
  cout << "\nInhalt Liste 2 nach il1.merge(il2) :\n";
  showContent(il2, cout);
  il1.unique();
  cout << "\nInhalt Liste 1 nach unique() :\n";
  showContent(il1, cout);
  il1.remove_if(bind2nd(not_equal_to<int>(), 2));
  cout << "\nInhalt Liste 1 nach remove_if(...) :\n";
  showContent(il1, cout);
  return 0;
}
```

Standard-Template-Library (STL) von C++ : Klassen-Template `list` (5)

- **Ausgabe des Demonstrationsprogramms `listdem`**

```
leere int-Liste :
size()     : 0
max_size() : 1073741823 (3fffffff)

Inhalt Liste 1 :
9 10 8 6 2 8 6 7 4 10

Inhalt Liste 2 :
10 9 8 7 6 5 4 3 2 1

Inhalt Liste 3 :
11 4 4 6 8 5 3 11 2 0

Inhalt Liste 1 nach il1.splice(il1.begin(), il3) :
11 4 4 6 8 5 3 11 2 0 9 10 8 6 2 8 6 7 4 10

Inhalt Liste 3 nach il1.splice(il1.begin(), il3) :
leer

Inhalt Liste 1 nach sort() :
0 2 2 3 4 4 4 5 6 6 6 7 8 8 8 9 10 10 11 11

Inhalt Liste 2 nach reverse() :
1 2 3 4 5 6 7 8 9 10

Inhalt Liste 1 nach il1.merge(il2) :
0 1 2 2 2 3 3 4 4 4 4 5 5 6 6 6 6 7 7 8 8 8 8 9 9 10 10 10 11 11

Inhalt Liste 2 nach il1.merge(il2) :
leer

Inhalt Liste 1 nach unique() :
0 1 2 3 4 5 6 7 8 9 10 11

Inhalt Liste 1 nach remove_if(...) :
2
```

10.11 (STL) von C++: Container-Adapter

- **Klassen-Template `stack<>`**
  - ◇ Implementiert die Funktionalität eines **Stacks** (Kellerspeicher, **LIFO**) unter Verwendung der Container-Klassen `deque` (Default), `vector` oder `list`.
  - ◇ Die **Definition** befindet sich in der Headerdatei **`<stack>`**

```
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& co = Container()) : c(co) {}
      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();     }
  };
```

- **Klassen-Template `queue<>`**
  - ◇ Implementiert die Funktionalität eines **Pufferspeichers** (**FIFO**) unter Verwendung der Container-Klassen `deque` (Default) oder `list`.
  - ◇ Die **Definition** befindet sich in der Headerdatei **`<queue>`**

```
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 :
      explicit queue(const Container& co = Container()) : c(co) {}
      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();    }
  `
```

Standard-Template-Library (STL) von C++ : Container-Adapter (2)

- **Klassen-Template `priority_queue<>`**
  - ◇ Implementiert einen **Pufferspeicher**, bei der die Elemente nach einer durch ein **Sortierkriterium** festgelegten "Priorität" wieder ausgelesen werden können. Es wird immer das Element mit der **höchsten Priorität** zurückgeliefert.
  - ◇ Das Sortierkriterium kann als Template-Parameter übergeben werden. **Default** ist die Funktionsobjekt-Klasse **`less<T>`**, d. h. die **höchste Priorität** hat das **größte** Element.
  - ◇ Priority-Queues verwenden zur Speicherung ihrer Elemente die Container-Klasse `vector` (Default) oder `deque`.
  - ◇ Für Priority-Queues sind **keine Vergleichsoperatoren** definiert.
  - ◇ Die **Definition** befindet sich in der Headerdatei **`<queue>`**

```
template <class T, class Container = vector<T>,
          class Compare = less<typename 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& x = Compare(),
                              const Container& co = Container())
                             : c(co), comp(x) {}

      template <class InputIterator>
        priority_queue (InputIterator first, InputIterator last,
                        const Compare& x = Compare(),
                        const Container& co = Container())
                       : c(co), comp(x)
       { c.insert(c.end(),first,last);
         make_heap(c.begin(), c.end(), comp);
       }

      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)
      { c.push_back(x);
        push_heap(c.begin(), c.end(), comp);
      }

      void pop()
      { pop_heap(c.begin(), c.end(), comp);
        c.pop_back();
      }
```

  - ◇ Die zur **Implementierung** der Priority-Queue verwendeten freien Funktionen **`make_heap()`**, **`push_heap()`** und **`pop_heap()`** sind Funktionen aus der **Algorithmen-Bibliothek** der STL.
    Sie stellen sicher, dass der Container als "**Heap**" verwaltet wird, d.h. das sein erstes Element immer das Element mit der **"höchsten" Erfüllung** des **Vergleichskriteriums** ist.

Standard-Template-Library (STL) von C++ : Container-Adapter (3)

- **Einfaches Demonstrationsprogramm `priqueuedem` zum Container-Adapter `priority_queue`**

```
// C++-Quelldatei priqueuedem_m.cpp --> Programm priqueuedem
// Einfaches Demo-Programm zum Container-Adapter priority_queue<> der STL

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

int main(void)
{
  priority_queue<double> pqd;
  double w;
  cout << "\nleere double-Priority-Queue :\n";
  cout << "size() : " << pqd.size() << endl;
  cout << "\nAblage der folgenden Elemente :\n";
  pqd.push(w=27.33); cout << w << endl;
  pqd.push(w=63.12); cout << w << endl;
  pqd.push(w=12.84); cout << w << endl;
  pqd.push(w=21.37); cout << w << endl;
  cout << "\nEntfernen der beiden \"obersten\" Elemente :\n";
  cout << pqd.top() << endl;
  pqd.pop();
  cout << pqd.top() << endl;
  pqd.pop();
  cout << "\nAblage eines weiteren Elements :\n";
  pqd.push(w=99.88); cout << w << endl;
  cout << "\nEntfernen der beiden \"obersten\" Elemente :\n";
  cout << pqd.top() << endl;
  pqd.pop();
  cout << pqd.top() << endl;
  pqd.pop();
  cout << "\nverbleibende Anzahl von Elementen :\n";
  cout << "size() : " << pqd.size() << endl;
  return 0;
```

- **Ausgabe des Demonstrationsprogramm `priqueuedem`**

```
leere double-Priority-Queue :
size() : 0

Ablage der folgenden Elemente :
27.33
63.12
12.84
21.37

Entfernen der beiden "obersten" Elemente :
63.12
27.33

Ablage eines weiteren Elements :
99.88

Entfernen der beiden "obersten" Elemente :
99.88
21.37

verbleibende Anzahl von Elementen :
```

10.12 STL von C++ : Klassen-Template `set` (1)

- **Eigenschaften**
  - ◇ Implementierung von **Sets** (Set-Containern)
    Ein Set (-Container) verwaltet nach ihrem "Wert" **sortierte Elemente** – typischerweise in einem balancierten Binärbaum. Das Sortierkriterium (Funktionsobjekt-Klasse) ist Template-Parameter (default : `less<>`)
    Jedes Element darf **nur einmal** in einem Set vorkommen.
  - ◇ Aufgrund der automatischen Sortierung in einem Binärbaum ermöglichen Sets ein **schnelles Suchen** und **Finden** von Elementen. Suchschlüssel sind die Elemente selbst.
    Auch das **Einfügen** u/o. **Löschen** von Elementen besitzt an **allen Stellen** ein **gutes Zeitverhalten**.
  - ◇ Die Elemente von Sets können auch **sequentiell durchlaufen** werden.
    Ein direkter **wahlfreier Zugriff** zu einzelnen Elementen ist jedoch **nicht möglich**. Indexoperator und Direktzugriffs-Iteratoren sind daher nicht implementiert.
    Beim Zugriff zu Elementen über Iteratoren werden die Elemente als Konstante betrachtet.
  - ◇ Die Elemente können **nicht direkt geändert** werden. Eine Änderung könnte die Sortierung ungültig machen. Um ein Element zu ändern, muß es daher aus dem Set entfernt werden und nach der Änderung als neues Element wieder eingefügt werden.
  - ◇ Die **Definition** des Klassen-Templates `set<>` befindet sich in der **Headerdatei `<set>`**

```
template <class Key, class Compare = less<Key>,
          class Allocator = allocator<Key> >
  class set
  { // ...
  };
```

- **Konstruktoren**

`explicit set(const Compare& = Compare(), const Allocator& = Allocator());`	Erzeugung eines leeren Sets
`template <class InputIterator> set(InputIterator first, InputIterator last, const Compare& =Compare(), const Allocator& = Allocator());`	Erzeugung eines zunächst leeren Sets, in das dann die zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elemente eines anderen Containers eingefügt werden
`set(const set<Key, Compare, Allocator>& x);`	Copy-Konstruktor

- **Zusätzliche spezifische Memberfunktionen (Auswahl)**
  - ◇ **Zusätzliche Methode zum Löschen eines Elements**

`size_type erase(const Key& x);`	Löschen des Elements `x` Rückgabewert = 1, falls gelöscht wurde (Element war enthalten) 0, falls nicht gelöscht wurde (Element war nicht enthalten)

Standard-Template-Library (STL) von C++ : Klassen-Template `set` (2)

- **Zusätzliche spezifische Memberfunktionen (Auswahl, Forts.)**

  ◇ **Methoden zum Einfügen von Elementen**

`pair<iterator, bool> insert(const Key& x);`	Einfügen des Objekts `x` als neues Element, falls es noch nicht im Set vorhanden ist. **Rückgabewert** : Wertepaar mit Einfügeposition (Komponente `first`) und Einfügeerfolg (Komponente `second`) Einfügeerfolg = `true`, wenn eingefügt wurde
`iterator insert(iterator pos, const Key& x);`	Einfügen des Objekts `x` als neues Element, falls es noch nicht im Set vorhanden ist. Der Parameter `pos` dient als ein Hinweis, wo im Set mit der Suche nach der Einfügeposition begonnen werden sollte. **Rückgabewert** : Einfügeposition bzw Position, an der sich das Element bereits befindet.
`template <class InputIterator> void insert(InputIterator first, InputIterator last);`	Einfügen von Kopien der zwischen den Positionen `first` (einschl.) und `last` (ausschl.) eines anderen Containers befindlichen Elemente. Ein Element wird nur eingefügt, wenn es noch nicht im Set vorhanden ist.

  ◇ **Methoden zum Suchen**

`iterator find(const Key& x) const;`	Suchen des Elements, das gleich dem Objekt `x` ist. **Rückgabewert** : - Position von `x`, falls vorhanden - `end()`, falls nicht vorhanden
`size_type count(const Key& x) const;`	Ermittlung, ob das Objekt `x` im Set enthalten ist **Rückgabewert** : - `1`, falls vorhanden - `0`, falls nicht vorhanden
`iterator lower_bound(const Key& x) const;`	Rückgabe der Position des ersten Elements im Set, das nicht kleiner als `x` (d.h. größer gleich `x`) ist
`iterator upper_bound(const Key& x) const;`	Rückgabe der Position des ersten Elements im Set, das größer als `x` ist
`pair<iterator, iterator> equal_range(const Key& x) const;`	Rückgabe eines `pair`-Objekts aus `lower_bound(x)` und `upper_bound(x)`. `lower_bound(x)` und `upper_bound(x)` sind nur dann verschieden, wenn das Objekt `x` im Set enthalten ist

Standard-Template-Library (STL) von C++ : Klassen-Template `set` (3)

```
//set_simple
#include <iostream>
#include <set>  //Headerdatei für Set u.Map
using namespace std;

int main()
{
      //Set anlegen
      //set<int> menge;  //aufsteigend sortiert
      //typedef set<int> IntSet;
      typedef set<int, greater<int> > IntSet; //absteigend sortiert
      IntSet menge;
      menge.insert(2);
      menge.insert(3);
      menge.insert(1);

      //insert liefert Wertepaar Einfügeposition und Erfolg
      pair<IntSet::iterator, bool> success= menge.insert(1);
      if(success.second) cout << "Einfuegen erfolgreich" <<endl;
      else               cout << "Einfuegen nicht erfolgreich" <<endl;

      //Iterator
      //set<int>::iterator pos;
      IntSet::iterator pos;

      for(pos=menge.begin(); pos!= menge.end(); ++pos)
      {
            cout << *pos << endl;
      }
      cout << endl;
      return 0;
}
```

Ausgabe des Demonstrationsprogramms `set_simple`:

```
3
2
1
Einfügen nicht erfogreich
```

Standard-Template-Library (STL) von C++ : Klassen-Template `set` (4)

- **Einfaches Demonstrationsprogramm `setdem`**

```
// C++-Quelldatei setdem_m.cpp --> Programm setdem
// Einfaches Demo-Programm zum Klassen-Template set<> der STL

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

typedef set<string, greater<string> > StringDownSet;
typedef set<string>                   StringSet;

template <class SetType>
void einfuegen(SetType& sds, const string& str)
{ pair<SetType::iterator, bool> success = sds.insert(str);
  cout << "\"" << str << "\"";
  if (success.second) cout << " erfolgreich eingefuegt\n";
  else cout << " bereits vorhanden\n";
}

ostream& operator<<(ostream& out, StringSet& menge)
{ StringSet::iterator loc;
  for (loc=menge.begin(); loc!=menge.end(); ++loc)
    out << *loc << ' ';
  return out << endl;
}


int main(void)
{ StringDownSet wmenge1;
  wmenge1.insert("man");
  wmenge1.insert("politikern");
  wmenge1.insert("kann");
  wmenge1.insert("tauben");
  wmenge1.insert("zutrauen");
  wmenge1.insert("sowie");
  einfuegen(wmenge1, "kann");
  einfuegen(wmenge1, "wesentliches");
  cout << endl;
  StringDownSet::iterator pos;
  for (pos=wmenge1.begin(); pos!=wmenge1.end(); ++pos)
    cout << *pos << ' ';
  cout << endl;

  //Copy Konstruktor
  //In wmenge2 wird wmenge2 wmenge1 eingefügt
  StringSet wmenge2(wmenge1.begin(), wmenge1.end());

  cout << wmenge2;
  wmenge2.erase(wmenge2.find("sowie"), wmenge2.find("wesentliches"));
  cout << wmenge2;
  wmenge2.erase(wmenge2.begin(), wmenge2.find("politikern"));
  cout << wmenge2 << endl;
  einfuegen(wmenge2, "niemals");
  cout << endl;
  cout << wmenge2;
```

Standard-Template-Library (STL) von C++ : Klassen-Template `set` (4)

- **Ausgabe des Demonstrationsprogramms `setdem`**

```
"kann" bereits vorhanden
"wesentliches" erfolgreich eingefuegt

zutrauen wesentliches tauben sowie politikern man kann
kann man politikern sowie tauben wesentliches zutrauen
kann man politikern wesentliches zutrauen
politikern wesentliches zutrauen

"niemals" erfolgreich eingefuegt

niemals politikern wesentliches zutrauen
```

10.13 (STL) von C++ : Klassen-Template `multiset`

- **Eigenschaften**
  - ◇ Implementierung von **Multisets** (Multiset-Containern)
    Ein Multiset (-Container) entspricht einem Set mit dem Unterschied, dass Elemente auch mehrfach enthalten sein können.
  - ◇ Die **Definition** des Klassen-Templates `multiset<>` befindet sich ebenfalls in der **Headerdatei `<set>`**

```
template <class Key, class Compare = less<Key>,
          class Allocator = allocator<Key> >
  class multiset
  { // ...
  };
```

- **Konstruktoren**

`explicit multiset(const Compare& = Compare(), const Allocator& = Allocator());`	Erzeugung eines leeren Multisets
`template <class InputIterator> multiset(InputIterator first, InputIterator last, const Compare& =Compare(), const Allocator& = Allocator());`	Erzeugung eines zunächst leeren Multisets, in das dann die zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elemente eines anderen Containers eingefügt werden
`multiset(const multiset<Key, Compare, Allocator>& x);`	Copy-Konstruktor

- **Anmerkungen zu den zusätzlichen spezifischen Memberfunktionen**
  - ◇ Das Klassen-Template `multiset` besitzt die **gleichem Memberfunktionen** wie das Klassen-Template **`set`.**
    Sie **unterscheiden** sich lediglich durch die **Auswirkungen** des möglichen **Mehrfachenthaltenseins** von Elementen.
  - ◇ Im wesentlichen bestehen die **folgenden Unterschiede** :
    - ▻ **`erase`**`(x)` Löschen aller Elemente, die gleich dem Objekt `x` sind.
    - ▻ **`insert`**`(x)` Der Rückgabewert ist vom Typ `iterator` (statt `pair<...>`)
      Einfügen des Objekts `x` als neues Element auch dann, wenn es bereits im Multiset enthalten ist. Es wird immer die Einfügeposition zurückgegeben.
    - ▻ **`insert`**`(pos, x)` Einfügen des Objekts `x` als neues Element auch dann, wenn es bereits im Multiset enthalten ist. Der Rückgabewert ist immer die Einfügeposition
    - ▻ **`insert`**`(first, last)` Einfügen immer aller Elemente aus dem Bereich `first` (einschl.) bis `last` (ausschl.), auch die, die bereits vorhanden sind.
    - ▻ **`count`**`(x)` Rückgabe der Anzahl Elemente, die gleich dem Objekt `x` sind
    - ▻ **`find`**`(x)` Suchen des ersten Elements, das gleich dem Objekt `x` ist.

10.14 STL von C++ : Klassen-Template `map`

- **Eigenschaften**
  - ◇ Implementierung von **Maps** (Map-Containern)
    Ein(e) Map (-Container) speichert **Schlüssel-/Werte-Paare** als Elemente (Klasse **`pair<const Key, T>`**).
    Die Elemente sind nach dem **(Such-)Schlüssel** (Klasse `Key`) **sortiert**. Der Suchschlüssel dient zum Auffinden des mit ihm assoziierten Werts (das eigentliche verwaltete Objekt, Klasse `T`). Jeder **Suchschlüssel** darf **nur einmal** in einer Map vorkommen, d.h. mit einem Suchschlüssel kann nur ein einziger Wert assoziiert sein ("***one-to-one mapping***")
  - ◇ Auch eine Map verwaltet ihre Elemente typischerweise in einem balancierten Binärbaum. Das Sortierkriterium (Funktionsobjekt-Klasse) ist ebenfalls Template-Parameter (default : `less<>`)
  - ◇ Der **Schlüssel** eines Elements darf **nicht geändert** werden, wohl **aber** sein **Wert**.
  - ◇ Die übrigen Eigenschaften (bezüglich Zeitverhalten bei Suchen/Finden und Einfügen/Löschen sowie den Zugriffsmöglichkeiten zu den Elementen) entsprechen denen eines Sets, mit dem Unterschied, dass die enthaltenen Elemente Paare sind.
  - ◇ Die **Definition** des Klassen-Templates `map<>` befindet sich in der **Headerdatei `<map>`**

```
template <class Key, class T,
          class Compare = less<Key>,
          class Allocator = allocator<pair<const Key,T> > >
  class map
  { // ...
  };
```

- **Konstruktoren**

`explicit map(const Compare& = Compare(), const Allocator& = Allocator());`	Erzeugung einer leeren Map
`template <class InputIterator> map(InputIterator first, InputIterator last, const Compare& =Compare(), const Allocator& = Allocator());`	Erzeugung einer zunächst leeren Map, in die dann die zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elemente eines anderen Containers eingefügt werden Diese Elemente müssen vom Typ `pair<const Key, T>` sein.
`map(const map<Key, T, Compare, Allocator>& x);`	Copy-Konstruktor

- **Zusätzliche spezifische Memberfunktionen (Auswahl)**
  - ◇ **Zusätzliche Methode zum Löschen eines Elements**

`size_type erase(const Key& k);`	Löschen des Elements (Schlüssel-/Wert-Paar) mit dem Schlüssel `k` Rückgabewert = - 1, falls gelöscht wurde (Element mit Schlüssel `k` war enthalten) - 0, falls nicht gelöscht wurde (kein Element mit Schlüssel `k` enthalten.)

Standard-Template-Library (STL) von C++ : Klassen-Template `map` (2)

- **Zusätzliche spezifische Memberfunktionen (Auswahl, Forts.)**

  ◇ **Methoden zum Einfügen von Elementen**

`pair<iterator, bool>` `  insert(pair<const Key, T>& x);`	Einfügen des Schlüssel-/Werte-Paares `x` als neues Element, falls noch kein Element mit dem Schlüssel `x.first` in der Map vorhanden ist. **Rückgabewert** : Wertepaar mit Einfügeposition (Komponente `first`) und Einfügeerfolg (Komponente `second`). Einfügeerfolg = `true`, wenn eingefügt wurde
`iterator insert(iterator pos,` `  pair<const Key, T>& x);`	Einfügen des Schlüssel-/Werte-Paares `x` als neues Element, falls noch kein Element mit dem Schlüssel `x.first` in der Map vorhanden ist. Der Parameter `pos` dient als Hinweis, wo in der Map mit der Suche nach der Einfügeposition begonnen werden sollte. **Rückgabewert** : Einfügeposition bzw Position, an der sich ein Element mit dem Schlüssel bereits befindet.

`template <class InputIterator>`
`  void insert(InputIterator first,`
`  InputIterator last);`
	Einfügen von Kopien der zwischen den Positionen `first` (einschl.) und `last` (ausschl.) eines anderen Containers (i. a. einer Map oder Multimap) befindlichen Elemente.
	Ein Element wird nur eingefügt, wenn es noch kein Element mit seinem Schlüssel in der Map gibt.

  ◇ **Methoden zum Suchen**

`iterator find(const Key& k);` `const_iterator find(const Key& k) const;`	Suchen des Elements, das den Schlüssel `k` hat. **Rückgabewert** : - Position des Elements, falls vorhanden - `end()`, falls nicht vorhanden
`size_type count(const Key& k) const;`	Ermittlung, ob ein Element mit dem Schlüssel `k` in der Map enthalten ist **Rückgabewert** : - `1`, falls vorhanden - `0`, falls nicht vorhanden
`iterator lower_bound(const Key& k);` `const_iterator lower_bound(const Key& k) const;`	Rückgabe der Position des ersten Elements in der Map, dessen Schlüssel nicht kleiner als `k` (d.h. größer gleich `k`) ist
`iterator upper_bound(const Key& k);` `const_iterator upper_bound(const Key& k) const;`	Rückgabe der Position des ersten Elements in der Map, dessen Schlüssel größer als `k` ist

`pair<iterator, iterator>`
`  equal_range(const Key& k);`
 	
`pair<const_iterator, const_iterator>`
`  equal_range(const Key& k) const;`
	Rückgabe eines `pair`-Objekts aus `lower_bound(k)` und `upper_bound(k)`. `lower_bound(k)` und `upper_bound(k)` sind nur dann verschieden, wenn ein Element mit dem Schlüssel `k` in der Map enthalten ist

Standard-Template-Library (STL) von C++ : Klassen-Template `map` (3)

- **Zusätzliche spezifische Memberfunktionen (Auswahl, weitere Forts.)**
  - ◇ **Methode zum direkten Elementzugriff (Indexoperator)**

`T& operator[](const Key& k);`	Ermittlung der Wert-Komponente des Elements mit dem Schlüssel `k`. Falls kein Element mit dem Schlüssel `k` existiert, wird ein neues Element angelegt (dessen Wert-Komponente mit ihrem Default-Konstruktor initialisiert wird)

Diese Methode macht eine Map zur Implementierung eines **assoziativen Arrays**.
Die Auswahl eines "Array"-Elements erfolgt über einen Teil seines Inhalts (hier den Schlüssel `k`).
Der Auswahl-"Index" kann damit von einem beliebigen (und nicht nur einem ganzzahligen) Typ sein.

Interessant ist, dass es **keinen unerlaubten** "Wert" für den **Index** gibt.
Existiert kein Element für den Index, wird ein neues Element erzeugt.

Beispiel :
```
// ...
map<string, float> preisliste;   // leere Map
preisliste["Uhr"]=24.50;
// ...
```
→ Es wird ein neues Element mit dem Schlüssel `"Uhr"` angelegt.
Eine Referenz auf die Wert-Komponente diese Elements, die zunächst undefiniert ist, wird von der Index-Operatorfunktion zurückgegeben.
Anschließend wird dieser Komponente der Wert `24.50` zugewiesen.

- **Einige innere Datentypen**
  - ◇ Innerhalb des Klassen-Templates `map` sind – wie in allen Container-Klassen – mehrere Datentypen definiert.
    Es handelt sich bei ihnen um `public`-Komponenten
  - ◇ Einige dieser Datentypen sind :

