Lehrstuhl für
Datenverarbeitung

Projektkurs C++

Objektorientiertes Programmieren in C++

Sommersemester 2005

Neue Auflage nach dem ANSI C++ Standard

Technische Universität München, Arcisstraße 21, D-80333 München
Innenhof, Gebäude 9, Sekretariat Z946, Tel. 089/289-23601

1 Vorbemerkungen

C++ makes it much harder to shoot yourself in the foot, but when you do, it blows off your whole leg.

- Bjarne Stroustrup

In C we had to code our own bugs. In C++ we can inherit them.

-Prof. Gerald Karam

Zu diesem Skript:

Dieses Skript dient der Vorlesungsbegleitung und soll eine Unterstützung beim Durcharbeiten der Vorlesung sein. Es ist kein C++-Lehrbuch und erhebt keinerlei Anspruch, sämtliche Aspekte des Themas umfassend abzudecken.

Ursprünglich wurde es von den ehemaligen Mitarbeitern des Lehrstuhls Manfred Kräss, Volkmar Pleßer und Andreas Rinkel unter Zuhilfenahme zahlreicher Unterlagen und Bücher, die im Literaturverzeichnis aufgelistet sind, erstellt

Von 1997 bis 2005 ist das Skript laufend den Vorlesungen von Peter Krauß angepasst worden. Eine größere Änderung war die Verwendung der UML-Notation, die sich mittlerweile zu einem Standard entwickelt hat.

Der aktuelle ANSI-Standard für C++ aus dem Jahr 2004 erforderte eine grundlegende Überarbeitung. Vor allem sind viele Code-Beispiele eingefügt worden, um die grundlegenden Neuerungen leichter in die Praxis umsetzen zu können. Ob ein Compiler dem neuen Standard entspricht, kann man daran erkennen, dass er `#include <iostream>` (bisher `#include <iostream.h>`) akzeptiert.

Für Verbesserungsvorschläge und Korrekturhinweise sind wir jederzeit sehr dankbar.

Zur Arbeitsumgebung:

Die Projektarbeit erfolgt auf dem Workstation-Cluster des Lehrstuhls für Datenverarbeitung (Raum -1981, im Keller). Zusätzlich kann man auch im Turm arbeiten - soweit Platz ist.

2 Einführung und Software-Engineering

2.1 Zielsetzung und Grenzen der Objektorientierung

Erwartungen an das objektorientierte Paradigma

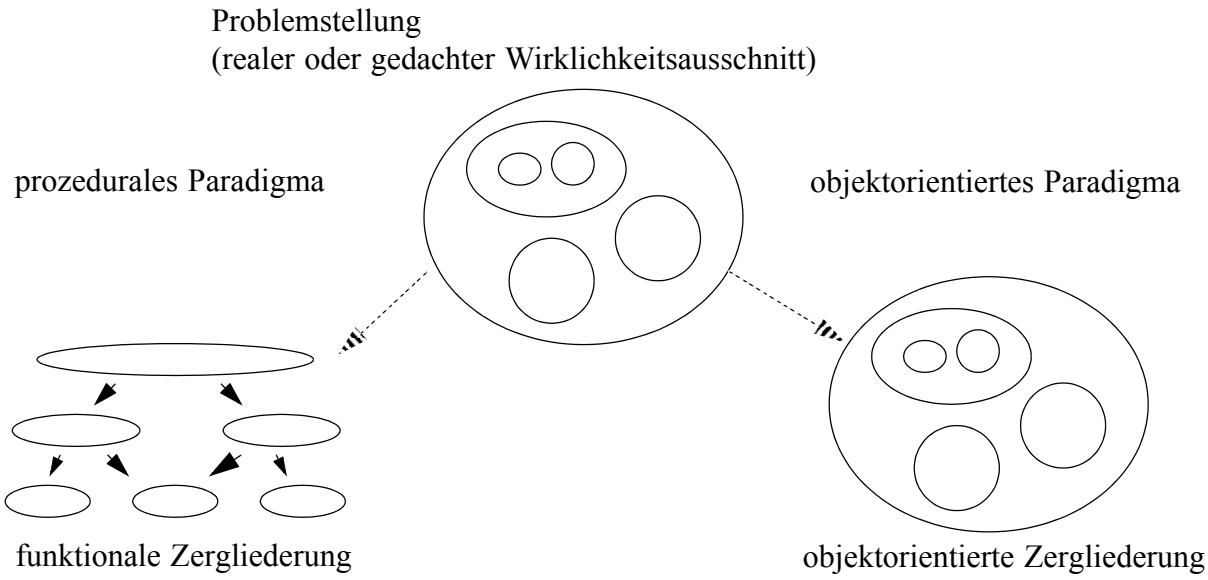
- Wiederverwendbarkeit
- Erweiterbarkeit
- Leichte Wartbarkeit
- ...