```
template <class Key, class T,
          class Compare = less<Key>,
          class Allocator = allocator<pair<const Key,T> > >
  class map
  { public :
      typedef Key                    key_type;
      typedef T                      mapped_type;
      typedef pair<const Key, T>     value_type;
      // ...
  };
```

- **Möglichkeiten zur Erzeugung von Schlüssel-/Werte-Paaren (z.B. als Parameter für `insert(...)`)**
  - ◇ mit **`pair<>`** : z.B. **`pair<const string, double>("Fahrrad", 259.00);`**
    Vereinfachung : Definition eines eigenen Typnamens für den `pair`-Typ.
    z.B. `typedef pair<const string, double>` **`StrDoubPair;`**
    `StrDoubPair("Fahrrad", 259.00);`
  - ◇ mit **`value_type`** : z.B. **`map<string, double>::value_type("Fahrrad", 259.00);`**
  - ◇ mit **`make_pair()`** : z.B. **`make_pair(string("Fahrrad"), 259.00);`**
    Achtung : Die Typen von `pair` müssen eindeutig aus den aktuellen Parametern von `make_pair()` erkennbar sein

10.15 (STL) von C++ : Klassen-Template `multimap (1)`

- **Eigenschaften**
  - ◇ Implementierung von **Multimaps** (Multimap-Containern)
    Ein Multimap (-Container) entspricht einer Map mit dem Unterschied, dass mehrere Elemente mit dem gleichen Schlüssel enthalten sein können ("***one-to-many mapping***")
  - ◇ Die **Definition** des Klassen-Templates `multimap<>` befindet sich ebenfalls in der **Headerdatei `<map>`**

```
template <class Key, class T,
          class Compare = less<Key>,
          class Allocator = allocator<pair<const Key,T> > >
  class multimap
  { // ...    };
```

- **Konstruktoren**

`explicit multimap(const Compare& = Compare(), const Allocator& = Allocator());`	Erzeugung einer leeren Multimap
`template <class InputIterator> multimap(InputIterator first, InputIterator last, const Compare& =Compare(), const Allocator& = Allocator());`	Erzeugung einer zunächst leeren Multimap, in die dann die zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elemente eines anderen Containers eingefügt werden. Diese Elemente müssen vom Typ `pair<const Key, T>` sein.
`multimap(const multimap<Key, T, Compare, Allocator>& x);`	Copy-Konstruktor

- **Anmerkungen zu den zusätzlichen spezifischen Memberfunktionen**
  - ◇ Die Indexoperatorfunktion **`operator[]()`** ist für das Klassen-Template `multimap` **nicht definiert**.
    Da mehrere Elemente mit dem gleichen Schlüssel vorkommen können, kann dieser nicht zur Elementauswahl verwendet werden. → Multimaps eignen sich nicht als assoziative Arrays.
  - ◇ Im übrigen besitzt das Klassen-Template `multimap` die **gleichen Memberfunktionen** wie **`map`**.
    Sie **unterscheiden** sich lediglich durch die **Auswirkungen** des möglichen **Mehrfachenthaltenseins** von Schlüsseln.
  - ◇ Im wesentlichen bestehen die **folgenden Unterschiede** :

▻ **`erase`**`(k)`	Löschen aller Elemente, die dem Schlüssel `k` besitzen.
▻ **`insert`**`(x)`	Der Rückgabewert ist vom Typ `iterator` (statt `pair<...>`) Einfügen des Schlüssel-/Wert-Paares `x` als neues Element auch dann, wenn bereits ein Element mit dem Schlüssel `x.first` in der Multimap enthalten ist. Es wird immer die Einfügeposition zurückgegeben.
▻ **`insert`**`(pos, x)`	Einfügen des Schlüssel-/Wert-Paares `x` als neues Element auch dann, wenn bereits ein Element mit dem Schlüssel `x.first` in der Multimap enthalten ist. Der Rückgabewert ist immer die Einfügeposition
▻ **`insert`**`(first, last)`	Einfügen immer aller Elemente aus dem Bereich `first` (einschl.) bis `last` (ausschl.), auch die, deren Schlüssel bereits vorhanden sind.
▻ **`count`**`(k)`	Rückgabe der Anzahl Elemente, die den Schlüssel `k` besitzen.
▻ **`find`**`(k)`	Suchen des ersten Elements, dessen Schlüssel gleich `k` ist.

(STL) von C++ : Klassen-Template multimap (2)

- **Einfaches Demonstrationsprogramm `multimapdem`**

```
// C++-Quelldatei multimapdem_m.cpp --> Programm multimapdem
// Einfaches Demo-Programm zum Klassen-Template multimap<> der STL

#include <map>
#include <string>
#include <iostream>
#include <utility>
#include <iomanip>
using namespace std;

typedef multimap<string, string>   StrStrMMap;
typedef pair<const string, string> StringPaar;


int main(void)
{
  StrStrMMap dict;
  dict.insert(make_pair(string("clever"), string("klug")));
  dict.insert(pair<const string, string>("clever", "gewandt"));
  dict.insert(StrStrMMap::value_type("clever", "raffiniert"));
  dict.insert(StringPaar("smart", "klug"));
  dict.insert(StringPaar("smart", "gewandt"));
  dict.insert(StringPaar("smart", "elegant"));
  dict.insert(StringPaar("wise", "klug"));
  dict.insert(StringPaar("wise", "erfahren"));
  dict.insert(StringPaar("strange", "fremd"));
  dict.insert(StringPaar("strange", "seltsam"));
  dict.insert(StringPaar("odd", "seltsam"));
  dict.insert(StringPaar("odd", "ungerade"));
  dict.insert(StringPaar("quick", "schnell"));
  dict.insert(StringPaar("quick", "gewandt"));
  dict.insert(StringPaar("car", "Auto"));
  dict.insert(StringPaar("again", "nochmals"));
  dict.insert(StringPaar("ship", "Schiff"));

  // Ausgabe aller Eintraege (Schluessel-Werte-Paare)
  cout << "Enthalten sind " << dict.size() << " Eintraege\n" << left << endl;
  StrStrMMap::iterator pos;
  for (pos=dict.begin(); pos!=dict.end(); ++pos)
    cout << "english : " << setw(15) << pos->first
         << "deutsch : " << setw(15) << pos->second << endl;
  cout << right << endl;;

  // Ausgabe aller Werte zu einenm bestimmten Schluessel
  string wort("clever");
  cout << wort << " : " << dict.count(wort) << " Eintraege" << endl;
  for (pos=dict.lower_bound(wort);
       pos!=dict.upper_bound(wort) ; ++pos)
    cout << "   " << pos->second << endl;
  cout << endl;

  // Ausgabe aller Schluessel zu einem bestimmten Wert
  wort="klug";
  cout << wort << " : " <<endl;
  for (pos=dict.begin(); pos!=dict.end(); ++pos)
    if (pos->second==wort)
      cout << "   " << pos->first << endl;
  cout << endl;
```

Standard-Template-Library (STL) von C++ : Klassen-Template `multimap (3)`

- **Ausgabe des Demonstrationsprogramms `multimapdem`**

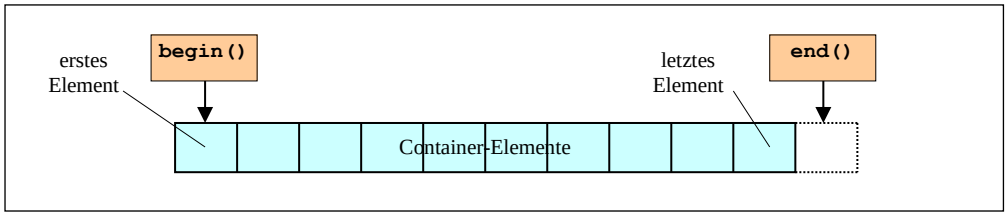
```
Enthalten sind 17 Eintraege

english : again          deutsch : nochmals
english : car            deutsch : Auto
english : clever         deutsch : klug
english : clever         deutsch : gewandt
english : clever         deutsch : raffiniert
english : odd            deutsch : seltsam
english : odd            deutsch : ungerade
english : quick          deutsch : schnell
english : quick          deutsch : gewandt
english : ship           deutsch : Schiff
english : smart          deutsch : klug
english : smart          deutsch : gewandt
english : smart          deutsch : elegant
english : strange        deutsch : fremd
english : strange        deutsch : seltsam
english : wise           deutsch : klug
english : wise           deutsch : erfahren

clever : 3 Eintraege
   klug
   gewandt
   raffiniert

klug :
   clever
   smart
   wise
```

10.16 STL von C++ : Iteratoren (1)

- **Einführung**
  - ◇ **Iteratoren** sind Objekte, die einen Mechanismus zum **Durchlaufen** von **Container-Objekten** zur Verfügung stellen.
  - ◇ Die in der STL implementierten Iteratoren ermöglichen einen **zeigerähnlichen Zugriff** zu den einzelnen **Elementen** eines Containers. Analog zu einem Zeiger, der die Position eines Array-Elements angibt, **repräsentiert** ein derartiges **Iterator-Objekt** die **Position** eines **Container-Elements**.
    Solche Iteratoren können somit als **Verallgemeinerung von Zeigern** aufgefasst werden.
  - ◇ Da Iteratoren **Zustandsinformationen** des jeweiligen **Containers**, auf dem sie arbeiten, auswerten und die **Organisationsform** seiner Elemente berücksichtigen müssen, ist eine **Iterator-Klasse** immer an eine **bestimmte Container-Klasse gebunden**. D. h. zu jeder Container-Klasse gehört jeweils eine **spezifische** passende **Iterator-Klasse**.
    Typischerweise ist diese als **innere Klasse** der Containerklasse definiert. Bei den Iteratoren der STL ist das immer der Fall.
  - ◇ Alle **Iterator-Klassen** der STL stellen – unabhängig von der Container-Klasse, an die sie jeweils gebunden sind und unabhängig von ihrer eigenen Implementierung – eine **einheitliche Schnittstelle** zur Verfügung.
    Diese wird durch **Operator-Funktionen** gebildet, die die für Zeiger anwendbaren Operationen implementieren (wie Dereferenzierung, Inkrementierung usw.).
  - ◇ Die **Container-Klassen** ihrerseits stellen eine **einheitliche Schnittstelle** zur **Bereitstellung** und **Verwendung** der **Iteratoren** bereit:
    - ▻ In **jeder Container-Klasse** – außer den Container-Adaptern – sind die implementierungsabhängigen `public`-Typen **`iterator`** und **`const_iterator`** definiert. Container-Elemente, die von Iterator-Objekten des Typs `const_iterator` referiert werden, können nicht geändert werden.

```
template < ... >
  class ...
  {
    public :
       // ...
       typedef implementation defined iterator;
       typedef implementation defined const_iterator;
       // ...
  };
```

    - ▻ **Jede Container-Klasse** – außer den Container-Adaptern – stellt **Memberfunktionen** zur Ermittlung des **Iteratorbereichs** eines Container-Objekts zu Verfügung. :
      - **`begin()`** liefert den Iterator, der auf das erste Element im Container zeigt
      - **`end()`** liefert einen Iterator, der auf die Position unmittelbar nach dem letzten Element zeigt.



      Der Bereich **`[ begin(), end() )`** bildet also ein **halboffenes Intervall**, über das durch alle Container-Elemente iteriert werden kann.
      Ist `begin()==end()`, ist das Intervall leer, d.h. der Container enthält keine Elemente.

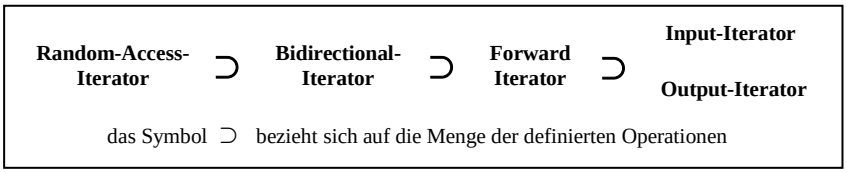
      Beide Funktionen existieren jeweils in **zwei überladenen Formen**, die eine liefert ein Iterator-Objekt vom Typ `iterator`, die andere ein Iterator-Objekt vom Typ `const_iterator`.

  - ◇ Diese einheitlichen Schnittstellen ermöglichen es, dass die **Objekte** der **unterschiedlichen Container-Klassen**, die **unterschiedliche Datenstrukturen** implementieren, **gleichartig bearbeitet** werden können.

Standard-Template-Library (STL) von C++ : Iteratoren (2)

- **Überblick über die Iterator-Bibliothek**
  - ◇ Die in der Iterator-Bibliothek enthaltenen Definitionen **iterator-spezifischer Datentypen** und **Funktionen** der STL sind in der Headerdatei **`<iterator>`** zusammengefasst.
  - ◇ Im wesentlich handelt es sich hierbei um
    - ▻ **grundlegende Datentypen**, die im Container-Teil der STL für die Definition der container-spezifischen Iterator-Klassen verwendet werden, sowie zur Definition eigener Iterator-Klassen eingesetzt werden können.
      U.a. ist hier auch das als **Iterator-Basisklasse** dienende Klassen-Template **`iterator`** definiert
    - ▻ **freie Iterator-Funktionen** (Funktions-Templates), die bestimmte Operationen auf Iterator-Objekte ausführen.
    - ▻ **Iterator-Adapter**, durch die Iterator-Klassen mit spezifischen Eigenschaften vordefiniert werden.
      Sie erweitern die Anwendungsmöglichkeiten der Algorithmen der STL erheblich.
      Hierbei handelt es sich um
      - **Reverse-Iteratoren**, mit denen die Durchlaufrichtung von Container umgekehrt wird.
      - **Insert-Iteratoren**, mit denen sich die Kopier-Algorithmen der STL zum Einfügen (statt zum normalerweise stattfindenden Überschreiben) einsetzen lassen.
      - **Stream-Iteratoren** und **Streambuffer-Iteratoren**, mit denen zu Eingabe- bzw. Ausgabe-Streams wie zu Containern zugegriffen werden kann.
        Das Lesen und Schreiben von Daten lässt sich damit unter Verwendung von Iteratoren durchführen.
  - ◇ Zur Arbeit mit Iteratoren muss die Headerdatei `<iterator>` in der Regel aber nicht explizit eingebunden werden, da sie von allen Headerdateien für Container und der Headerdatei für Algorithmen bereits eingebunden wird.

- **Iterator-Kategorien**
  - ◇ In Abhängigkeit von den auf Iteratoren **anwendbaren Operationen** werden **fünf Iterator-Kategorien** unterschieden :
    - ▻ Input-Iteratoren
    - ▻ Output-Iteratoren
    - ▻ Forward-Iteratoren
    - ▻ Bidirectional-Iteratoren
    - ▻ Random-Access-Iteratoren



  - ◇ **Input-Iteratoren** (InpIter, *input iterators*)
    Sie erlauben lediglich einen **lesenden Zugriff** auf das durch sie referierte Element des Containers, sowie ein Durchlaufen
    des Containers (der Sequenz) in Vorwärtsrichtung (Implementierung des Increment-Operators). Dabei ist mit einem Input-Iterator nur ein einmaliger Durchlauf möglich, d. h. er kann nur für *one-pass algorithms* eingesetzt werden
    Außerdem lassen sich auf Input-Iteratoren der Zuweisungs-Operator, sowie der Gleichheits- und der Ungleichheits-Operator anwenden.
  - ◇ **Output-Iteratoren** (OutpIter, *output iterators*)
    Sie erlauben lediglich einen **schreibenden Zugriff** auf das durch sie referierte Container-Element, sowie ein Durchlaufen des Containers (der Sequenz) in Vorwärtsrichtung. Auch ein Output-Operator gestattet nur einen einmaligen Durchlauf.
    Der Zuweisungs-Operator lässt sich auf sie anwenden, nicht jedoch der Gleichheits- und der Ungleichheits-Operator
  - ◇ **Forward-Iteratoren** (ForwIter, *forward iterators*)
    Sie kombinieren die Fähigkeiten von Input- und Output-Operatoren. Zusätzlich kann mit dem gleichen Iterator eine Elemente-Menge auch mehrfach durchlaufen werden, d.h. diese Iteratoren unterstützen *multi-pass algorithms*.
  - ◇ **Bidirectional-Iteratoren** (BidirIter, *bidirectional iterators*)
    Sie besitzen alle Fähigkeiten der Vorwärts-Iteratoren. Zusätzlich ermöglichen sie ein Durchlaufen des Containers in Rückwärtsrichtung (Implementierung des Decrement-Operators)

- ◇ **Random-Access-Iteratoren** (RanAccIter, *random access iterators*)
  Sie besitzen alle Fähigkeiten der Bidirectional-Iteratoren. Zusätzlich erlauben sie einen direkten wahlfreien Zugriff zu jedem Element des Containers. Hierzu implementieren sie den Index-Operator sowie eine "Adress-Arithmetik". Außerdem können auf diese Iteratoren auch die Vergleichs-Operationen `>, >=, <, <=` angewendet werden. Damit stellen sie die **volle gleiche Funktionalität** wie normale typgebundene **Pointer** zu Verfügung.

- **Überblick über die Iterator-Operationen**
  - ◇ In der folgenden Übersicht
    - sind `p` und `q` Iteratoren
    - bedeutet `X` : die Operation steht für die jeweilige Iterator-Kategorie zur Verfügung

Operation	Bemerkung	InpIter	OutpIter	ForwIter	BidirIter	RanAccIter
Kopieren						
`p=q`	Zuweisung	X	X	X	X	X
Dereferenzieren						
`x=*p`	Lesen (Rvalue)	X		X	X	X
`x=p->k`	`x=(*p).k`	X		X	X	X
`*p=x`	Schreiben (Lvalue)		X	X	X	X
Inkrementieren						
`++p`	Ergebnis= neuer Wert	X	X	X	X	X
`p++`	Ergebnis= alter Wert	X	X	X	X	X
Dekrementieren						
`--p`	Ergebnis= neuer Wert				X	X
`p--`	Ergebnis= alter Wert				X	X
Vergleichen						
`p==q`	gleich	X		X	X	X
`p!=q`	ungleich	X		X	X	X
`p<q`	kleiner					X
`p<=q`	kleiner gleich					X
`p>q`	größer					X
`p>=q`	größer gleich					X
Direktzugriff						
`p=p+n`	`n` Positionen nach `p`					X
`p+=n`	`n` Positionen nach `p`					X
`p=p-n`	`n` Positionen vor `p`					X
`p-=n`	`n` Positionen vor `p`					X
`x=p[n]`	`x=*(p+n)`					X
`p[n]=x`	`*(p+n)=x`					X
Iteratordistanz						
`n=p-q`	Entfernung in Anz. Positionen					X

  - ◇ **Anmerkung zum Inkrementieren/Dekrementieren** :
    Da der **Pre-Increment**- und der **Pre-Decrement-Operator** den aktuellen – neuen – Wert (und nicht den alten Wert) zurückliefern, lassen sie sich wesentlich **effizienter** als die entsprechenden Post-Operatoren **implementieren**.
    Man sollte sie daher gegenüber diesen bevorzugt einsetzen : **`++p`** ist **effizienter** als `p++`.

- **Iterator-Kategorien der STL-Container-Klassen**
  - ◇ Die STL-Containerklassen **`vector`** und **`deque`** unterstützen **Random-Acces-Iteratoren**
  - ◇ Die STL-Containerklassen **`list`, `set`, `multiset`, `map`** und **`multimap`** unterstützen **Bidirectional-Iteratoren**

Standard-Template-Library (STL) von C++ : Iteratoren (4)

- **Einfaches Demonstrationsprogramm zu Random-Access-Iteratoren `randiterdem`**

```
// C++-Quelldatei randiterdem_m.cpp --> Programm randiterdem
// Einfaches Demo-Programm zu Random-Access-Iteratoren der STL

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


int main(void)
{
  vector<int> zahlen;
  int i;
  vector<int>::iterator pos;

  for (i=1; i<=10; i++)
    zahlen.push_back(i);

  cout << "\nAnzahl : " << zahlen.end()- zahlen.begin() << endl;

  cout << "Inhalt :\n";
  cout << "Verwendung *-Operator\n";
  for (pos=zahlen.begin(); pos<zahlen.end(); ++pos)
    cout <<*pos << "  ";
  cout << endl;

  cout << "Zugriff ueber Index-Operator\n";
  for (i=0; i<zahlen.size(); i++)
    cout << zahlen.begin()[i] << "  ";
  cout << endl;

  cout << "jedes 2. Element\n";
  for (pos=zahlen.begin(); pos<zahlen.end(); pos+=2)
    cout <<*pos << "  ";
  cout << endl;


  return 0;
}
```

- **Ausgabe des Demonstrationsprogramms `randiterdem`**

```
Anzahl : 10
Inhalt :
Verwendung *-Operator
1  2  3  4  5  6  7  8  9  10
Zugriff ueber Index-Operator
1  2  3  4  5  6  7  8  9  10
jedes 2. Element
1  3  5  7  9
```

Standard-Template-Library (STL) von C++ : Iteratoren (5)

- **Freie Iterator-Funktionen**
  - ◇ Die STL stellt **zwei freie Funktionen** zur Verfügung, mit denen einige nur für Random-Access-Iteratoren definierte Operationen auch für Input-Iteratoren (und damit auch für Bidirectional- und Forward-Iteratoren) ermöglicht werden.
  - ◇ Die beiden als **Funktions-Templates** in der Headerdatei **<iterator>** definierten Funktionen werden nachfolgend in **vereinfachter Darstellung** deklariert.

◇ **`void advance(InputIterator& pos, Dist n);`**

- ▻ **Weitersetzen** des (Input-)**Iterators** `pos` um `n` Elemente
- ▻ Für Bidirectional- und Random-Access-Iteratoren darf `n` auch negativ sein.
- ▻ Die Typen `InputIterator` und `Dist` sind tatsächlich Template-Parameter.
  Da alle Iteratoren, außer den Output-Iteratoren, die Funktionalität von Input-Iteratoren implementieren, kann die aktuelle Iteratorklasse eine Input-, Forward-, Bidirectional- oder Random-Access-Iterator-Klasse sein.
  Der den formalen Typ-Parameter `Dist` ersetzende aktuelle Typ muss ein für die jeweilige aktuelle Iteratorklasse anwendbarer Ganzzahltyp zur Darstellung von "Iteratorabständen" sein.

◇ **`Dist distance(InputIterator first, InputIterator last);`**

- ▻ **Ermittlung des Abstands** zwischen den (Input-)Iteratoren `first` und `last`.
  = Anzahl der Increments bzw. Decrements, um von `first` nach `last` zu gelangen.
- ▻ Beide Iteratoren müssen zum gleichen Container gehören.
- ▻ Der Iterator `last` muss vom Iterator `first` aus erreichbar sein.
  Das bedeutet, dass bei allen Iteratoren, die keine Random-Access-Iteratoren sind, `last` hinter `first` liegen muss (d. h. über Increments erreichbar sein muss).
- ▻ Der Typ `InputIterator` ist tatsächlich Template-Parameter.
  Aktuell kann es sich um eine Iteratorklasse der Kategorien Input-Iterator, Forward-Iterator, Bidirectional-Iterator oder Random-Access-Iterator handeln, da alle Iteratoren dieser Klassen die Funktionalität von Input-Iteratoren implementieren.
- ▻ Der Rückgabetyp `Dist` ist der für die jeweilige aktuelle Iteratorklasse definierte Ganzzahltyp zur Darstellung von "Iteratorabständen" ("Abstands-Typ" der Iteratorklasse).

◇ Eine **weitere freie Iterator-Funktion** ist in der Algorithmen-Bibliothek (Headerdatei **<algorithm>**) definiert.
Es handelt sich um eine Funktion, mit der die von zwei Iteratoren referierten **Elemente vertauscht** werden können.
Auch diese Funktion ist als Funktions-Template definiert.
Ihre Deklaration wird nachfolgend ebenfalls in vereinfachter Darstellung angegeben.

◇ **`void iter_swap(ForwardIterator1 a, ForwardIterator2 b);`**

- ▻ **Vertauschen** der durch die Iteratoren `a` und `b` **referierten Elemente**.
- ▻ Die Typen `ForwardIterator1` und `ForwardIterator2` sind tatsächlich Template-Parameter.
  Aktuell kann für sie jede Iteratorklasse der Kategorien Forward-Iterator, Bididirectional-Iterator oder Random-Access-Iterator verwendet werden.
- ▻ Die beiden Iteratorklassen müssen nicht gleich sein. Die referierten Elemente müssen sich lediglich gegenseitig zuweisen lassen.

10.17 Standard-Template-Library (STL) von C++ : Reverse-Iteratoren

- **Allgemeines**

  ◇ In der STL ist – in der Headerdatei **`<iterator>`** – der **Iterator-Adapter `reverse_iterator`** als Klassen-Template definiert. Template-Parameter ist eine Iterator-Klasse.
  Dieser Iterator-Adapter erzeugt **Reverse-Iteratoren**

  ◇ Reverse-Iteratoren sind Iteratoren, bei denen die **Durchlaufrichtung** durch einen Container **umgekehrt** ist, d. h. ein **Inkrementieren** bewegt den Iterator zum **vorhergehenden Element**.
  Mit ihrer Hilfe kann bei allen Algorithmen die Verarbeitungsreihenfolge der Elemente umgekehrt werden, ohne dass entsprechende neue Algorithmen implementiert werden müssen.

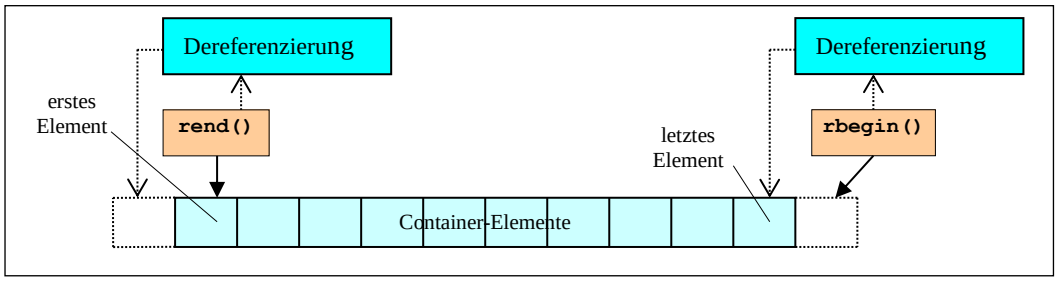
  ◇ Reverse-Iteratoren können nur zu Bidirectional- (und damit auch zu Random-Access-) Iteratoren gebildet werden.
  Da **alle in der STL definierten Container-Klassen** Iteratoren dieser Kategorien zu unterstützen, lassen sich für sie auch Reverse-Iteratoren erzeugen.

  ▻ In **jeder Container-Klasse** – außer den Container-Adaptern – sind deshalb auch die implementierungsabhängigen `public`-Typen **`reverse_iterator`** und **`const_reverse_iterator`** definiert.
  Container-Elemente, die von Reverse-Iterator-Objekten des Typs `const_reverse_iterator` referiert werden, können nicht geändert werden.

```
template < ... >
  class ...
  {
    public :
       // ...
       typedef std::reverse_iterator<iterator> reverse_iterator;
       typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
       // ...
  };
```

  ▻ Außerdem stellt **jede Container-Klasse** – außer den Container-Adaptern – auch **Memberfunktionen** zur Ermittlung des **Reverse-Iteratorbereichs** eines Container-Objekts zu Verfügung. :

  - **`rbegin()`** liefert den Reverse-Iterator, der auf die Position unmittelbar nach dem letzten Element zeigt
    Dies ist der in einen Reverse-Iterator umgewandelte Funktionswert von `end()`.
  - **`rend()`** liefert einen Reverse-Iterator, der auf das erste Element im Container zeigt.
    Dies ist der in einen Reverse-Iterator umgewandelte Funktionswert von `begin()`.



  Auch diese **beiden Funktionen** existieren jeweils in **zwei überladenen Formen**, die eine liefert ein Reverse-Iterator-Objekt vom Typ `reverse_iterator`, die andere ein Reverse-Iterator-Objekt vom Typ `const_reverse_iterator`.

  ◇ Bei der **Erzeugung eines Reverse-Iterators** aus einem – normalen – Iterator **bleibt** die **Iterator-Position gleich**.
  Um beim Durchlaufen von Containers-Bereichen mit Reverse-Iteratoren die **gleiche Semantik** wie mit – normalen – Iteratoren verwenden zu können (halboffenes Intervall, letzter Iterator zeigt auf nicht mehr vorhandenes Element), wird bei der **Dereferenzierung eines Reverse-Iterators** das **Element** zurückgeliefert, das sich an der **Position unmittelbar vor der eigentlich referierten Position** befindet.

Standard-Template-Library (STL) von C++ : Reverse-Iteratoren (2)

- **Definition des Klassen-Templates `reverse_iterator` (unvollständiger Auszug)**

```
template <class Iterator>
  class reverse_iterator : public iterator< ... >
{
  protected :
    Iterator current;
  public :
    // Typ-Definitionen

    reverse_iterator();
    explicit reverse_iterator(Iterator x);
    template <class U> reverse_iterator(const reverse_iterator<U>& u);

    Iterator base() const;

    // Operatorfunktionen zum Element-Zugriff und zur Iterator-Aenderung

};

// freie Vergleichs-Operatorfunktionen
```

- **Konstruktoren**
  - ▻ Konstruktor zur Erzeugung eines **nichtinitialisierten** Reverse-Iterator-Objekts.
  - ▻ Konstruktor zur Erzeugung eines Reverse-Iterator-Objekts, das **mit einem** – normalen – **Iterator-Objekt initialisiert** wird
  - ▻ Copy-Konstruktor

- **Operationen mit Reverse-Iteratoren**
  - ◇ Für Reverse-Iteratoren sind die **gleichen Operationen** wie für die entsprechenden – normalen – Iteratoren definiert.
  - ◇ Allerdings sind **einige Wirkungs-Unterschiede** zu beachten :
    - ▻ Ein **Inkrementieren** des Reverse-Iterators wird in ein **Dekrementieren umgesetzt** (statt `++p` → `--p`) und umgekehrt (`--p` → `++p`).
    - ▻ Eine Veränderung eines Reverse-Iterators um einen **positiven Abstand** wird in eine Veränderung um einen **negativen Abstand umgesetzt** und umgekehrt.
    - ▻ Die **Dereferenzierung** eines Inverse-Iterators führt zur Rückgabe des **unmittelbar davor befindlichen Elements**
    - ▻ Bei **Vergleichsoperationen** werden die **Reihenfolgen der Operanden vertauscht** :
      `x>y` bzw **`x>=y`** führt zur Auswertung von **`y.current>x.current`** bzw **`y.current>=x.current`**
      `x<y` bzw **`x<=y`** führt zur Auswertung von **`y.current<x.current`** bzw **`y.current<=x.current`**
  - ◇ Zur **Rückwandlung eines Reverse-Iterators in einen – normalen – Iterator** existiert die Memberfunktion

    `Iterator base() const;`

    Sie liefert den – normalen – Iterator, der auf die gleiche Position wie der Reverse-Operator verweist.

    Wenn **`rpos`** ein Reverse-Iterator ist, kann somit mittels **`*rpos.base()`** auf das **tatsächlich referierte** (und nicht auf das davor befindliche) **Element zugegriffen** werden.

10.18 STL von C++ : Stream-Iteratoren (1)

- **Allgemeines**
  - ◇ **Iteratoren** setzen voraus, dass die **Elemente** der von ihnen durchlaufbaren Datenmenge in einer **Reihenfolge** (Sequenz) **angeordnet** sind
  - ◇ **Stream-Iteratoren** sind **Iterator-Adapter**, die es ermöglichen, dass zu Streams, die ja auch aus einer Folge von Daten bestehen, mit der **gleichen Schnittstelle** wie zu Containern zugegriffen werden kann.
    Dadurch lassen sich zahlreiche **Algorithmen der STL** (z.B. zum Kopieren) auch sinnvoll **direkt auf Streams** anwenden, d.h. Algorithmen können Eingabedaten statt aus Containern aus Streams beziehen und Ausgabedaten statt in Container in Streams ausgeben.
  - ◇ **Ostream-Iteratoren** dienen zum **Ausgeben** von Elementen in Streams.
    Istream-Iteratoren dienen zum **Einlesen** von Elementen aus Streams.
  - ◇ Neben Stream-Iteratoren gibt es auch **Streambuffer-Iteratoren**. Diese ermöglichen einen **zeichenweisen** (und nicht wertweisen) Zugriff zu den von Streams verwendeten Streambuffern.

- **Ostream-Iteratoren**
  - ◇ Ostream-Iteratoren gehören zur Kategorie der **Output-Iteratoren**.
    Statt an ein Container-Objekt sind sie an ein **Ausgabestream-Objekt gebunden**.
    Dieses muss bei der Erzeugung eines Ostream-Iterator-Objekts bereitgestellt werden.
    Ein Ostream-Iterator referiert immer die nächste Ausgabeposition im Stream.
    Die **wesentliche Operation** – schreibender Zugriff zum referierten Element – führt zur **Ausgabe** des dem Element **zugewiesenen Werts** in den **Ausgabestream**.
  - ◇ Zur Erzeugung von Ostream-Iteratorklassen definiert die STL das **Klassen-Template `ostream_iterator`**
    Template-Parameter sind :
    - der **Typ `T`** der referierten (d.h. auszugebenden) **Daten**.
    - Zeichentyp **`charT`** (Default : `char`)
    - Zeicheneigenschaften-Typ `traits` (Default : `char_traits<charT>`).
  - ◇ **Konstruktoren** (Vereinfachte Darstellung für die Default-Template-Parameter):

`ostream_iterator``(ostream& s);`	Erzeugung eines Ostream-Iterators für das `ostream`-Objekt `s`
`ostream_iterator``(ostream& s, const char* del);`	Erzeugung eines Ostream-Iterators für das `ostream`-Objekt `s`, der nach jeder Ausgabe eines Elements den C-String `del` ausgibt
`ostream_iterator``(const ostream_iterator<T>& x);`	Copy-Konstruktor

  - ◇ **Zuweisungs-Operatorfunktion** (Vereinfachte Darstellung für die Default-Template-Parameter):

`ostream_iterator<T>&` **`operator=`**`(const T& val);`	Ausgabe des Werts `val` in den Ausgabestream

    Statt durch eine Zuweisung an den dereferenzierten Iterator kann ein Wert auch durch direkte Zuweisung an den Iterator ausgegeben werden, d.h. **`*p=val`** `und` **`p=val`** sind **wirkungsgleich**. (`p` sei ein Ostream-Iterator)
  - ◇ **Weitere Operationen**
    - ▻ Die beiden **Increment-Operatorfunktionen** sind zwar definiert, sie bewirken aber nichts und geben lediglich eine Referenz auf das aktuelle Ostream-Iterator-Objekt zurück.
    - ▻ Genaugenommen gibt auch die **Dereferenzierungs-Operatorfunktion** lediglich eine Referenz auf das aktuelle Objekt zurück. Die Anwendung des Zuweisungs-Operators auf dieses führt dann erst zur Ausgabe eines Werts.

Standard-Template-Library (STL) von C++ : Stream-Iteratoren (2)

- **Istream-Iteratoren**

  ◇ Istream-Iteratoren sind **Input-Iteratoren**.
  Statt an ein Container-Objekt sind sie an ein **Eingabestream-Objekt** gebunden.
  Dieses muss bei der Erzeugung eines Istream-Iterator-Objekts dem Konstruktor übergeben werden.
  Ein Istream-Iterator **referier**t immer die **aktuelle Eingabeposition im Stream**. Beim **Erzeugen** eines Istream-Iterators sowie bei jedem Incrementieren des Iterators wird ein **Element** aus dem Stream **gelesen**.
  Jeder **dereferenzierende Zugriff zum Iterator**, der **nur lesend** zulässig ist, liefert das **zuletzt eingelesene Element**.

  ◇ Die STL definiert das **Klassen-Template `istream_iterator`** zur Erzeugung von Istream-Iteratorklassen
  Template-Parameter sind : – der **Typ `T`** der referierten (d. h. auszugebenden) **Daten**.
  – Zeichentyp **`charT`** (Default : `char`)
  – Zeicheneigenschaften-Typ **`traits`** (Default : `char_traits<charT>`).
  – Abstands-Typ **`Dist`** (Default : `ptrdiff_t`, ein vorzeichenloser Ganzzahltyp)

  ◇ **Konstruktoren** (Vereinfachte Darstellung für die Default-Template-Parameter):

`istream_iterator``();`	Erzeugung eines End-of-Stream-Iterators (referiert `EOF`)
`istream_iterator``(istream& s);`	Erzeugung eines Istream-Iterators für das `istream`-Objekt `s`, Einlesen des ersten Elements aus dem Stream
`istream_iterator``(const istream_iterator<T>& x);`	Copy-Konstruktor

  ◇ **Member-Operatorfunktionen** (Vereinfachte Darstellung für die Default-Template-Parameter):

`const T&` **`operator*`**`() const;`	Rückgabe des zuletzt gelesenen Elements
`const T*` **`operator->`**`() const;`	Rückgabe eines Pointers auf das zuletzt gelesene Element, über diesen kann ein Zugriff zu seinen Komponenten erfolgen
`istream_iterator<T>&` **`operator++`**`();`	Einlesen des nächsten Elements aus dem Stream, Rückgabe des fortgeschalteten neuen Iterators
`istream_iterator<T>` **`operator++`**`(int);`	Einlesen des nächsten Elements aus dem Stream, Rückgabe des alten Iterators

  Falls beim **Inkrementieren** des Istream-Iterators das **Stream-Objekt** in einen **Fehlerzustand** gelangt (z. B. auch bei Erreichen von `EOF`), wird der fortgeschaltete neue Iterator zu einem **End-of-Stream-Iterator**.

  ◇ **freie Vergleichsfunktionen**
  - ▻ Die Operatorfunktionen für den **Vergleich auf Gleichheit** (**`operator==()`**) und **Ungleichheit** (**`operator!=()`**) sind als **freie Funktions-Templates** definiert
  - ▻ Zwei Istream-Iteratoren sind dann **gleich**,
    - wenn sie beide kein gültiges Element mehr referieren (d. h. End-of-Stream-Iteratoren sind)
    - oder wenn beide keine End-of-Stream-Iteratoren sind und zum gleichen `istream`-Objekt gehören
  - ▻ **Anmerkung zur Überprüfung auf Streamende (`EOF`)** :
    Nach jedem Inkrementieren eines Istream-Iterators sollte durch Vergleich des Iterators mit einem End-of-Stream-Iterator der Fehlerzustand des Streams überprüft (und damit auch auf `EOF` geprüft) werden.

Standard-Template-Library (STL) von C++ : Stream-Iteratoren (3)

- **Einfaches Demonstrationsprogramm zu Stream-Iteratoren `streamiterdem`**

```
// C++-Quelldatei streamiterdem_m.cpp --> Programm streamiterdem
// Einfaches Demo-Programm zu Stream-Iteratoren der STL

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


int main(void)
{
  cout << "\nGeben Sie Integer-Werte ein : ";

  istream_iterator<int> input(cin); // Erzeugung Istream-Iterator
                                    // erster int-Wert wird aus Stream cin gelesen
  istream_iterator<int> eof;        // Erzeugung End-of-Stream-Iterator
  int i1, cnt=0, sum=0;

  while (input!=eof)
  {
    i1=*input;                  // Rückgabe letzter gelesener int-Wert
    cnt++;
    sum+=i1;
    ++input;                    // naechster int-Wert wird aus Stream cin gelesen
  }

  ostream_iterator<int> output(cout, "\n");
  cout << "\nAnz. eingegebener Werte : ";
  *output=cnt;                  // Ausgabe Anzahl Werte, alternativ : output=cnt;
  cout << "Summe aller Werte       : ";
  *output=sum;                  // Ausgabe Summe,        alternativ : output=sum;

  return 0;
}
```

- **Ein- und Ausgabe des Demonstrationsprogramms `streamiterdem` (Beispiel)**

```
Geben Sie Integer-Werte ein : 3 4 6 7 9 10
^Z
^Z

Anz. eingegebener Werte : 6
Summe aller Werte       : 39
```

10.19 STL von C++ : Insert-Iteratoren (1)

- **Allgemeines**
  - ◇ Bei "**normalen**" **Iteratoren** bewirkt eine **Zuweisung** an das durch den Iterator referierte Element, dass dieses **Element überschrieben** wird (d. h. einen neuen Wert bekommt).

    Beispiel :
    ```
    // first, last und res seien normale Iteratoren
    while (first != last) *res++ = *first++;
    ```
    Obige Schleife bewirkt ein Kopieren der Elemente des Bereichs `[first, last)` (Quellbereich) in die Elemente eines Bereichs der mit `res` beginnt (Zielbereich). Die Elemente des Zielbereichs müssen hierzu existieren.
  - ◇ Insert-Iteratoren sind **Iterator-Adapter**, die eine **Zuweisung** an den Iterator bzw an das referierte Element in ein **Einfügen** umsetzen.
    Wenn `res` in obigem Beispiel ein Insert-Iterator ist, werden Kopien der Elemente aus dem Quellereich als **neue** zusätzliche **Elemente** in den Zielbereich **eingefügt**.
- **Operationen**
  - ◇ Insert-Iteratoren gehören zur Kategorie der **Output-Iteratoren**.
    Das bedeutet, dass zu einem dereferenzierten Iterator (also dem referierten Element) nur **schreibend** zugegriffen werden kann.
  - ◇ Analog zu Ostream-Iteratoren kann ein derartiger schreibender Zugriff (**Zuweisung)** auch **direkt an** den **Iterator** erfolgen.
    D.h. auch für einen Insert-Iterator `p` gilt : **`*p=val`** und **`p=val`** sind **wirkungsgleich**

    Dies wird dadurch ermöglicht, dass die **Dereferenzierungs-**Operatorfunktion **`operator*()`** nicht das referierte Element sondern das Iterator-Objekt selbst `(*this)` zurückgibt, d. h. für den Insert-Iterator `p` gilt **`*p==p.`**
  - ◇ Die **Zuweisungs-**Operatorfunktion **`operator=()`** ist so definiert, dass sie eine Kopie des ihr als Parameter übergebenen Elements als **neues Element** in den **Container** des Zielbereichs **einfügt**.
    Das Einfügen erfolgt **unmittelbar vor** der Position, auf die der Iterator zeigt.
  - ◇ Die beiden **Increment**-Operatorfunktionen sind zwar auch definiert, aber wie bei Ostream-Iteratoren **bewirken** sie **nichts,** sondern geben lediglich das aktuelle Iterator-Objekt bzw eine Referenz hierauf zurück.
    Insert-Iteratoren können also ihre Position nicht verändern.
- **Insert-Iterator-Arten**
  - ◇ In Abhängigkeit von der Position, an der sie ein Einfügen bewirken, werden **drei Arten** von Insert-Iteratoren unterschieden :
    - ▻ **Front-Insert-Iteratoren (Front-Inserter)**
      Sie bewirken ein Einfügen am **Anfang** eines Containers (vor der ersten Position)
      Zum Einfügen rufen sie die Container-Funktion **`push_front(val)`** auf.
      Sie sind nur bei Containern, die diese Funktion zur Verfügung stellen, möglich : **Deques** und **Listen**
    - ▻ **Back-Insert-Iteratoren (Back-Inserter)**
      Sie bewirken ein Einfügen am **Ende** eines Containers (hinter der letzten Position).
      Zum Einfügen rufen sie die Container-Funktion **`push_back(val)`** auf.
      Da nur **Vektoren**, **Deques** und **Listen** diese Funktion zur Verfügung stellen, sind sie nur bei diesen Container-Arten möglich.
    - ▻ (positionierbare) **Insert-Iteratoren (Inserter)**
      Sie bewirken ein Einfügen vor einer **beliebigen gültigen** Container-**Position**.
      Die Einfügeposition ist bei der Erzeugung eines Insert-Iterators anzugeben.
      Zum Einfügen rufen sie die Container-Funktion **`insert(pos, val)`** auf.
      Da diese Funktion für alle STL-Container-Arten implementiert ist, lassen sich diese Insert-Iteratoren bei **jedem STL-Container** verwenden.
      Anmerkung: Bei assoziativen Containern hat die anzugebende Einfügeposition `pos` nur Hinweischarakter.

Standard-Template-Library (STL) von C++ : Insert-Iteratoren (2)

- **Definition der Insert-Iterator-Klassen**
  - ◇ Für die drei Iterator-Arten sind in der Headerdatei **<iterator>** **Klassen-Templates** definiert und implementiert :
    - ▻ `template <class Container> class front_insert_iterator;`
    - ▻ `template <class Container> class back_insert_iterator;`
    - ▻ `template <class Container> class insert_iterator;`
  - ◇ **Definition des Klassen-Templates** **`front_insert_iterator`**

```
template <class Container>
  class front_insert_iterator : public iterator<...>
  {
    protected :
      Container* container;
    public :
      typedef Container container_type;
      explicit front_insert_iterator(Container&);
      front_insert_iterator<Container>&
        operator=(typename Container::const_reference val);
      front_insert_iterator<Container>& operator*();
      front_insert_iterator<Container>& operator++();
      front_insert_iterator<Container>  operator++(int);
  };
```

  - ◇ Die **Definition** der beiden **anderen Klassen-Templates** erfolgt **analog**.
    Lediglich beim Klassen-Template **`insert_iterator`** besteht ein wesentlicher **Unterschied** :
    Dem Konstruktor ist die Einfügeposition als zusätzlicher Parameter (Typ : `Container::iterator`) zu übergeben.
    Diese wird in einer zusätzlichen `protected`-Datenkomponente gespeichert.

- **Erzeugung von Insert-Iteratoren**
  - ◇ Mittels des jeweiligen **Konstruktors.**
    Diesem ist das **Container-Objekt**, für den der Insert-Iterator erzeugt werden soll, als **Parameter** zu übergeben.
    Der Konstruktor für einen **positionierbaren Insert-Iterator** (Klassen-Template **`insert_iterator`**) benötigt als **zusätzlichen Parameter** einen Iterator, der die **Einfügeposition** angibt.

    Beispiel :
```
vector<int> imeng;
back_insert_iterator<vector<int> > backint(imeng);
insert_iterator<vector<int> > posint(imeng, imeng.begin()+4);
```

  - ◇ Als **Alternative** steht für jede Insert-Iterator-Art eine **freie Erzeugungsfunktion** in Form eines **Funktions-Templates** zur Verfügung. Diese erzeugen jeweils ein neues Insert-Iterator-Objekt und geben es als Funktionswert zurück.
    Diese Funktions-Templates sind ebenfalls in der Headerdatei **<iterator>** definiert.

```
template <class Container>
  front_insert_iterator<Container> front_inserter(Container& x);

template <class Container>
  back_insert_iterator<Container> back_inserter(Container& x);

template <class Container, class Iterator>
  insert_iterator<Container> inserter(Container& x, Iterator i);
```

    Beispiel : **`back_inserter (imeng)`**`=23;  // Einfügen neues Element mit Wert 23 am Ende von imeng`

Standard-Template-Library (STL) von C++ : Insert-Iteratoren (3)

- **Einfaches Demonstrationsprogramm zu Insert-Iteratoren `insiterdem`**

```
// C++-Quelldatei insiterdem_m.cpp --> Programm insiterdem
// Einfaches Demo-Programm zu Insert-Iteratoren der STL

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

template <class Container>
  void ausgabe(Container& cont)
  { Container::iterator di;
    ostream_iterator<Container::value_type> output(cout, "  ");
    cout << endl; //value_type liefert den Typ der Elemente im Container
    for (di=cont.begin(); di!=cont.end(); ++di)
      *output=*di;
    cout << endl;
  }

int main(void)
{ deque<int> imeng;
  deque<int>::iterator di;
  int i;

  for (i=0; i<8; ++i)
  { imeng.push_back(i+1);
    imeng.push_front(i+11);
  }
  cout << "\nInhalt imeng :";
  ausgabe(imeng);

  back_insert_iterator<deque<int> > backint(imeng);
  for (i=0; i<4; i++)
    *backint++=i+20;                // increment ++ ist wirkungslos und uberfluessig
  cout << "\nnach back_insert :";
  ausgabe(imeng);

  di=imeng.begin()+8;
  *di=30;
  cout << "\nnach Zuweisung ueber normalen Iterator (Element Nr. 9):";
  ausgabe(imeng);
  //Insert-Iterator als Inserter erzeugen
  *inserter(imeng, di) = 0;         // gleichbedeutend : inserter(imeng, di) = 0;
  cout << "\nnach Zuweisung ueber Insert-Iterator (Element Nr. 9):";
  ausgabe(imeng);
```

- **Ausgabe des Demonstrationsprogramms `insiterdem`**

```
Inhalt imeng :
18  17  16  15  14  13  12  11  1  2  3  4  5  6  7  8

nach back_insert :
18  17  16  15  14  13  12  11  1  2  3  4  5  6  7  8  20  21  22  23

nach Zuweisung ueber normalen Iterator (Element Nr. 9):
18  17  16  15  14  13  12  11  30  2  3  4  5  6  7  8  20  21  22  23

nach Zuweisung ueber Insert-Iterator (Element Nr. 9):
18  17  16  15  14  13  12  11  0  30  2  3  4  5  6  7  8  20  21  22  23
```

10.20 Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (1)

- **Allgemeines**
  - ◇ Die STL stellt in der Algorithmen-Bibliothek ca. **70 Algorithmen** zur Bearbeitung von Containern und den in ihnen enthaltenen Elementen zur Verfügung.
  - ◇ Diese Algorithmen sind generisch als **freie Funktions-Templates** implementiert.
    Sie greifen zu den Elementen der Container nur **indirekt über Iteratoren** zu, d.h. sind unabhängig von speziellen Implementierungs-Details der verschiedenen Container-Klassen.
    Diese **Trennung** der **Algorithmen** von den **Containern** widerspricht zwar objektorientierten Grundkonzepten, hat aber den Vorteil einer **größeren Flexibilität**, **effizienteren Implementierung** (Algorithmus muss nicht für jede Container-Klasse gesondert als Memberfunktion realisiert werden) und **einfacheren Erweiterbarkeit** (Anwendung auch für selbstdefinierte Container- und Iterator-Klassen)
  - ◇ **Alle Algorithmen** bearbeiten einen oder mehrere **Bereiche** (*ranges*) von **Elementen**.
    Ein derartiger Bereich kann alle Elemente eines Containers umfassen oder nur aus einer Teilmenge derselben bestehen.
    Der zu bearbeitende Bereich wird durch **zwei** als **Funktionsparameter** zu übergebende **Iteratoren** festgelegt.
    Dabei referiert der erste Iterator das erste zu bearbeitende Element und der zweite Iterator das Element, das auf das letzte zu bearbeitende unmittelbar folgt.
    Die zu bearbeitende Menge wird also immer als eine **halboffene Menge** angegeben : Sie besteht aus allen Elementen zwischen dem ersten (einschließlich) und dem letzten (ausschließlich) Element.
    Dabei führen die Algorithmen **keinerlei Bereichs- und Gültigkeitsüberprüfungen** aus.
    Insbesondere muss der Anwender eines Algorithmus darauf achten, dass das übergebene **Ende** eines Bereiches **vom Anfang** aus **erreichbar** sein muss. Das bedeutet, dass beide übergebenen Iteratoren zu demselben Container-Objekt gehören müssen und die Ende-Position sich logisch nicht vor der Anfangsposition befinden darf.
  - ◇ Bei den meisten Algorithmen, die mit **mehreren Bereichen** arbeiten, wird **nur der erste** Bereich direkt durch **Anfangs- und End-Iterator** festgelegt. Für die **anderen Bereiche** wird **nur** ein **Anfangs-Iterator** angegeben. Das Ende ergibt sich dann aus der Funktion und der Arbeitsweise des Algorithmus.
    Bei Algorithmen, die auf einen derartigen Bereich **schreibend** zugreifen (z.B. durch Kopieren in einen Zielbereich), muss **sichergestellt** werden, dass dieser **Bereich** eine **ausreichende Groesse** besitzt.
    Als **Alternative** können einfügende Iteratoren (**Insert-Iteratoren**) für den Zielbereich verwendet werden.
  - ◇ Bei **mehreren Bereichen** können diese in **Containern unterschiedlichen Typs** liegen.
  - ◇ Sehr viele Algorithmen liefern einen **Iterator** als **Rückgabewert**.
    Dabei wird der **End-Iterator** des übergebenen Bereichs zur **Kennzeichnung** eines **Misserfolgs** (z.B. "nicht gefunden") oder einer sonstigen Fehlfunktion verwendet.
  - ◇ **Nicht alle Algorithmen** können auf **alle Container**-Klassen **angewendet** werden.
    U.a. bestimmt die vom Algorithmus verwendete **Iterator-Katogerie** die Anwendbarkeit. So lassen sich z.B. Algorithmen, die mit Random-Access-Iteratoren arbeiten, nicht auf assoziative Container anwenden.
    Weiterhin können Algorithmen, die Elemente verändern, nicht auf Container angewendet werden, deren **Elemente** als **konstant** betrachtet werden (z.B. assoziative Container).
  - ◇ Die **Anwendung** der Algorithmen ist **nicht auf die Container-Klassen** der STL **beschränkt**. Sie lassen sich vielmehr für **alle Sequenzen** von Daten, für die **Iteratoren** zum Durchlaufen existieren, einsetzen. Insbesondere arbeiten sie auch
    - ▻ mit **Ein- bzw. Ausgabe-Streams**,
    - ▻ den **String-Klassen** der Standardbibliothek (es existieren String-Memberfunktionen zur Erzeugung von Anfangs- und End-Iteratoren)
    - ▻ sowie **normalen C-Arrays** (als Iteratoren dienen Pointer auf die Array-Elemente)
  - ◇ Die **Deklarationen** (und **Implementierungen**) der Funktions-Templates der STL-Algorithmen befinden sich in der **Headerdatei `<algorithm>`.**
  - ◇ Eine besondere – sehr kleine – Gruppe von **numerischen Algorithmen** ist nicht in der STL sondern in der **numerischen Bibliothek** enthalten. Sie sind in der **Headerdatei `<numeric>`** definiert.

- ◇ **Template-Parameter** können sein:
  - ▻ **Iteratorklassen**
    Jedes Funktions-Template ist mit wenigstens einem Iteratorklassen-Typ parametrisiert
  - ▻ **Funktionsobjekt-Klassen**
    Eine Reihe von Algorithmen können durch einen derartigen Template-Parameter konfiguriert werden. Dadurch wird ihre Funktionalität modifiziert und erweitert.
  - ▻ **Typ** der zu bearbeitenden **Daten**
    Einen derartigen Template-Parameter besitzen einige Algorithmen
  - ▻ **Datentyp** zur Darstellung **ganzer Zahlen** (Größen-Angaben)
    Einen derartigen Template-Parameter besitzen einige wenige Algorithmen
- ◇ Die **Namen** der **formalen Template-Parameter** sind so gewählt, dass sie die **Anforderungen** ausdrücken, die an die aktuellen Typ-Parameter gestellt werden.
- ◇ So bedeuten z.B. die formalen Typangaben **`InputIterator`**, **`InputIterator1`** oder **`InputIterator2`**, dass der entsprechende aktuelle Typ die funktionellen Anforderungen eines **Input-Iterators** erfüllen muss.
  Diese werden von allen Iterator-Kategorien außer den Output-Iteratoren bereitgestellt, d.h. aktueller Iterator-Typ kann dann auch ein Forward-Iterator, ein Bidirectional-Iterator oder ein Random-Access-Iterator sein.
- ◇ Analog bedeutet die formale Typangabe **`Predicate`**, dass der aktuelle Typ einstellige Funktionsobjekte beschreiben muss, die – auf einen dereferenzierten Iterator angewendet – einen auf `true` prüfbaren Wert liefern.

  In den in der ANSI-Norm enthaltenen Funktions-Deklarationen werden ausführliche und damit relativ lange formale Typnamen verwendet, die den beiden o.a. Beispielen entsprechen.
  Zur Erhöhung der Übersichtlichkeit und besseren Lesbarkeit werden diese hier durch die folgenden **abgekürzten Typ-Namen** ersetzt.
  - ▻ **`InpIt`** für `InputIterator`
  - ▻ **`OutpIt`** für `OutputIterator`
  - ▻ **`ForwIt`** für `ForwardIterator`
  - ▻ **`BidirIt`** für `BidirectionalIterator`
  - ▻ **`RandIt`** für `RandomAccessIterator`
  - ▻ **`Pred`** für `Predicate` (Klasse für einstellige Prädikate)
  - ▻ **`BinPred`** für `BinaryPredicate` (Klasse für zweistellige Prädikate)
  - ▻ **`Comp`** für `Compare` (Klasse für Vergleich-Funktionsobjekte)
  - ▻ **`UnOp`** für `UnaryOperation` (Klasse für einstellige Funktionsobjekte)
  - ▻ **`BinOp`** für `BinaryOperation` (Klasse für zweistellige Funktionsobjekte)
  - ▻ **`Func`** für `Function` (allgemeine Funktionsobjekt-Klasse)
- ◇ **Anmerkung zu Funktionsobjekt-Parametern** :
  Für Funktionsobjekt-Parameter können den Algorithmen aktuell statt Funktionsobjekten auch **Funktions-Pointer** entsprechender Funktionalität übergeben werden.

Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (3)

- **Überblick über die Algorithmen**
  - ◇ **Nichtmodifizierende Algorithmen**

    Sie führen nur Lesezugriffe zu Elementbereichen aus und ändern damit weder den Wert noch die Reihenfolge von Elementen des jeweiligen Bereichs.
    - Suchen nach Elementen und Elementfolgen (diverse verschiedene Suchkriterien)
    - Zählen von Elementen
    - Vergleich von Elementbereichen
    - Ausführen einer – nicht modifizierenden – Funktion für alle Elemente

  - ◇ **Modifizierende Algorithmen**

    Sie führen auch Schreibzugriffe zu Elementbereichen aus und ändern damit den Wert und/oder die Reihenfolge von Elementen
    - ▻ **Wert-ändernde Algorithmen**

      Sie verändern den Wert von Elementen bzw fügen Elemente hinzu (in einem Quellbereich und/oder Zielbereich)
      - Kopieren und ersetzendes Kopieren von Elementbereichen
      - Vertauschen der Elemente von zwei Bereichen
      - Transformation von Elementen
      - Ersetzen von Elementwerten
      - Füllen von Elementbereichen
      - Zusammenfassung der Elemente zweier sortierter Bereiche zu einem neuen sortierten Bereich
    - ▻ **Löschende Algorithmen**

      Sie entfernen Elemente aus einem Bereich. Hierzu zählen auch Algorithmen, die einen Quellbereich unverändert lassen und aus diesem nur einen Teil der Elemente in einen Zielbereich kopieren.
      - Entfernen von Elementen
      - Teil-Löschendes Kopieren von Elementbereichen
    - ▻ **Mutierende Algorithmen**

      Sie verändern lediglich die Reihenfolge von Elementen, nicht aber ihren Wert.
      - Vertauschen von zwei Elementen
      - Umkehrung der Elementreihenfolge
      - Rotation von Elementen
      - Permutieren der Elemente eines Bereichs
      - Umverteilung ("*shuffle*") der Elemente eines Bereichs
      - Verschiebung von Element-Teilbereichen
    - ▻ **Sortierende Algorithmen**

      Sie verändern die Reihenfolge der Elemente eines Bereichs so, dass sie anschließend wenigstens teilweise sortiert sind. Diese Algorithmen existieren alle in zwei Versionen: Die eine Version verwendet zum Sortieren den <−Operator (Default-Sortierkriterium), die andere verwendet ein als Funktionsobjekt übergebenes Sortierkriterium.
      - Sortieren aller Elemente eines Bereichs
      - Sortieren der ersten n Elemente eines Bereichs
      - Sortieren hinsichtlich eines Vergleichselements
    - ▻ **Heap-Operationen**

      Ein Heap ist ein spezieller binärer Baum, bei dem das kleinste (bzw größte) Element Wurzelelement (1. Element) ist.
      - Erzeugen, Verändern (Element-Einfügen, -Entfernen) und Sortieren (=Zerstören) eines Heaps
    - ▻ **Mengen-Operationen**

      Diese Algorithmen arbeiten mit sortierten Bereichen (Mengen)
      - Bildung der Vereinigungs-, Schnitt-, Differenz- und Komplementär-Menge zweier Mengen (→ neuer Bereich)

Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (4)

- **Anmerkung zur Darstellung der Algorithmen**

  ◇ Zur Vereinfachung und Erhöhung der Übersichtlichkeit werden im Folgenden die Funktions-Deklarationen der Algorithmen nicht in der originalen Template-Form sondern in einer vereinfachten Darstellung wiedergegeben:
  Der Template-Zusatz wird weggelassen.
  Er kann aber leicht rekonstruiert werden, da die Typen aller Funktionsparameter auch Template-Parameter sind.

  ◇ **Beispiel** :

  statt :
```
template <class ForwIt, class Size, class T, class BinPred>
  ForwIt search_n(ForwIt first, ForwIt last, Size count,
                  const T& val, BinPred pred);
```
  wird formuliert :
```
ForwIt search_n(ForwIt first, ForwIt last, Size count,
                const T& val, BinPred pred);
```

- **Einige nichtmodifizierende Algorithmen**

  ◇ Die Algorithmen dieser Gruppe arbeiten mit Input- und Forward-Iteratoren und sind daher prinzipiell auf alle Container-Klassen anwendbar. Allerdings setzen einige von ihnen sortierte Bereiche voraus.

`InpIt find(InpIt first, InpIt last, const T& val);`	Suchen nach dem ersten Element mit dem Wert `val` Rückgabe: Position des Elements, falls gefunden, sonst `last`
`InpIt find_if(InpIt first, InpIt last, Pred prd);`	Suchen nach dem ersten Element für das das Prädikat `prd` erfüllt ist Rückgabe: Position des Elements, falls gefunden, sonst `last`
`ForwIt search(ForwIt1 first1, ForwIt1 last1, ForwIt2 first2, ForwIt2 last2);`	Suchen nach erstem Vorkommen des 2. Bereichs im 1. Bereich, Rückgabe : Pos. des 1.Elements des gefundenen Teilbereichs, bzw. `last1`
`ForwIt lower_bound(ForwIt first, ForwIt last, const T& val);`	Rückgabe: Position des ersten Elements dessen Wert `>= val` ist (Bereich muss sortiert sein)
`const T& max(const T& a, const T& b);`	Rückgabe: Referenz auf größeres Element
`ForwIt max_element(ForwIt first, FormIt last);`	Rückgabe: erste Position des größten Elements
`ForwIt max_element(ForwIt first, FormIt last, Comp cmp);`	Rückgabe: erste Position des größten Elements Vergleich erfolgt mittels `cmp`
`InpIt::difference_type count(InpIt first, InpIt last, const t& val);`	Suchen nach Elementen mit dem Wert val Rückgabe: Anzahl der gefundenen Elemente
`InpIt::difference_type count_if(InpIt first, InpIt last, Pred prd);`	Suchen nach Elementen, für die `prd` erfüllt ist Rückgabe: Anzahl der gefundenen Elemente
`bool equal(InpIt1 first1, InpIt1 last1, InpIt2 first2);`	Überprüfung zweier Bereiche auf Gleichheit Rückgabe: `true`, wenn gleich, sonst `false`
`UnOp for_each(InpIt first, InpIt last, UnOp f);`	Aufruf von `f(elem)` für alle Elemente `elem` des Bereichs `[first, last)`, Rückgabe: `f`

- **Einige wert-ändernde Algorithmen**
  - ◇ Da die Elemente von assoziativen Containern als konstant betrachtet werden, lassen sich diese Algorithmen nicht anwenden, wenn der Bereich, in dem Elemente geändert werden, in einem assoziativen Container liegt.

`OutpIt copy(InpIt first, InpIt last, OutpIt res);`	Kopieren der Elemente aus dem Bereich `[first,last)` in den bei `res` beginnenden Bereich `res` darf nicht im Bereich `[first,last)` liegen Rückgabe: `res + (last − first)`
`BidirIt copy_backward(BidirIt1 first, BidirIt1 last, BidirIt2 res);`	Kopieren der Elemente aus dem Bereich `[first,last)` in umgekehrter Reihenfolge in den Bereich ab `res` `res` darf nicht im Bereich `[first,last)` liegen Rückgabe : `res + (last − first)`
`OutpIt replace_copy(InpIt first, InpIt last, OutpIt res, const T& alt, const T& neu);`	Kopieren der Elemente aus dem Bereich `[first,last)` in den bei `res` beginnenden Bereich, wobei der Wert `alt` durch den Wert `neu` ersetzt wird. `res` darf nicht im Bereich `[first,last)` liegen Rückgabe: `res + (last − first)`
`ForwIt2 swap_ranges(ForwIt1 first1, ForwIt1 last1, ForwIt2 first2);`	Vertauschen der Elemente des Bereiches `[first1, last1)` mit den Elementen des Bereichs der mit `first2` beginnt. Rückgabe: `first2 + (last1 − first1)` Die beiden Bereiche dürfen sich nicht überlappen
`OutpIt transform(InpIt first, InpIt last, OutpIt res, UnOp op);`	Jedes Element `*pos` aus dem Bereich `[first,last)` wird mittels `op(*pos)` transformiert und als neues Element in den Bereich, der bei `res` beginnt, kopiert. `res` darf mit `first` übereinstimmen Rückgabe: `res + (last − first)`
`void replace(ForwIt first, ForwIt last, const T& alt, const T& neu);`	Ersetzen des Werts `alt` der Elemente aus dem Bereich `[first,last)` durch den Wert `neu`.
`void fill(ForwIt first, ForwIt last, const T& val);`	Zuweisen des Werts `val` an jedes Element aus dem Bereich `[first,last)`
`void generate(ForwIt first, ForwIt last, Generator gen);`	Aufruf von `gen()` für jedes Element aus dem Bereich `[first,last)` und Zuweisen des dadurch erzeugten Werts an das jeweilige Element
`void generate_n(OutpIt first, Size n, Generator gen);`	Aufruf von `gen()` für die ersten `n` Elemente des mit `first` beginnenden Bereichs und Zuweisen des dadurch erzeugten Werts an das jeweilige Element
`OutpIt merge(InpIt1 first1, InpIt1 last1, InpIt2 first2, InpIt2 last2, OutpIt res);`	Zusammenfassen der Elemente der beiden sortierten Bereiche `[first1, last1)` und `[first2, last2)` in einen bei `res` beginnenden ebenfalls sortierten Bereich Quell- und Zielbereiche dürfen sich nicht überlappen Rückgabe: Position hinter dem letzten Element im Zielber.

Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (6)

- **Einige löschende Algorithmen**

  ◇ Da die Elemente von assoziativen Containern als konstant betrachtet werden, lassen sich diese Algorithmen nicht anwenden, wenn der Bereich, in dem Elemente geändert werden, in einem assoziativen Container liegt.

`ForwIt remove(ForwIt first, ForwIt last, const T& val);`	Löschen aller Elemente aus dem Bereich `[first,last)`, die den Wert `val` haben Rückgabe: Position hinter dem letzten Element im modifizierten Bereich
`ForwIt remove_if(ForwIt first, ForwIt last, Pred prd);`	Löschen aller Elemente aus dem Bereich `[first,last)` für die das Prädikat `pred` erfüllt (`true`) ist Rückgabe: Pos. hinter dem letzten Element im mod. Bereich
`OutpIt remove_copy (InpIt first, InpIt last, OutpIt res); const T& val);`	Kopieren der Elemente aus dem Bereich `[first,last)`, die nicht den Wert `val` haben, in den bei `res` beginnenden Bereich Rückgabe: Position hinter dem letzten Element im Zielber.
`OutpIt remove_copy_if (InpIt first, InpIt last, OutpIt res); Pred prd);`	Kopieren der Elemente aus dem Bereich `[first,last)`, für die das Prädikat `prd` nicht erfüllt ist, in den bei `res` beginnenden Bereich Rückgabe: Position hinter dem letzten Element im Zielber.
`ForwIt unique(ForwIt first, ForwIt last);`	Löschen aller Elemente aus dem Bereich `[first,last)`, die einem Element mit gleichem Wert unmittelbar folgen Rückgabe : Pos. hinter dem letzten Element im mod. Bereich
`OutpIt unique_copy(InpIt first, InpIt last, OutpIt res);`	Kopieren der Elemente aus dem Bereich `[first,last)`, die nicht einem Element mit gleichem Wert unmittelbar folgen, in den bei `res` beginnenden Bereich,. `res` darf nicht im Bereich `[first,last)` liegen Rückgabe: Position hinter dem letzten Element im Zielber.

- **Einige sortierende Algorithmen** (nur für Container mit Random-Access-Iteratoren)

`void sort(RandIt first, RandIt last);`	Sortieren der Elemente im Bereich `[first,last)`, nach aufsteigender Reihenfolge (mit `operator<`)
`void sort(RandIt first, RandIt last, Comp cmp);`	Sortieren der Elemente im Bereich `[first,last)` mittels des durch `cmp` festgelegten Vergleichskriteriums
`void partial_sort(RandIt first, RandIt middle, RandIt last);`	Sortieren der Elemente im Bereich `[first,last)` so, dass anschließend die ersten `(middle-first)` sortierten Elemente im Bereich `[first, middle)` stehen. Sortierung nach aufsteig. Reihenfolge (mit `operator<`) Die restlichen Elemente sind unsortiert.
`void nth_element(RandIt first, RandIt nth, RandIt last);`	Platzierung des Elements aus dem Bereich `[first,last)` das bei vollständiger Sortierung an der Position `nth` stehen würde, an diese Position (Vergleichselement). Im Teilbereich `[first, nth)` befinden sich (unsortiert) nur Elemente, die kleiner gleich dem Vergleichselement sind, im Teilbereich `[nth+1, last)` die restlichen Elemente

Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (7)

- **Einige mutierende Algorithmen**
  - ◇ Da die Elemente von assoziativen Containern als konstant betrachtet werden, lassen sich diese Algorithmen nicht anwenden, wenn der Bereich, in dem Elemente geändert werden, in einem assoziativen Container liegt.

`void swap(T& a, T& b);`	Vertauschen der Elemente `a` und `b`
`void reverse(BidirIt first, BidirIt last);`	Umkehren der Reihenfolge der Elemente des Bereichs `[first,last)`
`void rotate(ForwIt first, ForwIt middle, ForwIt last);`	Rotieren der Elemente aus dem Bereich `[first,last)` um `(middle - first)` Positionen nach links, so dass sich anschließend das ehemalige Element an der Position `middle` an der Position `first` befindet `middle` muss im Bereich `[first,last)` liegen
`OutpIt rotate_copy(ForwIt first, ForwIt middle, ForwIt last, OutpIt res);`	Kopieren der Elemente aus dem Bereich `[first,last)` in den bei `res` beginnenden Bereich, wobei die Elemente um `(middle - first)` Positionen nach links rotiert werden, so dass sich anschließend das ehemalige Element an der Position `middle` an der Position `res` im Zielbereich befindet. Quell- und Zielbereich dürfen sich nicht überlappen Rückgabe: `res + (last - first)`
`void random_shuffle(RandIt first, RandIt last);`	Änderung der Reihenfolge der Elemente des Bereiches `[first, last)` nach einer gleichmäßigen Verteilung.
`void random_shuffle(RandIt first, RandIt last, RandNumGen& rand);`	Änderung der Reihenfolge der Elemente des Bereiches `[first, last)` nach einer durch den Zufallszahlen-generator `rand` festgelegten Verteilung
`BidirIt partition(BidirIt first, , BidirIt last, Pred prd);`	Verschieben aller Elemente aus dem Bereich `[first,last)`, für die das Prädikat `prd` erfüllt ist, vor alle Elemente, für die es nicht erfüllt ist.. Rückgabe: Position des ersten Elements, für die das Prädikat `prd` nicht erfüllt ist

- **Einige Mengen-Operationen**

`OutpIt set_intersection (InpIt1 first1, InpIt1 last1, InpIt2 first2, InpIt2 last2, OutpIt res);`	Bildung einer sortierten Menge aus den Elementen, die sowohl in dem sortierten Bereich `[first1,last1)` als auch in `[first2, last2)` enthalten sind und Kopieren derselben in den bei `res` beginnenden Bereich. (Bildung der Schnittmenge) Rückgabe: Position hinter dem letzten Element im Zielber.
`OutpIt set_union(/* Parameter wie oben */);`	Bildung der Vereinigungsmenge
`OutpIt set_difference(/* Par. wie oben */);`	Bildung der Differenzmenge (Menge1 – Menge2)

Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (8)

- **Einfaches Demonstrationsprogramm (1) zu Algorithmen `algodem1`**

```
// C++-Quelldatei algo1dem1_m.cpp --> Programm algo1dem1
// Einfaches Demo-Programm zu Algorithmen der STL

#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
using namespace std;

template <class T>
  class ValOut
  { public: void operator()(const T& x) { cout << x << "  "; }
  };

template <class T>
  void checkFor(list<T>& lis, const T& val)
  { list<T>::iterator p;
    p=find(lis.begin(), lis.end(), val);
    cout << "gesucht : " << val ;
    if (p==lis.end()) cout << " und nicht gefunden" << endl;
    else              cout << " und gefunden : " << *p << endl;
  }

bool mod3(int x) { return x%3==0; }

int main(void)
{ list<int> li1;
  list<int>::difference_type cnt;
  list<int>::iterator ilp;
  li1.push_back(2);   li1.push_back(7);
  li1.push_back(9);   li1.push_back(3);
  li1.push_back(6);   li1.push_back(5);
  li1.push_back(15);  li1.push_back(11);
  ValOut<int> ivout;                          // Funktionsobjekt
  cout << "\nInhalt der Liste :\n";
  for_each(li1.begin(), li1.end(), ivout);
  cout << endl << endl;;
  checkFor(li1, 7);
  checkFor(li1, 8);
  cnt=count_if(li1.begin(), li1.end(),mod3);
  cout << "\nAnzahl durch 3 teilbarer Zahlen : " << cnt << endl;
  ilp=find_if(li1.begin(), li1.end(),mod3);
  cout << "Erste durch 3 teilbare Zahl     : " << *ilp << endl;
  ilp=max_element(li1.begin(), li1.end());
  cout << "Groesste enthaltene Zahl        : " << *ilp << endl;
  vector<int> vi1(li1.size());
  vector<int>::iterator ivp;
  copy(li1.begin(), li1.end(), vi1.begin());
  cout << "\nInhalt des Vektors :\n";
  for_each(vi1.begin(), vi1.end(), ivout);
  cout << endl << endl;
  int i=8;
  ivp=lower_bound(vi1.begin(), vi1.end(), i);
  cout << "Erste Zahl >= " << i << " (unsortiert) : " << *ivp << endl;
  sort(vi1.begin(), vi1.end());
  cout << "\nInhalt des Vectors nach Sortierung :\n";
  copy (vi1.begin(), vi1.end(), ostream_iterator<int>(cout, "  "));
  cout << endl << endl;
  ivp=lower_bound(vi1.begin(), vi1.end(), i);
```

(STL) von C++ : Algorithmen-Bibliothek (8)

- **Ausgabe des Demonstrationsprogramms `algodem1`**

```
Inhalt der Liste :
2  7  9  3  6  5  15  11

gesucht : 7 und gefunden : 7
gesucht : 8 und nicht gefunden

Anzahl durch 3 teilbarer Zahlen : 4
Erste durch 3 teilbare Zahl     : 9
Groesste enthaltene Zahl        : 15

Inhalt des Vektors :
2  7  9  3  6  5  15  11

Erste Zahl >= 8 (unsortiert) : 15

Inhalt des Vectors nach Sortierung :
2  3  5  6  7  9  11  15

Erste Zahl >= 8 (sortiert)   : 9
```

Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (9)

- **Einfaches Demonstrationsprogramm (2) zu Algorithmen `algodem2`**

```
// C++-Quelldatei algo1dem2_m.cpp --> Programm algo1dem2
// Einfaches Demo-Programm zu Algorithmen der STL

#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
#include <set>
#include <cstdlib>
#include <functional>
#include <iterator>
using namespace std;

class GenInt
{ public :
    GenInt (unsigned s=1) { srand(s);}
    int operator()(void) { return (int)(rand()%100);}
};

int main(void)
{ vector<int> vi1;
  vi1.push_back(2);    vi1.push_back(7);    vi1.push_back(9);
  vi1.push_back(3);    vi1.push_back(24);   vi1.push_back(6);
  vi1.push_back(15);   vi1.push_back(5);    vi1.push_back(11);
  ostream_iterator<int> iaus(cout, "  ");
  cout << "\nInhalt des Vektors :\n";
  copy (vi1.begin(), vi1.end(), iaus);   cout << endl << endl;
  set<int> si1;
  copy(vi1.begin(), vi1.end(), inserter(si1, si1.begin()));
  cout << "\nInhalt des Sets :\n";
  copy (si1.begin(), si1.end(), iaus);   cout << endl << endl;
  GenInt rnum(1111);
  vi1.resize(6);
  generate(vi1.begin(), vi1.end(), rnum);
  cout << "\nInhalt des Vektors nach Zufallszahlen-Zuweisung :\n";
  copy (vi1.begin(), vi1.end(), iaus);   cout << endl << endl;
  generate_n(back_inserter(vi1), 5, rnum);
  cout << "\nInhalt des Vektors nach Verlaengerung um Zufallszahlen :\n";
  copy (vi1.begin(), vi1.end(), iaus);   cout << endl << endl;
  sort(vi1.begin(), vi1.end());
  list<int> li1(si1.size()+vi1.size());
  merge(si1.begin(), si1.end(), vi1.begin(), vi1.end(), li1.begin());
  cout << "\nInhalt der Zusammenfassung des Sets und des sortierten Vektors "
          "(--> Liste) :\n";
  copy (li1.begin(), li1.end(), iaus);   cout << endl << endl;;
  list<int>::iterator ilp = unique(li1.begin(), li1.end());
  cout << "\nInhalt der Liste nach Entfernung aufeinanderfolgender Duplikate :\n";
  copy (li1.begin(), li1.end(), iaus);   cout << endl << endl;;
  li1.erase(ilp, li1.end());
  cout << "\nInhalt der Liste nach Entfernung uberfluessiger End-Elemente :\n";
  copy (li1.begin(), li1.end(), iaus);   cout << endl << endl;;
  list<int> li2;
  remove_copy_if(li1.begin(), li1.end(), back_inserter(li2),
                 not1(bind2nd(modulus<int>(), 2)));
  cout << "\nInhalt der 2. Liste nach entfernenden Kopieren :\n";
```

Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (10)

- **Ausgabe des Demonstrationsprogramms `algodem2`**

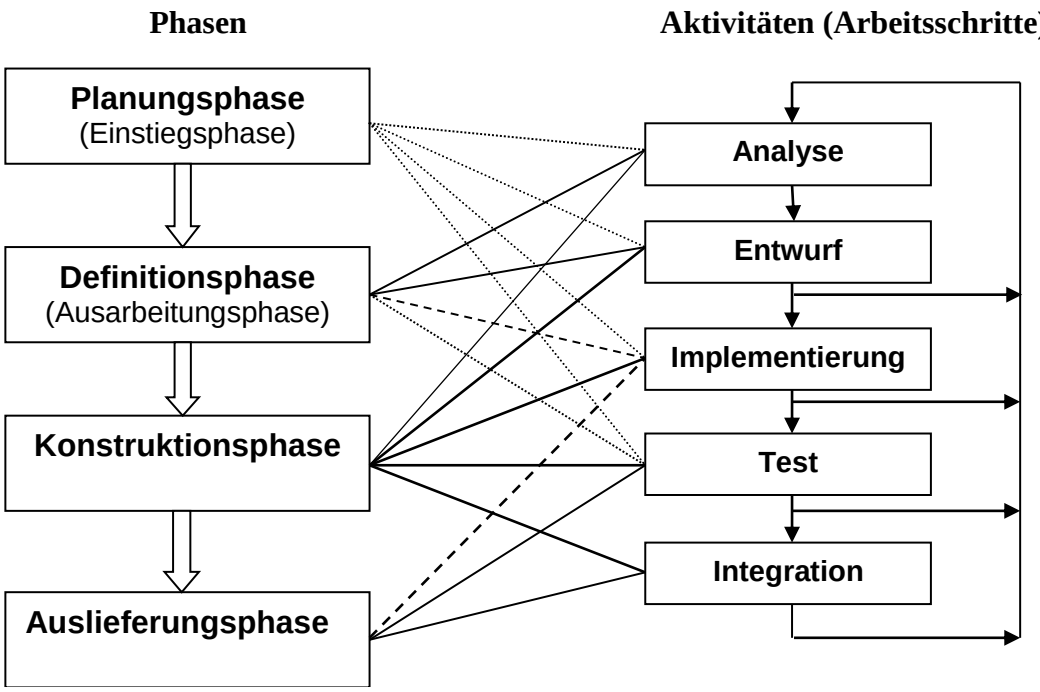
```
Inhalt des Vektors :
2  7  9  3  24  6  5  15  11


Inhalt des Sets :
2  3  5  6  7  9  11  15  24


Inhalt des Vektors nach Zufallszahlen-Zuweisung :
66  47  24  58  15  50


Inhalt des Vektors nach Verlaengerung um Zufallszahlen :
66  47  24  58  15  50  58  15  47  53  69


Inhalt der Zusammenfassung des Sets und des sortierten Vektors (--> Liste) :
2  3  5  6  7  9  11  15  15  15  24  24  47  47  50  53  58  58  66  69


Inhalt der Liste nach Entfernung aufeinanderfolgender Duplikate :
2  3  5  6  7  9  11  15  24  47  50  53  58  66  69  53  58  58  66  69


Inhalt der Liste nach Entfernung ueberfluessiger End-Elemente :
2  3  5  6  7  9  11  15  24  47  50  53  58  66  69


Inhalt der 2. Liste nach entfernenden Kopieren :
3  5  7  9  11  15  47  53  69
```

11 Entwicklung von OOP-Programmen

11.1 Der Softwareentwicklungsprozess (Überblick)

- **Entwicklungsziel** :
  - ◇ Das **Ergebnis eines erfolgreichen Softwareentwicklungsprozesses** soll ein **Software-System** sein, dass
    - ▻ die vom Benutzer gestellten **funktionalen** und **nichtfunktionalen Anforderungen und Erwartungen** erfüllt
    - ▻ **zeitgerecht** und **wirtschaftlich** entwickelt wurde
    - ▻ **änderungs**- und **anpassungsfähig** ist
  - ◇ Ein derartiger Prozess muss einerseits ein ausreichendes Maß an **Kreativität** und **Innovation** unterstützen, andererseits muss er die notwendige **Überprüfung** und **Steuerung des Entwicklungsfortschritts** sicherstellen.
  - ◇ Diese Forderungen werden von einem **iterativen** und **inkrementellen** Entwicklungsprozess erfüllt.
    Ein derartiger **Prozess** besteht aus zeitlich nacheinander ablaufenden **Phasen**, innerhalb denen bestimmte **Aktivitäts-Zyklen** (Arbeitsschrittzyklen) **wiederholt** - mit teilweise unterschiedlicher Gewichtung - ausgeführt werden.

- **Phasen und Aktivitäten des Entwicklungsprozesses** :



  - ◇ Insbesondere die **Konstruktionsphase** besteht aus zahlreichen **Iterationen**.
    In jeder Iteration wird ein vollständiger **Aktivitätszyklus** - z. Teil in sich auch wieder zyklisch – durchlaufen, wobei jeweils Software erstellt, getestet und integriert wird, die eine **Teilmenge der Gesamtanforderungen** des Projekts abdeckt.
    → Schrittweise Weiterentwicklung von **Systemprototypen** mit zunächst unvollständiger Funktionalität bis zum fertigen System.
    Ein vollständiger Aktivitätszyklus kann aber auch bereits in der **Planungsphase** und der **Definitionsphase** zur schnellen Erzeugung **explorativer Prototypen** durchlaufen werden.
  - ◇ In jeder Entwicklungsphase können sich **Rückwirkungen** auf Ergebnisse und Festlegungen **früherer Phasen** ergeben, insbesondere kann derartig die Konstruktionsphase auf die Definitionsphase rückwirken.
  - ◇ In **allen Phasen** und **Tätigkeiten** sollten die jeweiligen Ergebnisse und Festlegungen geeignet **dokumentiert** werden.

Die Phasen des Softwareentwicklungsprozesses

- **Planungsphase** (Einstiegsphase, Anfangsphase, Vorphase, inception phase)

  ◇ **Ziel** : Prüfung, ob ein Projekt realisiert werden soll.

  ◇ Festlegung von Zweck und Ziel eines Projektes,
  Durchführung von Trendstudien und Marktanalysen,
  Untersuchung der Durchführbarkeit
  (Aufwands-, Bedarfs- und Termin-Abschätzung, Risikoanalyse, Wirtschaftlichkeitsrechnung)
  ⇒ **Durchführbarkeitsstudie** (feasability study)
  bestehend aus : Lastenheft (grobes Pflichtenheft), Projektkalkulation und Projektplan

- **Definitionsphase** (Ausarbeitungsphase, Elaborationsphase, elaboration phase)

  ◇ **Ziel** : Festlegung der vollständigen, konsistenten, eindeutigen und durchführbaren Projektanforderungen

  ◇ Ermittlung und Analyse der Anforderungen (Systemanalyse)
  Problembereichanalyse (→erste Exploration von Klassen), Abgrenzung des Software-Systems zur Systemumgebung
  → Problembereichsmodell
  Identifikation der **Nutzungsfälle** (Use Cases) → Nutzungsfallmodell
  ⇒ **Anforderungsspezifikation (Pflichtenheft)**

  Entwurf der grundlegenden **Systemarchitektur** (→weitere Exploration von **Klassen**)
  Konzipierung der externen Schnittstellen (z.B. Benutzeroberfläche)
  Risikoanalyse und Priorisierung der Nutzungsfälle
  Festlegung der Entwurfsstrategie
  Planung der Systementwicklung in inkrementellen Entwicklungsschritten (→ Evolutionsplan, Entwurfsmodell)
  ⇒ **Entwurfsspezifikation**

- **Konstruktionsphase** (construction phase)

  ◇ **Ziel** : Erstellung eines die festgelegten Anforderungen erfüllenden Softwaresystems (Programm(e))

  ◇ Entwicklung einer detaillierten Systemstruktur
  Identifikation und Definition weiterer **Klassen/Objekte**
  Konzipierung und Analyse der statischen Klassenbeziehungen → **Klassendiagramme**
  Konzipierung und Analyse der dynamischen Objektinteraktion → **Interaktionsdiagramme**, K**ollaborationsdiagr.**
  → Erstellung **statischer und dynamischer Modelle**

  ◇ Erstellung des Systems in einer Folge von Iterationen (**zyklische Evolution**)
  (jeweils Analyse, Entwurf, Implementierung, Test und Integration)
  → inkrementelles Hinzufügen weiterer Funktionalitäten
  → iterative Änderung des Codes (z.B. Umstrukturierung)
  Validierung jeder Iteration durch einen geeigneten Test
  ⇒ **ausgetesteter Code mit Dokumentation**
  Benutzer- und Installationshandbücher
  Prüf- und Testprotokolle

- **Auslieferungsphase** (Abnahme- u. Einführungsphase, Überleitungsphase, transition phase)

  ◇ **Ziel** : Auslieferung eines fertiggestellten Produkts an den Kunden/Anwender

  ◇ Abnahme des Produkts durch den Kunden (→ Abnahmetest → **Abnahmeprotokoll**)
  Gegebenenfalls Korrektur von festgestellten Fehlern, Installation und Inbetriebnahme des Produkts
  Schulung der Benutzer

Klassenkategorien

- **Allgemeines**
  - ◇ Ein **objektorientiertes Softwaresystem** (OO-Programm) ist als eine Ansammlung **interagierender Objekte** organisiert.
  - ◇ Die einzelnen Objekte decken dabei unterschiedliche Aufgabenbereiche ab. Entsprechend ihrem jeweiligen Aufgabenbereich können sie und damit die sie beschreibenden Klassen unterschiedlichen **Kategorien** zugeordnet werden.
    → Objekte und Klassen besitzen einen **Stereotypen**.
  - ◇ Auch die gegebenenfalls zwischen Klassen bestehenden **Beziehungen** lassen sich in **verschiedene Arten** einteilen.

- **Gebräuchliche Klassenkategorien**

  ▻ **Entity-Klassen**
  Das sind Klassen, deren Objekte Bestandteil des Problembereichs sind (**Domänen-Klassen**).
  Sie reflektieren entweder **reale Objekte** des Problembereichs oder sie werden zur Wahrnehmung **interner Aufgaben** des Systems benötigt.
  Es kann sich dabei um **konkrete technische Objekte** (z.B. Pkw) handeln, es können aber auch **Personen** und deren **Rollen** (z.B. Student, Kunde) **Orte**, (z.B. Hörsaal), **Organisationseinheiten** (z.B. FAKULTÄT FÜR), **reignisse** (z.B. Unfall), **Informationen über (Inter**-)**Aktionen** (z.B. Kaufvertrag), **Konzepte** (z.B. Entwicklungsplan), sonstige **allgemeine** oder **problembereichsbezogene Begriffe** (z.B. Lehrveranstaltung) **usw**. sein Objekte von Entity-Klassen sind typischerweise **unabhängig von der Systemumgebung** und von der Art und Weise, wie das System mit der Umgebung kommuniziert.
  Häufig sind sie auch **applikationsunabhängig**, d.h. sie lassen sich in mehreren Applikationen einsetzen.

  ▻ **Interface-Klassen** (*Boundary Classes*)
  Objekte dieser Klassen sind für die **Kommunikation** zwischen der Systemumgebung und dem Inneren des Systems zuständig. Sie realisieren die **Schnittstelle** des Systems **zum Systembenutzer** (Benutzeroberfläche) bzw **zu anderen Systemen**.
  → Sie bilden den **umgebungsabhängigen** Teil eines Systems.

  ▻ **Controller-Klassen** (*Control Classes*)
  Objekte von Controller-Klassen **steuern** den **Ablauf**, d.h. die Zusammenarbeit der übrigen Objekte zur Realisierung eines oder mehrerer Use Cases. Es handelt sich um **aktive** Objekte. Typischerweise sind sie **applikationsspezifisch**.

  ▻ **Service-Klassen**
  Objekte von Service-Klassen stellen anderen Objekten (insbesondere den Controller-Objekten) **Dienste** zur Verfügung. Häufig handelt es um Klassen aus einer **Bibliothek**.
  Sie können aber auch durch **Auslagerung von Funktionalitäten** anderer Klassen gebildet werden.

  ▻ **Utility-Klassen**
  Sie **kapseln** Daten und Operationen (Funktionen), die **global** verfügbar sein sollen.
  Im Allgemeinen werden sie **nicht instantiiert**.
  Beispiel : mathematische Konstanten u. mathematische Funktionen.
  Die konsequente Unterscheidung und Trennung von Entity-, Interface- und Controller-Klassen führt zur "***Model-View-Controller***"-Architektur (MVC-Architektur), ein allgemein eingeführtes OOP-Paradigma.
  Das User-Interface (View) ist von der eigentlichen Problembearbeitung (Model) getrennt und weitgehend entkoppelt. Die Verbindung zwischen beiden wird über einen Controller hergestellt.

Beziehungen zwischen Klassen

- **Vererbungsbeziehung**

  ◇ Beziehung zwischen Klassen, deren Komponenten sich teilweise überdecken

  ◇ Eine abgeleitete Klasse erbt die Eigenschaften und Fähigkeiten (Komponenten) der Basisklasse(n).
  "ist"-Beziehung → ein Objekt der abgeleiteten Klasse ist auch ein Objekt der Basisklasse(n)

  ◇ Ordnungsprinzip bei der Spezifikation von Klassen.
  → **Generalisierung / Spezialisierung**

- **Nutzungsbeziehungen**

  ◇ Unter einer (statischen) Nutzungsbeziehung versteht man eine in einem konkreten Anwendungsbereich geltende Beziehung zwischen Klassen, deren Instanzen voneinander Kenntnis haben und die dadurch miteinander **kommunizieren** können
  → Nutzungsbeziehungen sind notwendig für die **Interaktion von Objekten**

  ◇ **Assoziation**

  Spezielle Beziehung zwischen Klassen bzw. Objekten, bei der die Objekte **unabhängig** voneinander existieren und **lose** miteinander **gekoppelt** sind
  Beispiel: einem Objekt wird ein anderes Objekt als Parameter übergeben.
  - **Name**: Kennzeichnung der Semantik der Beziehung zwischen den Klasseninstanzen
  - **Navigationsrichtung**: legt die Kommunikationsrichtung und die Richtung, in der ein Objekt der einen Klasse ein Objekt der anderen Klasse referieren kann, fest.
  - bidirektional (Kommunikation in beiden Richtungen möglich) oder unidirektional
  - **Rolle** : Ein Name für die Aufgabe, die ein Objekt der assoziierten Klasse aus der Sicht eines Objekts der assoziierenden Klasse wahrnimmt.
  - **Kardinalität** (*multiplicity*): bezeichnet die mögliche Anzahl der an der Assoziation beteiligten Instanzen einer Klasse.

  ◇ **Aggregation**
  - Spezielle Beziehung zwischen Klassen bzw Objekten, bei der die Objekte der einen Klasse **Bestandteile** (Komponenten) eines oder mehrerer Objekte der anderen Klasse sind.
    → zwischen den Objekten besteht eine **feste Kopplung**
  - **"hat"**-Beziehung bzw. **"ist Teil von"**-Beziehung
  - Das "umschließende" Objekt bildet einen Container für das bzw. die enthaltene(n) Objekt(e)
  - Aggregation kann als **Spezialfall der Assoziation** aufgefasst werden.
    → die für Assoziationen möglichen Kennungen (Name usw.) lassen sich auch für Aggregationen verwenden.
  - eine Aggregation ist i.a. aber eine **unidirektionale** Beziehung (Navigation vom umschließenden Objekt zu den Komponenten-Objekten)

  Je nach dem **Grad der Kopplung** unterscheidet man:

  ▻ **einfache Aggregation**

  Das umschließende Objekt (Aggregat) und die Komponenten sind **funktionell aneinander gebunden**.
  Eine Komponente kann zusätzlich noch **weiteren** Aggregaten der gleichen oder einer anderen Klasse zugeordnet sein. Bei Löschung des Aggregats **dürfen** die Komponenten unabhängig vom Aggregat auch erhalten bleiben.
  Beispiel : Klasse mit dynamisch erzeugten Komponenten

  ▻ **echte Aggregation** (Komposition, *composite aggregation)*

  Die Komponenten können **nur einem** Aggregat zugeordnet sein und nur **innerhalb** des Aggregats **existieren**.
  Bei Löschung des Aggregats werden auch die Komponenten gelöscht.
  Beispiel : Klasse mit statisch allozierten Komponenten

  ◇ **Anmerkung** :

  In der Praxis kann es im Einzelfall sehr schwierig sein, zwischen Assoziation und Aggregation und den verschiedenen Formen der Aggregation zu unterscheiden.

Identifikation von Klassen und ihren Beziehungen

- **Allgemeines**
  - ◇ Das **Identifizieren** von **Klassen** und ihren **Beziehungen** ist eine **zentrale Aufgabe** der **Analyse**-Tätigkeit innerhalb des OO-Entwicklungsprozesses.
    Es handelt sich um eine sehr schwierige, äußerst kreative Tätigkeit, die methodisch kaum unterstützt wird und für die kein allgemeingültiges "Kochrezept" angegeben werden kann.
    Zum Erwerb der notwendigen eigenen Erfahrungen kann man sich lediglich von Empfehlungen, Tipps und Tricks (**Heuristiken**) , die einschlägige Experten mittlerweile in der Literatur veröffentlich haben, leiten lassen.
  - ◇ Da der Entwicklungsprozess iterativ verläuft, wird sich die **Liste der gefundenen Klassen** im Laufe dieses Prozesses **ändern** → die im ersten Schritt gefundenen Klassen, werden i.a. nicht der endgültig realisierten Liste entsprechen, neue Klassen werden hinzukommen, bereits identifizierte Klassen werden modifiziert (z.B. geteilt oder zusammengelegt) bzw. wieder verworfen.
    Analoges gilt für die **Attribute** (Datenkomponenten) und **Operationen** (Funktionskomponenten) der Klassen, sowie für die **Beziehungen** zwischen den Klassen.

- **Identifikation von Klassen**
  - ▻ Ermittlung der **konkreten technischen Objekte** des Problembereichs (**Fachklassen**).
  - ▻ Ermittlung von Objekten/Klassen aus im Problembereich eingesetzten **Formularen** (Formularanalyse).
  - ▻ Untersuchung der Beschreibung der Szenarien der einzelnen *Use Cases* (oder sonstiger verbal formulierter Anforderungen an das System, wie z.B. das Pflichtenheft) nach **Hauptworten**. Diese sind i.a. Kandidaten für Klassen.
  - ▻ Einsatz von **CRC-Karten** (CRC – *Class*, *Responsibilities*, *Collaborations*)
    Anlegen kleiner Karten, auf denen der Klassenname, die Zuständigkeiten (*responsibilties*) und die mit ihnen kollaborierenden Klassen (zu denen also Assoziationen bestehen müssen) vermerkt werden.
    Überprüfung der einzelnen Anwendungsfälle mittels der CRC-Karten auf Erfüllung der geforderten Funktionalität.
    Übervolle Karten bedeuten entweder schlecht ausgearbeitete Zuständigkeiten oder nichtkohäsive Klassen.
  - ▻ Untersuchung jedes **Aktor-/Scenario-Paares** zur Ermittlung der **Interface-Klassen**.
  - ▻ Hinzufügen je einer **Controller-Klasse** für jedes **Aktor/Scenario-Paar**.
  - ▻ Zuordnung der gefundenen Klassen zu den verschiedenen **Klassenkategorien**.
  - ▻ Wählen **aussagekräftiger Namen** für die gefundenen Klassen.
  - ▻ Zuordnen von **Operationen** und **Attributen**, im ersten Schritt allerdings nur soweit es zur **Unterscheidung** der Klassen und zur Abdeckung der Zuständigkeiten notwendig ist.
  - ▻ **Filterung** der gefundenen Klassenmenge hinsichtlich **tatsächlicher Problembereichszugehörigkeit**, echter **Substanz**, eindeutiger **Abgrenzung**, **Überschneidungen**, **Redundanz** usw.
  - ▻ Gegebenenfalls sind Klassen wieder zu verwerfen, zusammenzufassen, zu teilen oder Komponenten in gemeinsame Basisklassen auszulagern (**Refaktorierung**).

- **Identifikation der Klassenbeziehungen**
  - ▻ Untersuchung der gefundenen Klassen auf Gemeinsamkeiten (→ Vererbungsbeziehungen)
  - ▻ Ermittlung der Nutzungsbeziehungen kann parallel zur Identifikation der Klassen beginnen (CRC-Karten !)
  - ▻ Untersuchung der einzelnen Scenarios auf **Verben**. Diese können auf zwischen Objekten auszutauschende Botschaften hinweisen. Der Austausch von Botschaften erfordert Nutzungsbeziehungen.
  - ▻ Rangordnung der beteiligten Klassen überprüfen.
    Über-/Unterordnung kann auf Aggregation hinweisen.
  - ▻ Überprüfung, ob "hat"- bzw "ist Teil von"-Beziehung vorliegt.
  - ▻ Im Zweifelsfall Assoziation wählen.
  - ▻ Kennungen (Name, Navigationsrichtung, Rollenname, Kardinalität usw) der gefundenen Beziehungen festlegen.

11.2 Modellierung

- **Allgemeines**
  - ◇ Das zu entwickelnde **Software-System** sowie der von ihm abzudeckende **Problembereich** sind häufig so **komplex**, dass sie sich nur mit Hilfe geeigneter **Hilfsmittel** ausreichend erfassen lassen.
    Ein derartiges Hilfsmittel stellt die **Modellierung** dar.
  - ◇ I.a. ist es wenig sinnvoll das Gesamtsystem in einem einzigen – alle Einzelheiten erfassenden – Modell darzustellen.
    Vielmehr setzt man **unterschiedliche Modelle** ein, die jeweils verschiedene Teilaspekte des Systems repräsentieren.
    Diese Modelle erleichtern nicht nur das Problembereichs- und Systemverständnis, sondern bilden auch ein wesentliches Kommunikationsmittel aller an der Systementwicklung beteiligten Personen und stellen damit auch einen wichtigen Bestandteil der Systemdokumentation dar.
  - ◇ Voraussetzung für die Bildung adäquater, aussagekräftiger, eindeutiger und leicht verständlicher Modelle ist eine **geeignete Notation** zur Modelbeschreibung.
    Für den Bereich der objektorientierten Systementwicklung ist dies die **Unified Modeling Language** (**UML**), mit der Modelle in überwiegend **graphischer** Notation (→ **Diagramme**) dargestellt werden können.
  - ◇ Die **UML** stellt Sprachmittel zur Formulierung zahlreicher **unterschiedlicher Diagramme** zur Verfügung, die zur Beschreibung der verschiedenen Modelle geeignet sind.

- **Modelle und zugeordnete UML-Diagramme**

  ◇ **Modell der Systemnutzung**

  ▻ **Nutzungsfallmodell** (Anwendungsfallmodell) → **Use-Case-Diagramm**

  ◇ **Logisches Modell**

  ▻ **Statisches Modell**

  grobe Systemarchitektur (Aufbaustruktur) → **Paketdiagramm**
  detaillierte Systemarchitektur → **Klassendiagramm**

  ▻ **Dynamisches Modell**

  Zusammenarbeit mehrerer Objekte (Objekt-Interaktion) → Interaktionsdiagramme:
  Sequenzdiagramm
  Kollaborationsdiagramm
  Objektverhalten (in mehreren Use Cases) → **Zustandsdiagramm**
  Systemverhalten → **Aktivitätsdiagramm**

  ◇ **Physikalisches Modell**

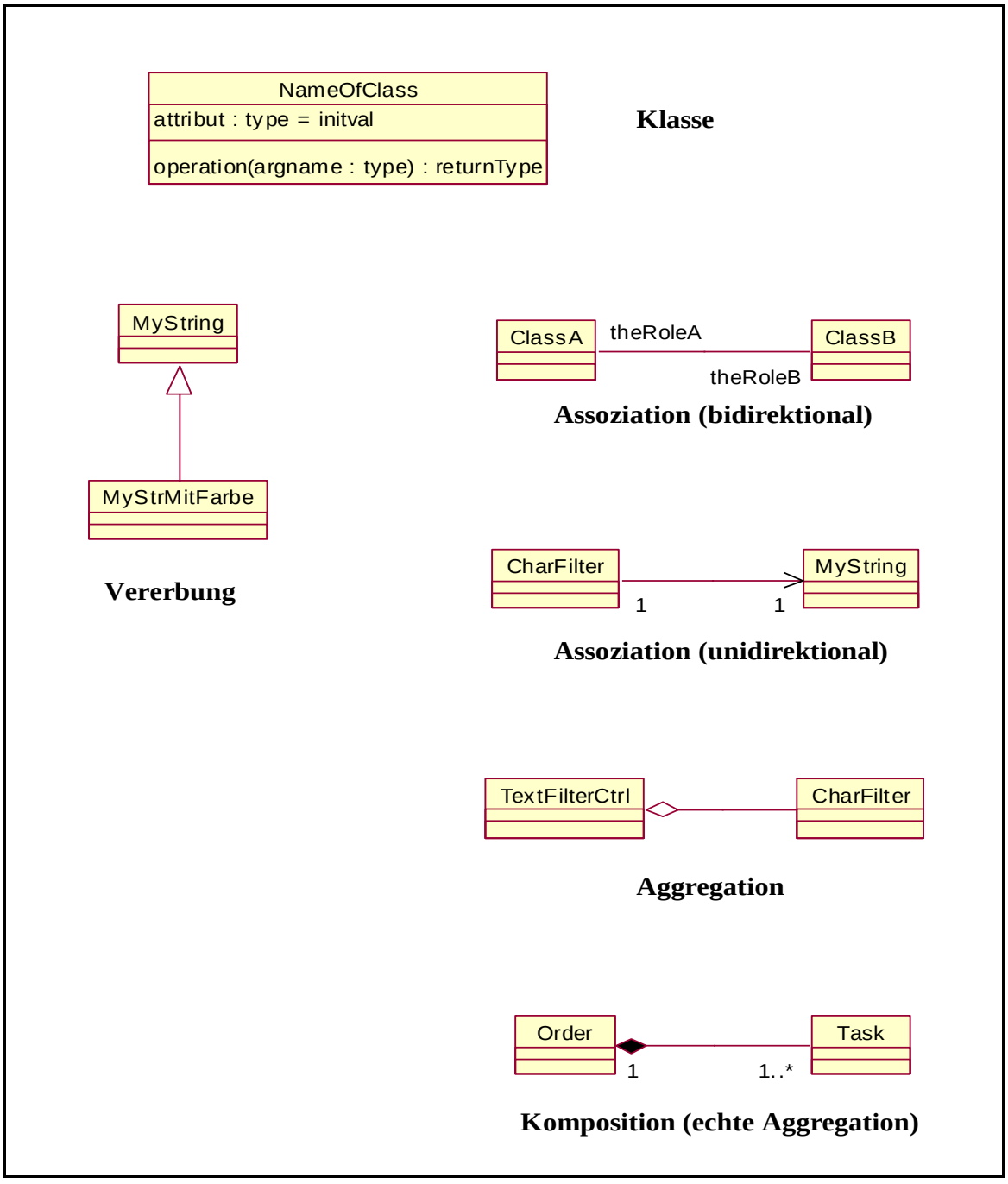
  ▻ **Implementierungsmodell** → **Komponentendiagramm** (Moduldiagramm)

  ▻ **Konfigurierungsmodell** → **Verteilungsdiagramm** (Deployment Diagram)
  (topologische Strukturierung und Verteilung/Zuordnung der Soft- und Hardwarekomponenten des Gesamtsystems)

11.3 Unified Modelling Language (UML)

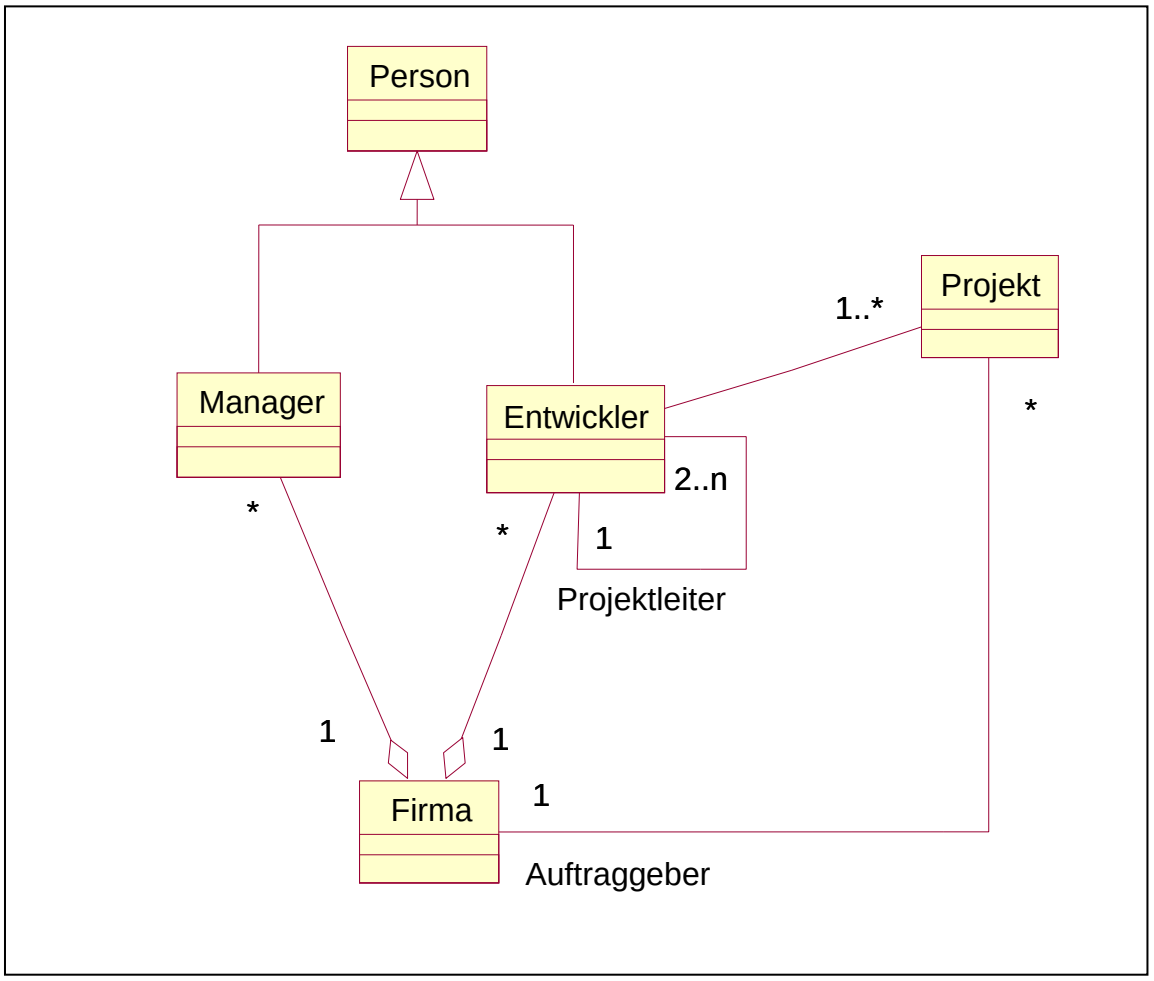
- **Klassendiagramm**
  - ◇ Ein Klassendiagramm beschreibt die im System eingesetzten **Klassen** und ihre statischen **Beziehungen** (Vererbungsbeziehungen und Nutzungsbeziehungen) → **detailliertes statisches Systemmodell**
  - ◇ Zur Erhöhung der Übersichtlichkeit kann die Gesamtheit der Klassen eines Systems auf mehrere **Teildiagramme** aufgeteilt sein.

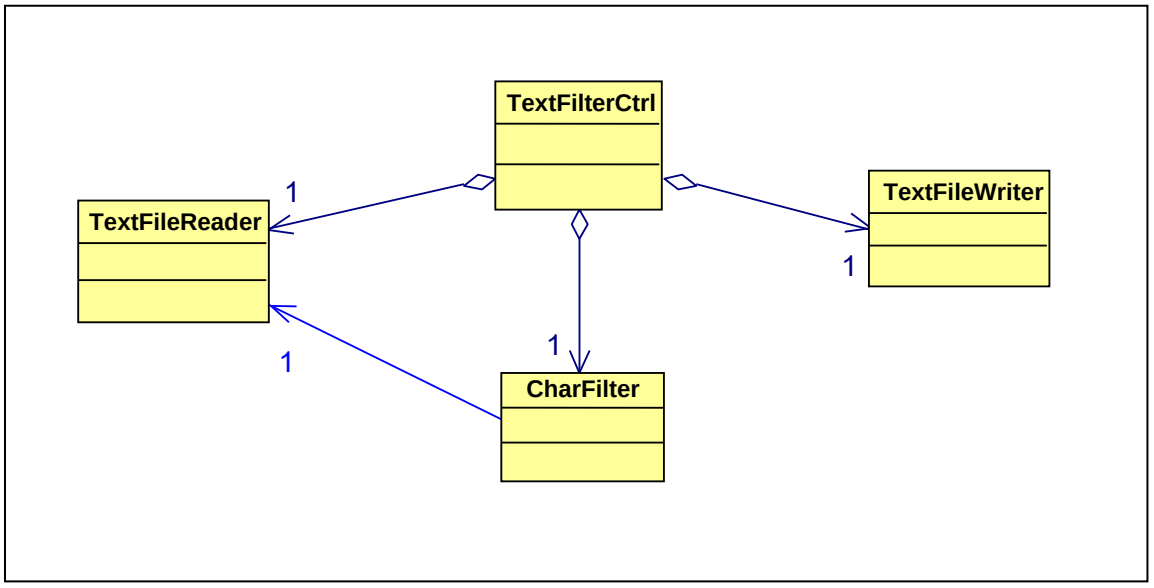
- **Elemente des Klassendiagramms**



Unified Modelling Language (UML) (2)

- **Beispiel für Klassendiagramme**

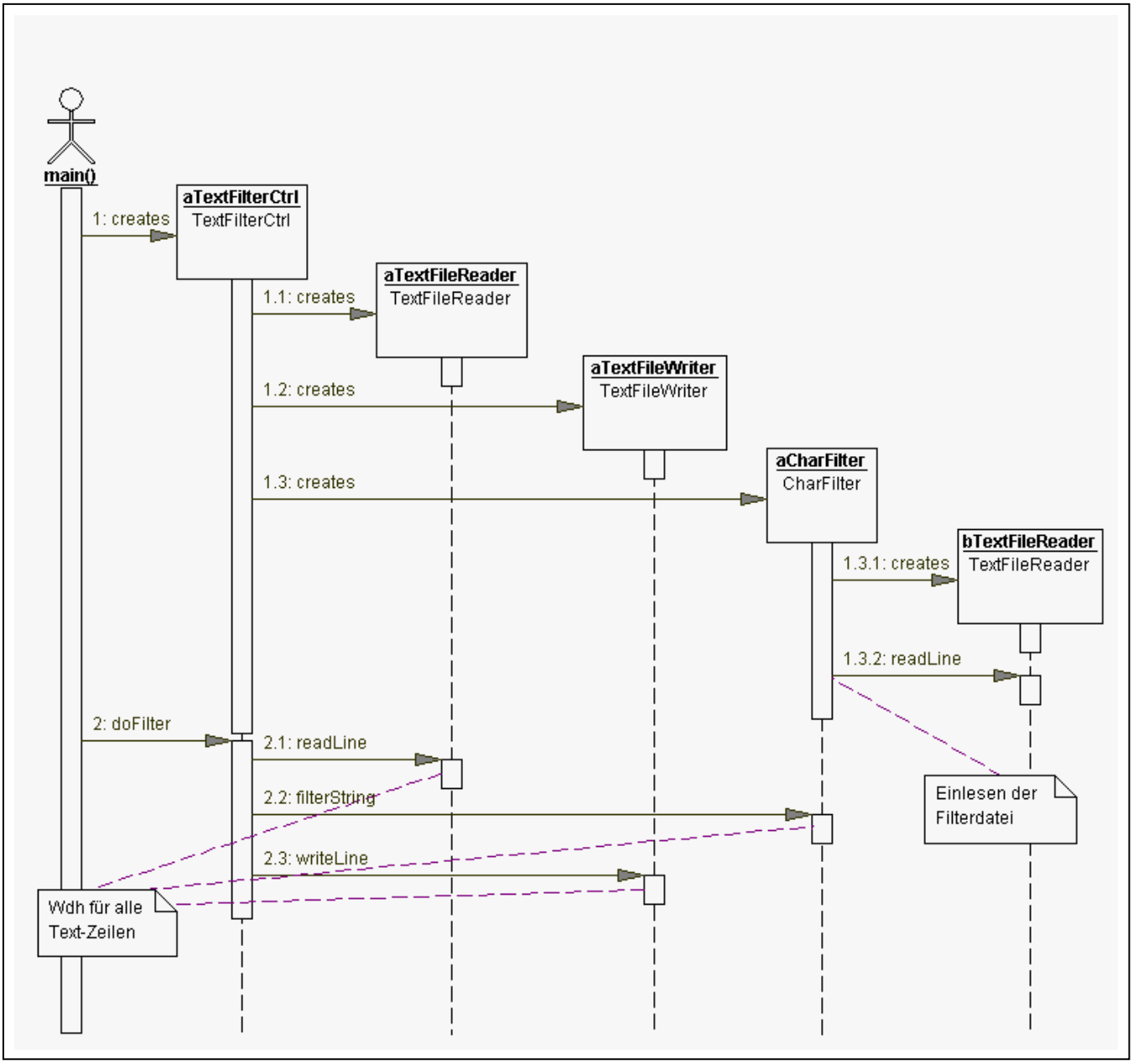




Unified Modelling Language (UML) (3)

- **Sequenzdiagramm**
  - ◇ Ein Sequenzdiagramm beschreibt die **Interaktion mehrerer Objekte**.
  - ◇ Für jedes Scenario jedes Use Cases sollte ein eigenes Diagramm erstellt werden.

- **Beispiel für ein Sequenzdiagramm**



11.4 Technische Hilfsmittel

- **Allgemeines**

  ◇ An jeder der Entwicklungsphasen sind **personelle Ressourcen** (*human resources*) nötig und beteiligt. Zentrales Thema des Softwareentwicklungsprozesses ist deshalb auch die **Kommunikation**, der am Entwicklungsprozess beteiligten Personen.
  Diese erstreckt sich über alle Entwicklungsphasen und Aktivitäten hinweg.

  ◇ Es sollten deshalb **Konventionen** der Kommunikation vorab festgelegt werden, die eine **gemeinsame** Art der Kommunikation für bestimmte Aktivitäten festlegen. Dies geschieht auch durch die Auswahl und Festlegung **technischer Hilfsmittel**.

  ◇ Technische Hilfsmittel zur Unterstützung der personellen Ressourcen sollen deshalb:

  - Die Beteiligten mit **aktuellen** und **relevanten** Informationen des Softwareentwicklungsprozesses versorgen.
  - **Einfach** zu handhaben sein und am Besten in gewissen Punkten auch **automatisierte Mechanismen** zur Verfügung stellen.
  - die nötigen Informationen möglichst **ortsunabhängig** (und über **Zugriffsrechte geregelt**) die nötigen Informationen zur Verfügung zu stellen.

  ◇ Ein gewähltes technisches Hilfsmittel begleitet den Entwicklungsprozess **phasenübergreifend** vom Zeitpunkt des Einsatzes bis zum Ende des Entwicklungsprozesses.

  ◇ Änderung eines technischen Hilfsmittels während eines Entwicklungsprozesses ist möglich und manchmal auch nötig (z. B. Wechsel der Entwicklungsumgebung), oft aber nicht gewünscht.
  ➔ Migration der Daten ➔ Mehraufwand

  ◇ Ziel der Entwicklung solcher Hilfsmittel ist es, so früh wie möglich im Softwareentwicklungsprozess zum Einsatz zu kommen.

  ◇ Solche Produkte werden wie folgt bezeichnet:

  - Projektmanagement-Tools
  - Softwareentwicklungstools
  - CASE-Tools (*Computer-Aided Software Engineering*)

  ◇ Meist umfassen diese aber nicht alle Entwicklungsphasen, deshalb trifft man (besonders im Opensource-Entwicklungsbereich) auf eine Sammlung mehrerer technischer Hilfsmittel.

- **Einige Beispiele solcher unterstützenden technischen Hilfsmittel**

  ◇ **Projektorganisation/-verwaltung**
  (z. B. MS Project, ...)

  - Dieses Werkzeug kommt bereits ab der **Planungsphase** zum Einsatz.
  - Diese Werkzeuge besitzen oft auch automatisierte Mechanismen zur Erstellung von Berichten, Analysen und Präsentationen ➔ **Kommunikation**.

  ◇ **Office-Produkte**
  (z. B. MS Office, ...)

  - Kommen ebenfalls bereits ab der **Planungsphase** zum Einsatz.
  - Die wichtigsten Komponenten sind Textverarbeitung, Präsentationssoftware, Tabellenkalkulation.
  - Dienen zur Kommunikation (Präsentation, Dokumentation) in jeder Entwicklungsphase.
  - Werden inzwischen teilweise durch andere Werkzeuge (Projektorganisationsssoftware, Webpräsentation, u.ä.) abgelöst oder über diese aufgerufen.

Technische Hilfsmittel (2)

- **Einige Beispiele solcher unterstützenden technischen Hilfsmittel (Forts.)**

◇ **Modellierungssoftware**
(z. B.Visual Paradigm, ...)

- Dieses Werkzeug kommt meist ab der **Definitionsphase** (u. U. auch schon in der Planungsphase) zum Einsatz, verliert aber **leider** manchmal in den späteren Entwicklungsphasen (Konstruktion, Auslieferung) an Bedeutung.
- Die gemeinsame Kommunikation findet hier über **UML** statt.

◇ **Versionsverwaltung**
(z. B. Concurrent Versions System (cvs), Subversion (svn), LinCVS, WinCVS, …)
Dient zur zentralen Ablage (Repository) von Quellcode und Dokumentation. Dieses Repository kann **auch** auf einem **entfernten Server** liegen, somit wird der Softwareentwicklungsprozess **ortsunabhängig**.

- Historisch bedingt kommt die Versionsverwaltung meist erst in der **Konstruktionsphase** zum Einsatz, jedoch wäre auch ein früherer Einsatz denkbar (Ablegen von Spezifikationen, Lastenheft, u. ä.) .
- Einzeländerungen können **kommentiert** in das Repository "eingecheckt" werden → **Kommunikation** (Dokumentation).
- Änderungen können wieder rückgängig gemacht werden.
- Frontends zum Zugriff auf die Versionsverwaltung von Dateien werden bereits in Integrierte Entwicklungsumgebungen und Internetdiensten eingebettet.

◇ **Integrierte Entwicklungsumgebung (*IDE*)**
(z. B. Visual Studio, Eclipse, ...)

- Dieses Werkzeug kommen meist ab der **Konstruktionsphase** zum Einsatz.
- Die Entwicklung der IDEs geht aber bereits in Richtung Integration der Modellierung (z. B. über UML), um sie somit bereits in der in der **Definitionsphase** einsetzen zu können.
- Die Kommunikation findet hier meist über **kommentierten Code** und in das Projekt **integrierte Dokumentation** statt (z. B. API-Dokumentation, CHANGELOG, ...).

◇ **Internetdienste**
(z. B. Newsgruppen, Mailinglisten, Wiki, elektronische Foren, Bugtracking-Systeme)

- Gewinnen besonders ab der **Konstruktionsphase** an Bedeutung.
- Werden meist zur **Kommunikation** nach außen eingesetzt → interaktive Informationsplattform für den Anwender (Kunden).
  So z. B. zur **Produktpräsentation** (Anleitungen, HOWTOs, ...) und als **Feedback-Möglichkeit** (Bugreports, Verbesserungsvorschläge, ...).
- Auch unmittelbare Beteiligte des Softwareentwicklungsprozesses nutzen diese Formen der Kommunikation untereinander (z. B. geschlossene Newsgruppen).
  In diesem Fall findet man diese Werkzeuge bereits in **früheren Phasen** der Softwareentwicklung.
- Erreichbarkeit der Beteiligten wird durch die Verwendung dieser Werkzeuge erhöht → **Ortsunabhängigkeit**.

12 Entwurfsmuster (Design Pattern)

12.1. Allgemeines und Überblick

12.2. Beispiele

12.1 Entwurfsmuster – Allgemeines

- **Allgemeines**
  - ◇ **Entwurfsmuster** (**Design Pattern**) sind wiederverwendbare **bewährte** generische **Lösungen** für bestimmte immer wiederkehrende Entwurfsprobleme. Dabei ist charakteristisch, dass in **unterschiedlichen Anwendungsbereichen** auftretende **gleichartige Probleme** durch Anwendung des gleichen Entwurfsmusters **gleichartig gelöst** werden können.
  - ◇ Entwurfsmuster können auf unterschiedlichen Abstraktionsebenen gebildet werden.
    Im engeren Sinn versteht man darunter *Beschreibungen von interagierenden **Objekten und Klassen, die so aufeinander abgestimmt sind, dass sie ein allgemeines Entwurfsproblem in einem bestimmten Kontext lösen*** können
  - ◇ Ein Entwurfsmuster **identifiziert** und **abstrahiert** die **Kern-Aspekte** einer allgemeinen Entwurfsstruktur.
    Es identifiziert und benennt die beteiligten Klassen und Objekte, ihre Rollen, Verantwortlichkeiten und ihre Beziehungen.

> Entwurfsmuster **dokumentieren** erprobte **Entwurfserfahrungen**.
> Sie fördern dadurch in besonderen Maße die **Wiederverwendbarkeit** bewährter Lösungsstrukturen und ermöglichen einen Entwurf auf **höherem Abstraktionsniveau**.
> Ihre Kenntnis und sinnvolle Anwendung **erleichtert** und **beschleunigt** den **Entwicklungsprozess** und **verhindert** gleichzeitig den Einsatz **"schlechterer" Lösungsalternativen**.

- **Beschreibungselemente**
  - ◇ Im Laufe der Zeit sind zahlreiche Entwurfsmuster "entdeckt" und in Büchern und Fachzeitschriften veröffentlicht worden.
    Neue Muster kommen laufend hinzu.
  - ◇ Die Beschreibung eines Entwurfsmusters sollte wenigstens die folgenden vier Elemente umfassen :
    - ▻ **Name des Musters** (*pattern name*)
      Prägnante Zusammenfassung des Entwurfsproblems, seiner Lösung und deren Anwendungskonsequenzen in ein oder zwei Worten.
    - ▻ **Problembeschreibung** (*problem*)
      Darstellung des Anwendungsbereichs des Entwurfsmusters durch Erläuterung des Problems, seines Kontexts und gegebenenfalls spezifischer Entwurfsprobleme
    - ▻ **Problemlösung** (*solution*)
      Beschreibung der Komponenten (Objekte und Klassen) der Entwurfsstruktur, ihrer Verantwortlichkeiten, ihrer Beziehungen und ihrer Zusammenarbeit.
      Dabei bezieht sich die Beschreibung weder auf einen konkreten Entwurf noch auf eine konkrete Implementierung, da ein Entwurfsmuster wie eine Schablone in verschiedenen – aber strukturell ähnlichen - Situationen anwendbar sein soll.
      Entwurfsmuster beruhen auf einer abstrakten Darstellung eines Entwurfsproblems und stellen eine allgemeine Anordnung von Elementen (Klassen und Objekten), die das Problem lösen kann, zur Verfügung.
      Prinzipielle Realisierungsbeispiele – in einer konkreten Implementierungssprache – können gegebenenfalls angegeben werden.
    - ▻ **Anwendungskonsequenzen** (*consequences*)
      Auflistung der Ergebnisse und Folgen ("*trade-offs*"), die sich aus der Anwendung des Entwurfsmusters ergeben.
      Die Kenntnis der Konsequenzen sind für eine kritische Untersuchung von Entwurfsalternativen und die Abwägung von Kosten und Nutzen einer Lösung wichtig.
      Häufig betreffen die Konsequenzen Speicher- und/oder Zeiteffizienz.
      Sie können sich aber auch auf Sprach- und Implementierungsgesichtspunkte beziehen.
      Darüber hinaus ist es grundsätzlich wichtig, den Einfluss eines Entwurfsmusters auf die Flexibilität, die Erweiterbarkeit und die Portabilität eines Systems zu kennen.

12.2 Entwurfsmuster – Klassifizierung

- **Klassifikation**

  ◇ Die verschiedenen bisher veröffentlichten Entwurfsmuster **differieren** bezüglich ihres **Anwendungsbereiches** und ihrer **strukturellen Auflösung** (*granularity*).
  Sie lassen sich nach **unterschiedlichen Gesichtspunkten klassifizieren.**

  ◇ In Anlehnung an das Standardwerk von Gamma/Helm/Johnson/Vlissides ("Design Patterns") ist vor allem eine Klassifizierung nach zwei Gesichtspunkten üblich :
  - ▻ **Einsatzzweck** (*purpose*)
  - ▻ **Geltungsbereich** (*scope*)

  ◇ Der **Einsatzzweck** spiegelt wieder, ***was*** ein Entwurfsmuster ***bewirkt***. Man unterscheidet :
  - ▻ **Erzeugende Muster** (*creational patterns*)
    Sie beziehen sich auf die Erzeugung von Objekten
  - ▻ **Strukturelle Muster** (*structural patterns*)
    Sie befassen sich mit der Zusammensetzung (*composition*) von Klassen und Objekten
  - ▻ **Verhaltensmuster** (*behavioral patterns*)
    Sie bestimmen die Zusammenarbeit zwischen Klassen bzw. Objekten und die Verteilung ihrer Verantwortlichkeiten

  ◇ Der **Geltungsbereich** legt fest, ***worauf*** sich ein Entwurfsmuster primär ***bezieht*** (auf Klassen oder auf Objekte):
  - ▻ **Klassenmuster** (*class patterns*)
    Sie befassen sich primär mit Vererbungsbeziehungen zwischen Klassen.
    Diese Beziehungen sind statisch und liegen zur Compile-Zeit fest.
  - ▻ **Objektmuster** (*object patterns*)
    Sie befassen sich primär mit den Nutzungsbeziehungen zwischen Objekten.
    Diese Beziehungen sind i.a. zur Laufzeit änderbar und daher dynamisch.
    Die meisten Entwurfsmuster gehören dieser Kategorie an.

Klassifizierung der Standard-Entwurfsmuster nach Gamma (u.a.)		**Geltungsbereich**	
		Klassenmuster	**Objektmuster**
Einsatzzweck	Erzeugende Muster	Factory Method	Abstract Factory Builder Prototype Singleton
	Strukturelle Muster	Adapter (class)	Adapter (Object) Bridge Composite Decorator Facade Flyweigth Proxy
	Verhaltensmuster	Interpreter Template Method	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategie Visitor

12.3 **Entwurfsmuster – Überblick**

- **Überblick über die Standard-Entwurfsmuster nach Gamma (u.a.)**

Erzeugende Muster (*Creational Patterns*)

◇ **Abstract Factory** (*Kit*) : Stellt ein Interface zur Erzeugung ganzer Familien verwandter oder voneinander abhängiger Objekte zur Verfügung, ohne dass deren konkrete Klassen spezifiziert werden müssen

◇ **Builder** : Trennt die Konstruktion eines komplexen Objekts von seiner Repräsentation, so dass derselbe Konstruktionsprozess zur Erzeugung unterschiedlicher Repräsentationen eingesetzt werden kann.

◇ **Factory Method** (*Virtual Constructor*): Definiert ein Interface zur Erzeugung eines Objekts, überträgt aber Subklassen die Entscheidung welches konkrete Objekt tatsächlich erzeugt wird.

◇ **Prototype** : Instantiiert Objekt-Prototypen und erzeugt neue Objekte durch Kopieren dieser Prototypen.
Dadurch wird die Erzeugung von erst zur Laufzeit spezifizierter Objekte ermöglicht.

◇ **Singleton** :Stellt sicher, dass nur eine Instanz einer Klasse erzeugt wird und stellt einen globale Zugriffsmöglichkeit zu dieser Instanz zur Verfügung.

Strukturelle Muster (*Structural Patterns*)

◇ **Adapter** (*Wrapper*) : Wandelt das Interface einer Klasse in ein anderes von einem Client erwartetes Interface um. Adapter ermöglichen die Zusammenarbeit von Klassen, die sonst wegen inkompatibler Interfaces nicht zusammenarbeiten könnten.

◇ **Bridge** (*Handle, Body*) : Entkoppelt eine Abstraktion (Klassenhierarchie) von ihrer Implementierung (Klassenhierarchie), so dass beide unabhängig voneinander variiert werden können.

◇ **Composite** (*Recursive Composition*) : Ermöglicht die Repräsentation von Teil-Ganzheits-Beziehungen durch den Aufbau baumartiger Objektstrukturen. Dadurch können Clients individuelle Objekte und Zusammensetzungen von Objekten gleichartig behandeln.

◇ **Decorator** : Ergänzt ein Objekt dynamisch um zusätzliche Fähigkeiten.
Das Entwurfsmuster stellt bezüglich dieser Erweiterung eine flexible Alternative zur Vererbung dar.

◇ **Facade** : Kapselt einen Satz von Interfaces innerhalb eines Subsystems nach außen durch ein vereinfachtes einheitliches Interface. Dadurch wird ein Interface höherer Ebene definiert, das die Benutzung des Subsystems erleichtert.

◇ **Flyweight** : Ermöglicht die Mehrfachnutzung von Objekten mit gleichem inneren Zustand aber unterschiedlichem äußeren Zustand (Kontext). Dadurch werden Strukturen mit einer großen Anzahl "einfacherer" Objekte effektiv unterstützt.

◇ **Proxy** (*Surrogate*) : Stellt einen Platzhalter (Stellvertreter) für ein Objekt zur Verfügung, um den Zugriff zu diesem Objekt zu kontrollieren.

Entwurfsmuster – Überblick (2)

- **Überblick über die Standard-Entwurfsmuster nach Gamma (u.a.)**, Forts.

Verhaltensmuster (*Behavioral Patterns*)

◇ **Chain of Responsibility** : Ermöglicht, dass mehr als ein Objekt die Gelegenheit zur Reaktion auf eine Botschaft erhält. Die möglichen Botschaftenempfänger werden miteinander verkettet und die Botschaft wird die Kette entlang gereicht, bis ein Objekt diese bearbeitet.

◇ **Command** (*Action, Transaction*) : Kapselt eine Botschaft als Objekt. Dadurch werden Sender und Empfänger einer Botschaft voneinander entkoppelt. Dies ermöglicht u.a. die Parametrisierung von Client-Objekten (Anwendungen) mit unterschiedlichen Botschaften für unterschiedliche daraus folgende Operationen.

◇ **Interpreter** : Beschreibt die Definition der Grammatik einer einfachen Sprache sowie die Darstellung von Sätzen in der Sprache zusammen mit einem Interpreter zu ihrer Interpretation.

◇ **Iterator** (*Cursor*) : Ermöglicht den sequentiellen Zugriff zu den Elementen eines Aggregat-Objekts ohne dessen internen Aufbau offenzulegen.

◇ **Mediator** : Definiert ein Objekt, das die Interaktion einer Objektmenge kapselt. Das Entwurfsmuster unterstützt eine lose Objektkopplung durch Verhinderung eines expliziten Bezugs der Objekte aufeinander. Es ermöglicht eine unabhängige Änderung der Objektinteraktion.

◇ **Memento** (*Token*) : Ermöglicht das Festhalten und die äußere Darstellung des inneren Zustands eines Objekts zu seiner späteren Wiederherstellung, ohne die Objektkapselung zu verletzen.

◇ **Observer** (*Dependents, Publish-Subscribe*) : Ermöglicht die dynamische Registrierung von Objektabhängigkeiten, so dass bei einem Zustandswechsel eines Objekts alle von ihm abhängigen Objekte benachrichtigt werden.

◇ **State** (*Object for States*) : Kapselt die Zustände eines Objekts in separate Objekte, die jeweils zu einer Zustandsklasse gehören, die von einer gemeinsamen abstrakten Basis-Zustandsklasse abgeleitet sind. Dieses Entwurfsmuster ermöglicht einem Objekt, sein Verhalten bei einer Zustandsänderung zu ändern. Nach außen scheint das Objekt dadurch seine Klasse zu ändern.

◇ **Strategy** (*Policy*) : Kapselt verwandte Algorithmen in jeweils einer eigenen Klasse, die von einer gemeinsamen Basisklasse abgeleitet ist. Dies ermöglicht die Auswahl und Änderung der Algorithmen zur Laufzeit ohne dass Änderungen an den Clients, die sie benutzen, notwendig sind.

◇ **Template Method** : Definiert das Gerüst eines Algorithmus in einer abstrakten Basisklasse, wobei die Konkretisierung einiger Algorithmus-Schritte an abgeleitete Klassen übertragen wird, ohne dass dadurch die Struktur des Algorithmus geändert werden muss.

◇ **Visitor** : Implementiert eine Operation, die mehrere Objekte in einer komplexen Struktur betrifft, in einem separaten Objekt. Das Entwurfsmuster ermöglicht die Definition neuer Operationen ohne Änderung der Klassen der Objekte, auf die sich die Operation bezieht.

12.4 Entwurfsmuster : Singleton

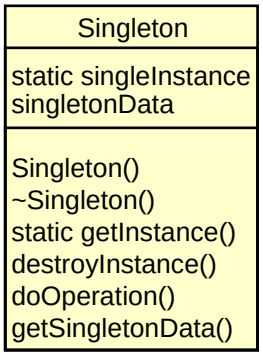
- **Name : Singleton**

- **Kurzbeschreibung**
  Stellt sicher, dass **nur eine Instanz** einer Klasse erzeugt wird und stellt einen **globale Zugriffsmöglichkeit** zu dieser Instanz zur Verfügung.

- **Problembeschreibung**
  Oft ist es wichtig, dass von einer Klasse nur eine einzige Instanz erzeugt wird.
  Zusätzlich besteht häufig die Forderung, dass diese Instanz global zugreifbar sein soll.
  Beispielsweise sollte in einem zentralisierten Workflow-Managementsystem nur ein Manager-Objekt vorhanden sein.

- **Problemlösung**
  In einer sinnvollen Lösung ist die Klasse selbst dafür verantwortlich, dass sie nur einmal instantiiert wird.
  Dies wird dadurch erreicht, dass alle Konstruktoren der Klasse privat sind
  Zur Objekterzeugung und zum Zugriff auf das Singleton-Objekt dient eine öffentliche statische Memberfunktion.
  Diese überprüft zuerst, ob bereits eine Instanz der Klasse angelegt ist.
  Falls nein wird eine Instanz – durch Aufruf eines privaten Konstruktors - angelegt und ein Pointer/Referenz auf diese in einer privaten statischen Membervariablen abgelegt. Dieser Pointer wird gleichzeitig als Funktionswert zurückgegeben.
  Falls ja, gibt sie den in der statischen Membervariablen gespeicherten Pointer/Referenz auf die Instanz zurück.
  Anbei Klassendiagramm für C++

- **Struktur**



- **Anwendungskonsequenzen**
  - ▻ Da die Singleton-Klasse ihre einzige Instanz kapselt, hat sie die strikte Kontrolle über den Zugriff zu dieser
  - ▻ Die Singleton-Klasse kann leicht zu einer Klasse erweitert werden, die eine definierte Anzahl von Instanzen größer als eins erzeugen kann.
  - ▻ Der Einsatz einer Singleton-Klasse als Basisklasse kann problematisch sein.
    In diesem Fall darf der Konstruktor nicht privat sein. Außerdem lässt sich die statische Objekterzeugungs-Funktion nicht überschreiben.

Entwurfsmuster : Singleton (2)

- **Implementierungsbeispiel**
  - ◇ **Definition der Singleton-Klasse :**

```
class Singleton
{  public :
     static Singleton* getInstance();
     void destroyInstance();
     // weitere nicht-statische Memberfunktionen
   private :
     static Singleton* singleInstance;
     // weitere nicht-statische Datenkomponenten
     Singleton();
     ~Singleton();
```

  - ◇ **Implementierung der Singleton-Klasse :**

```
Singleton* Singleton::singleInstance = NULL;   // == NULL-Pointer

Singleton::Singleton()
{  // privater Konstruktor }

Singleton::~Singleton()
{  // privater Destruktor }

Singleton* Singleton::getInstance()
{  if (singleInstance==NULL){
     singleInstance = new Singleton;
     cout << "Instanz von Singleton erzeugt" <<endl;
   }else cout << "Instanz von Singleton bereits vorhanden" <<endl;

   return singleInstance;
}

void Singleton::destroyInstance()
{ if(singleInstance!=NULL){
    delete this;
    singleInstance=NULL;       // == NULL-Pointer
   `
```

  - ◇ **Applikation der Singleton-Klasse :**

```
int main()
{
          Singleton* si_1 = Singleton::getInstance();
          Singleton* si_2 = Singleton::getInstance();
          si_1->destroyInstance();
          si_2->destroyInstance();
          return 0;
}
```

Ergebnis: `Instanz von Singleton erzeugt`
`Instanz von Singleton bereits vorhanden`

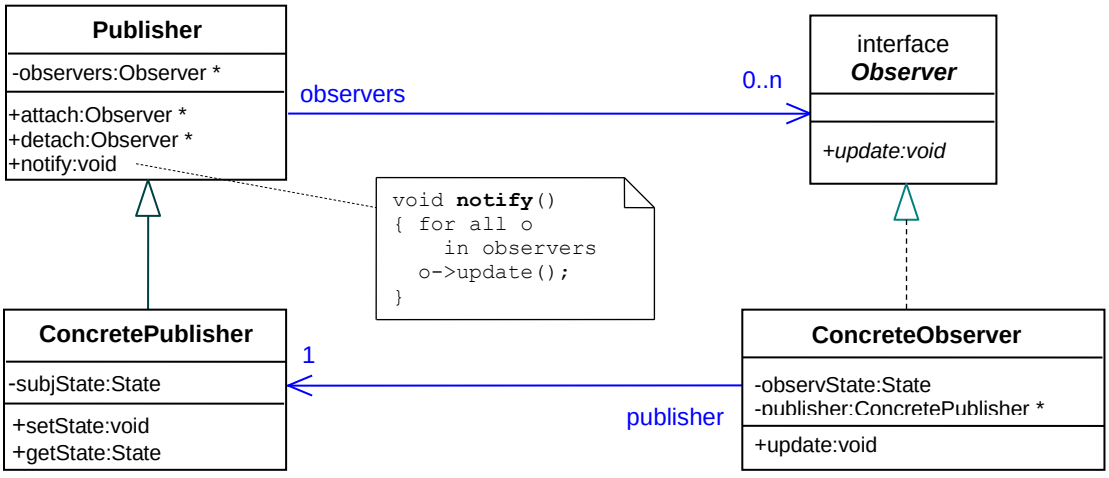
12.5 Entwurfsmuster : Observer

- **Name : Observer** (*Publish-Subscribe*, *Dependents*)
- **Kurzbeschreibung**
  Ermöglicht die **dynamische Registrierung** von **Objektabhängigkeiten**, so dass bei einem **Zustandswechsel** eines Objekts alle von ihm **abhängigen Objekte benachrichtigt** werden.

- **Problembeschreibung**
  In vielen Anwendungsbereichen soll sich der Zustandswechsel eines Objekts direkt auf den Zustand bzw. das Verhalten anderer Objekte auswirken.
  Beispiel : Ein Datenbestand und seine – u.U. gleichzeitige – Darstellung in verschiedenen Formaten in einem GUI-System (z. B. Tabelle, Balkendiagramm, Tortendiagramm). Jede Änderung des Datenbestandes muss sich umgehend auf alle Darstellungen auswirken.
  Eine enge Kopplung der beteiligten Objekte (→ die beteiligten Objekte haben voneinander Kenntnis) ist meist nicht wünschenswert, da dadurch die unabhängige Verwendung und Modifikation ihrer jeweiligen Klassen stark eingeschränkt wird. Außerdem müssen in diesem Fall die Abhängigkeiten bereits zur Compilezeit bekannt sein, was eine flexible dynamische Anpassung zur Laufzeit verhindert.

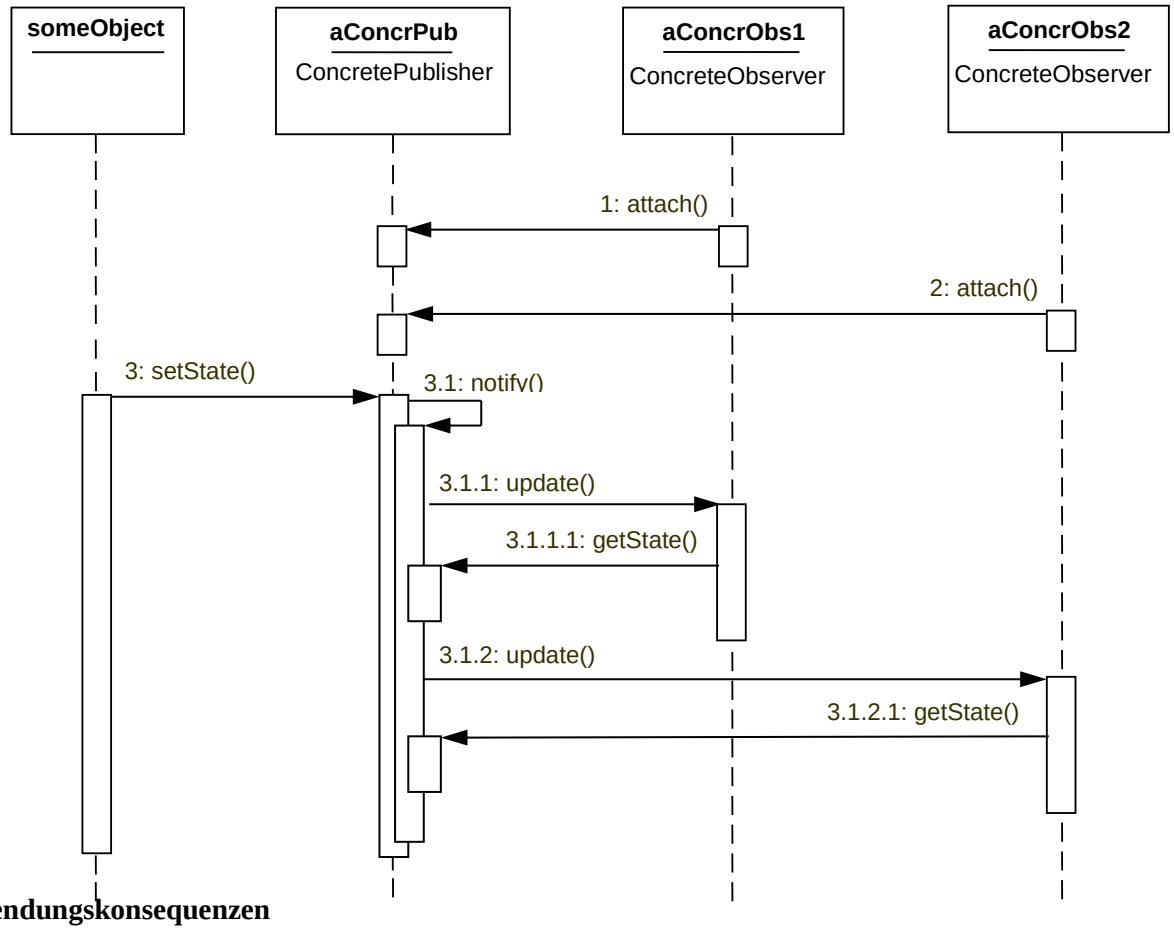
- **Problemlösung**
  Bei dem Objekt, von dessen Zustand andere Objekte abhängen (***Publisher*-Objekt**), wird eine **Liste der abhängigen Objekte** geführt. In diese Liste tragen sich alle Objekte, die über den Zustand dieses Objekts auf dem Laufenden gehalten werden wollen (weil sie von ihm abhängen), ein (***Subscriber*-Objekte**, ***Observer*-Objekte**).
  Bei einem **Wechsel** seines Zustands **informiert** das *Publisher*-Objekt alle eingetragenen *Observer*-Objekt hierüber (***notify***). Diese können dann den neuen Zustand vom *Publisher*-Objekt ermitteln und entsprechend der eingetretenen Änderung reagieren (z.B. ihren eigenen Zustand an den Zustand des *Publisher*-Objektes anpassen).
  Die Anzahl der *Observer*-Objekte muss dem *Publisher*-Objekt a priori nicht bekannt sein. Bei ihm können sich zur Laufzeit beliebig viele *Observer*-Objekte an- bzw. auch wieder abmelden.
  Publisher-Objekt und *Observer*-Objekte sind von einander entkoppelt und können unabhängig voneinander modifiziert werden.
  Die ***Publisher*-Schnittstelle** wird durch eine geeignete Klasse (**`Publisher`**) zur Verfügung gestellt. Diese Klasse dient als **Basisklasse** für konkrete *Publisher*-Klassen (**`ConcretePublisher`**), die die Datenkomponenten für den jeweiligen Zustand und die Methoden zum Setzen und Ermitteln derselben bereitstellen.
  Das **Interface** zum Informieren von *Observer*-Objekten wird durch eine abstrakte Klasse (***`Observer`***) definiert
  Dieses Interface wird von konkreten *Observer*-Klassen (**`ConcreteObserver`**) implementiert. Objekte dieser Klassen enthalten – neben ihren eigentlichen Zustands-Datenkomponenten – eine Referenz auf das konkrete *Publisher*-Objekt, bei dem sie sich eingetragen haben.

- **Struktur**



Entwurfsmuster : Observer (2)

- **Ablauf der Interaktion**



- **Anwendungskonsequenzen**
  - ▻ ***Publisher*** und ***Observer*** können **unabhängig** voneinander **geändert** und **ausgetauscht** werden.
    Publisher können wiederbenutzt werden, ohne ihre *Observer* wiederzubenutzen und umgekehrt.
    Observer können hinzugefügt werden, ohne den *Publisher* oder andere *Observer* zu verändern.
  - ▻ Die **Kopplung** zwischen *Publisher* und *Observer* ist **abstrakt** und **minimal**.
    Ein *Publisher*-Objekt weiß lediglich, dass es eine Liste von *Observer*-Objekten besitzt, die das *Observer*-Interface implementieren. Es kennt nicht die konkrete Klasse seiner *Observer*.
  - ▻ *Observer*-Objekte können gegebenenfalls selbst auch einen Zustandswechsel beim *Publisher*-Objekt bewirken
  - ▻ Die Information über den Zustandswechsel (**Notifikation**) durch das *Publisher*-Objekt ist automatisch eine **Broadcast-Kommunikation**. Es werden immer alle eingetragenen *Observer*-Objekte informiert, unabhängig davon, wieviel es sind. Es obliegt einem *Observer*-Objekt, eine Notifikation gegebenenfalls zu ignorieren.
  - ▻ Falls **sehr viele** *Observer*-Objekte eingetragen sind, kann eine Notifikation und der dadurch ausgelöste *Observer*-Update u.U. eine **längere Zeit** dauern.
  - ▻ Da das eine Zustandsänderung bewirkende Objekt keine Information über die *Observer* und die mit diesen verbundenen "Update"-Kosten hat, können **"leichtfertige" Zustandsänderungen** einen erheblichen zeit- und damit kostenintensiven **Aufwand** bewirken.
  - ▻ Das **einfache Notifikations-Protokoll** liefert **keine Informationen** darüber, **was** sich im Publisher-Objekt **geändert** hat. Dies festzustellen, obliegt dem Observer-Objekt, was den Update-Aufwand gegebenenfalls noch erhöht.
    Als Alternative kann u.U. ein **erweitertes Protokoll** definiert werden, dass detailliertere Zustandsänderungs-Info liefert.
  - ▻ Es gibt Fälle, bei denen sinnvoll ist, dass sich **ein *Observer***-Objekt bei **mehreren *Publisher***-Objekten für eine Notifikation einträgt. In diesen Fällen muss die Update-Botschaft eine Information über das absendende Publisher-Objekt enthalten

Entwurfsmuster : Observer (3)

- **Implementierungsbeispiel** (Teil1)
  Achtung: Beispiel ist nicht vollständig und soll nur die prinzipielle Programm-Struktur zeigen
  - ◇ **Definition einer Publisher-Basisklasse** (`Publisher`)

```
class Publisher
{ public :
    virtual ~Publisher() {};
    virtual void attach(Observer*);
    virtual void detach(Observer*);
    virtual void notify();
  protected :
    Publisher() {};
  private :
    Set<Observer*> observs;       // Liste der eingetragenen Observer-Objekte
};
```

  - ◇ **Implementierung der Klasse `Publisher`**

```
void Publisher::attach(Observer* no)
{ observs.insert(no);
}

void Publisher::detach(Observer* no)
{ observs.remove(no);
}

void Publisher::notify()

{ Observer** ppo;
  for(ppo=observs.first(); ppo; ppo=observs.next())
    (*ppo)->update(this);
}
```

  - ◇ **Definition einer konkreten Publisher-Klasse** (`ConcretePublisher`)

```
class ClockTimer : public Publisher
{ public :
    ClockTimer();
    ~ClockTimer();
    int getHour();
    int getMinute();
    int getSecond();
    void tick();          // Veränderung des Objekt-Zustands
  private :
    // Datenkomponente(n) zur Speicherung der Zeit
};
```

  - ◇ **Implementierung der Klasse `ClockTimer`**

```
// Implementierung der uebrigen Memberfunktionen
void ClockTimer::tick()
{
  // update der privaten Datenkomponente(n) zur Speicherung der Zeit
  notify();
}
```

Entwurfsmuster : Observer (4)

- **Implementierungsbeispiel** (Teil2)

  ◇ **Definition einer abstrakten Observer-Basisklasse** (*Observer*)

```
class Publisher;

class Observer
{ public :
    virtual ~Observer() { };
    virtual void update(Publisher*) = 0;
  protected :
    Observer() {};
};
```

  ◇ **Definition einer konkreten Observer-Klasse** (ConcreteObserver)

```
class Widget;      // Fenster-Basisklasse mit graph. Fähigkeiten
class ClockTimer;

class DigitalClock : public Observer, public Widget
{ public :
    DigitalClock(ClockTimer*);
    ~DigitalClock();
    void update(Publisher*);
    void draw();             // virtuelle Methode von Widget
  private :
    ClockTimer* m_publ;
};
```

  ◇ **Implementierung der Klasse `DigitalClock`**

```
DigitalClock::DigitalClock(ClockTimer* ct)
{ m_publ=ct;
  m_publ->attach(this);
}

DigitalClock::~DigitalClock()
{ m_publ->detach(this);
}

void DigitalClock::update(Publisher* pub)
{ if (pub==m_publ)
    draw();
}

void DigitalClock::draw()             // virtuelle Methode von Widget
{ int h = m_publ->getHour();
  int m = m_publ->getMinute();

  // Zeichnen der Digital-Uhr
}
```

◇ **Beispiel für Anwendungscode** (Auszug)

```
ClockTimer timer;
DigitalClock digClock(&timer);
// bei jedem Aufruf von timer.tick() stellt digClock die neue Zeit dar
```

12.6 Entwurfsmuster: Composite

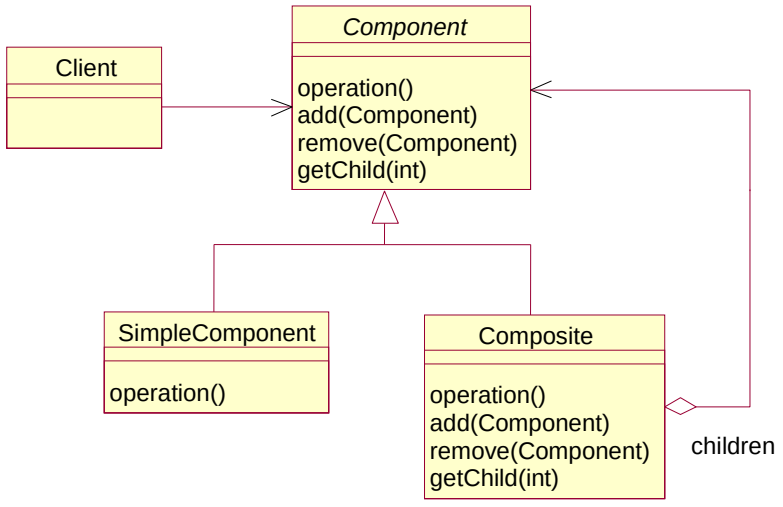
- **Name : Composite** (*Recursive Composition*)

- **Kurzbeschreibung**
  Ermöglicht die Repräsentation von Teil-Ganzheits-Beziehungen durch den Aufbau baumartiger Objektstrukturen. Dadurch können Clients individuelle Objekte und Zusammensetzungen von Objekten gleichartig behandeln.

- **Problembeschreibung**
  In vielen Anwendungsbereichen treten Strukturen auf, die aus einfachen Objekten und zusammengesetzten Objekten (Containern) aufgebaut sind. Dabei können i.a. die zusammengesetzten Objekte wiederum zusammengesetzte Objekte als Komponenten enthalten ⇒ Rekursive Teil-Ganzheits-Hierarchien.
  (Beispiel : Fenster-System → Ein Fenster enthält u.a. Text, einfache Steuerungselemente und "Unter"-Fenster).
  Es ist sinnvoll und effektiv, wenn Code, der die entsprechenden Klassen benutzt (Client), einfache Objekte und zusammengesetzte Objekt völlig gleichartig behandeln kann, d.h. beim Aufruf von Komponenten-Methoden nicht zwischen einfachen und zusammengesetzten Objekten unterscheiden muß.

- **Problemlösung**
  Die Klassen für einfache Komponenten und für zusammengesetzte Komponenten werden von einer gemeinsamen abstrakten Basisklasse **`Component`** abgeleitet.
  Diese definiert sowohl das "allgemeine" Komponenten-Anwendungs-Interface als auch das spezielle "Management"-Interface für zusammengesetzte Komponenten. Von der gemeinsamen Komponenten-Basisklasse **`Component`** können durchaus mehrere Klassen für einfache Komponenten abgeleitet sein (im untenstehenden Klassendiagramm nur die Klasse **`SimpleComponent`**).
  Die Klasse für zusammengesetzte Komponenten **`Composite`** kann selbst wieder eine (abstrakte) Basisklasse für mehrere (konkrete) Composite-Klassen sein.
  Die Klasse für zusammengesetzte Komponenten **`Composite`** verwaltet ihre enthaltenen Objekte Kind-komponenten, "children") als Instanzen der gemeinsamen Komponenten-Basisklasse **`Component`.**
  An sie gerichtete Anwendungs-Methodenaufrufe ("`operation()`") delegiert sie an alle in ihr enthaltenen Komponenten.

- **Struktur**



- **Anwendungskonsequenzen**
  - ▻ Client-Code kann immer dann, wenn er ein einfaches Objekt erwartet auch mit einem zusammengesetzten Objekt arbeiten.
  - ▻ Da Clients einfache und zusammengesetzte Objekte gleich behandeln können, vereinfacht sich der Client-Code (z. B. keine `switch-case`-Anweisungen zu ihrer Unterscheidung notwendig).
  - ▻ Neue Komponenten-Klassen können einfach ohne Änderungen des Client-Codes hinzugefügt werden.
  - ▻ Andererseits ist es schwierig, Kindkomponenten auf bestimmte Typen zu begrenzen (→ zusätzliche Laufzeitüberprüfungen notwendig).

- **Implementierungsbeispiel** (Teil 1)
  - ◇ **Definition einer Komponenten-Basisklasse (`Component`) :**

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

//Abstrakte Klasse
class Equipment
{
  public :
    virtual ~Equipment (){};  //irb
    const char* getName() const { return m_szName; }
    virtual double getPrice() const = 0;
    // weitere Anwendungs-Memberfunktionen
    //Default-Implementierung, sinnvolle Imlementierung nur in Klasse CompositeEquipment
    //Vorteil: Funktionsaufruf kann für jedes Element der Liste gemacht werden
    //Anlternativ könnten add und remove nur für composite_equipment definiert werden
    virtual void add(Equipment*){cerr << "Call add() isn't allowed! " << endl;}
    virtual void remove(Equipment*)
    {cerr << typeid(*this).name()<<":" << "Call remove(Equip*) isn't allowed! " << endl;}

  protected:
     Equipment(const char* name) : m_szName(name){}

  private :
    const char* m_szName;
};
#endif
```

  - ◇ **Definition einer einfachen Komponentenklasse (`SimpleComponent`) :**

```
#ifndef _BASICPART_H
#define _BASICPART_H
#include "equipment.h"

class BasicPart : public Equipment
{
  public :
    BasicPart(const char*, double);
    ~BasicPart(){};
    virtual double getPrice() const;
    // weitere Anwendungs-Memberfunktionen
  private :
    double m_dPrice;
};
#endif
```

◇ **Definition einer zusammengesetzten Komponentenklasse (`Composite`) :**

```
#ifndef _COMPOSITE_EQUIPMENT_H
#define _COMPOSITE_EQUIPMENT_H
#include "equipment.h"
#include <list>
using namespace std;

typedef list<Equipment*> ListEquip;
class CompositeEquipment : public Equipment
{
  public :
    CompositeEquipment(const char*);
    virtual double getPrice() const;
    // weitere Anwendungs-Memberfunktionen
    virtual void add(Equipment*);
    virtual void remove(Equipment*);
   private :
    ListEquip m_pclEquipment;
};

#endif
```

- **Implementierungsbeispiel** (Teil 2)

◇ **Implementierung der Klasse `BasicPart` (Auszug) :**

```
#include "basicpart.h"

BasicPart::BasicPart(const char* name, double price) : Equipment(name)
{
  m_dPrice=price;
}

double BasicPart::getPrice() const
{
  return m_dPrice;
}
```

◇ **Implementierung der Klasse `CompositeEquipment`:**

```
#include "composite_equipment.h"
#include <iterator>
using namespace std;

CompositeEquipment::CompositeEquipment(const char* name) :
  Equipment(name) { }

double CompositeEquipment::getPrice() const
{
  ListEquip::const_iterator it = m_pclEquipment.begin();
  double sum=0.0;
  for( ; it != m_pclEquipment.end(); ++it){
         sum+=(*it)->getPrice();  //Rekursion
  }
  return sum;
}

void CompositeEquipment::add(Equipment* pEqui){
       if(pEqui != NULL)
              m_pclEquipment.push_back(pEqui);
}

void CompositeEquipment::remove(Equipment* pEqui) {
        m_pclEquipment.remove(pEqui);  //Zeiger aus Liste entfernen
        cout << "Objekt " <<pEqui->getName() << " entfernt" <<endl;
}
```

◇ **Beispiel für Anwendungscode :**

```
include "equipment.h"
#include "basicpart.h"
#include "composite_equipment.h"

#include <iostream>
using namespace std;

int main()
{
        CompositeEquipment* cabinet  = new CompositeEquipment("PC Cabinet");
        CompositeEquipment* motherboard =  new CompositeEquipment("PC Motherboard");

        BasicPart* emptyCabinet  = new BasicPart("Cabinet (empty)", 110.00);
        BasicPart* emptyBoard     = new BasicPart("ASUS-Board Pentium II 350", 370.00 );


        cabinet->add(emptyCabinet);
        cabinet->add(motherboard);
        motherboard->add(emptyBoard);

        CompositeEquipment* bus = new CompositeEquipment("PCI-Bus");
        bus->add(new BasicPart("ATI Graphic Card", 50.00));
        motherboard->add(bus);
        motherboard->add(new BasicPart("3.5 Floppy Disc", 65.00));

        cout << "The total price is : " << cabinet->getPrice() << endl;

        //Zusatztest stückweiser Abbau
        /*
        motherboard->remove(emptyBoard);
        cout << "The total price is : " << cabinet->getPrice() << endl<<endl;
       cabinet->remove(motherboard);
        cout << "The total price is : " << cabinet->getPrice() << endl<<endl;
        cabinet->remove(emptyCabinet);
        cout << "The total price is : " << cabinet->getPrice() << endl<<endl;
        */

        cout << "fertig" <<endl;
        //Speicher wird durch Programmende freigegeben

        return 0;
}
```

ANHANG

Präzedenztabelle der Operatoren in C++

Die folgende Tabelle gibt Auskunft über den Vorrang von Operatoren gegenüber anderen.
Je niedriger die Rangnummer desto höher der Vorrang in der Auswertung. Operatoren mit gleicher Rangnummer besitzen gleiche Präzedenz. Die **Assoziativität** gibt an in welcher Reihenfolge die Operatoren gleicher Präzedenz ausgewertet werden.

Rang	Operatoren		Assoziativität
1	:: ()	Scope Resolution Operator (binär und unär) Klammerung	links → rechts
2	. -> [] *functionname*() *Lval*++ *Lval*-- **typeid**() *type*() **dynamic_cast**<*type*>() **static_cast**<*type*>() **reinterpret_cast**<*type*>() **const_cast**<*type*>()	Objektelement-Operatoren Indexoperator Funktionsaufruf Postfix-Operatoren Typobjekt Wertkonstruktion/ Typkonvertierungen	links → rechts
3	**new** **new[]** **delete** **delete[]** **sizeof** ++*Lval* --*Lval* ~ ! - + & * (*type*)	Dynamische Speicherallokation Dynamische Speicherdeallokation Objekt-/Typgrößenermittlung Präfix-Operatoren Unäre Modifikatoren Adressermittlung/Dereferenzierung C-Typkonvertierung	rechts → links
4	->* .*	Dereferenzierte Pointer auf Klassenkomponenten (Komponentenauswahloperatoren)	links → rechts
5	* / %	Arithmetische Operatoren	links → rechts
6	+ -	Arithmetische Operatoren	links → rechts
7	<< >>	Schiebeoperatoren	links → rechts
8	< <= > >=	Vergleichsoperatoren	links → rechts
9	== !=	Vergleichsoperatoren	links → rechts
10	&	Binäres Und	links → rechts
11	^	Binäres Exklusiv-Oder	links → rechts
12	|	Binäres Oder	links → rechts
13	&&	Logisches Und	links → rechts
14	||	Logisches Oder	links → rechts
15	? : *(1. Operand)*	bedingter Auswerteoperator	links → rechts
16	= *= /= %= += -= <<= >>= &= |= ^=	Zuweisungsoperatoren	rechts → links
17	**throw**	Werfen von Ausnahmen	links → rechts
18	,	Sequentieller Auswerteoperator	links → rechts
19	? : *(2. und 3. Operand)*	bedingter Auswerteoperator	links → rechts

- ◇ Die Reihenfolge der Auswertung von **Operanden eines Operators** ist undefiniert!
- ◇ Die Tabelle ist nicht Teil des ISO-C++-Standards. Sie ergibt sich aus den festgelegten Grammatikregeln.
- ◇ Das logische Und (&&) und das logische Oder (||) werden nur so weit erforderlich von links nach rechts ausgewertet.