Aber:

- Objektorientierte Programmiersprachen und Entwicklungsumgebungen erzeugen für sich noch keine solche Software.
- *... object-orientation is not a magic formula to ensure reusability, however. Reuse does not just happen; it must be planed by thinking beyond the immediate application and investing extra effort in a more general design* ([Rumbaugh 93], zitiert nach [Schäfer 94])
- Hybride Sprachen wie C++ (Obermenge von C) erlauben zudem rein prozeduralen Programmierstil.

2.2 Anschauungsweisen (Paradigmen)

Prozedurale und objektorientierte Anschauungsweise



Grundgedanke des objektorientierten Paradigmas:

- Entwicklung von Systemen nicht durch Aufspaltung in Prozeduren und Module

- Abstraktionen der Realität als Systemkern
- Beispiel: betriebswirtschaftliche Anwendung enthält Objekte und Klassen wie Kunde, Rechnung, Auftrag
- „Many people who have no idea how a computer works, find the idea of object-oriented systems quite natural. In contrast, many people who have experience with computers initially think there is something strange about object-oriented systems.“ (Robson 1981, zitiert nach [Schäfer 94], Seite 9)
- Vorteil: realitätsnahe Strukturierung

Abgrenzung zum prozeduralen Paradigma:

- vollkommen neue Denkweise beim Wechsel von der strukturierten Programmierung nötig
- traditionelle Entwurfsmethoden: Aufbrechen komplexer Operationen in kleine Schritte (Prozeduren und Module)
- bei objektorientierten Sprachen: Strukturierung des Systems nach Abstraktionen des Anwendungsbereiches
- wichtigste Bestandteile: Objekte und Klassen
- Erlernen einer neuen (eigentlich intuitiven) Art zu denken und Probleme zu analysieren

2.3 Software-Engineering-Aspekte

Ziel des Programmentwurfs:

Abbildung einer Realität auf ein Software-Modell.

Problem:

Erstellen eines Programms zur Problemlösung mit meist ungenauen und unvollständigen Anfangsanforderungen.

Software-Engineering:

Anwendung ingenieursmäßiger Arbeitsweisen auf den Prozess der Softwareentwicklung

Ziel des Software-Engineerings:

Erstellen von Programmen, die

- korrekt bzgl. der Problemstellung bzw. Anforderungsspezifikation (-> Validierung),
- korrekt bzgl. der Programmierung (-> Verifizierung),
- valdierbar und verifizierbar,
- leicht wartbar und
- kostengünstig

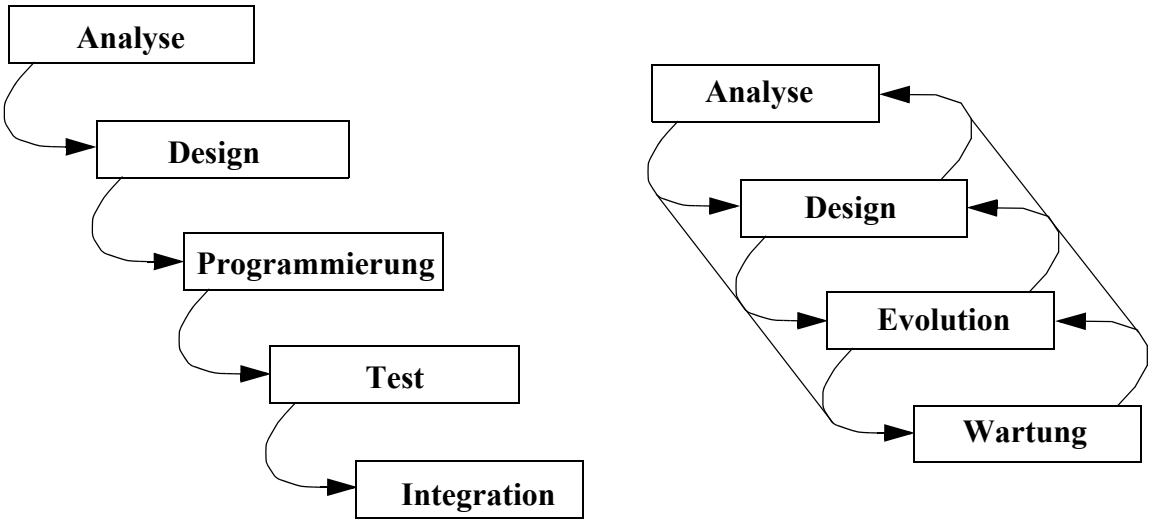
sind.

Grundschema des Entwurfs objektorientierter Software:

- Identifizierung der Klassen und Objekte

- Zuweisen von Attributen und Methoden
- Identifizierung der Beziehungen zwischen Klassen und Objekten
- Implementierung des Entwurfs

Gegenüberstellung Wasserfallmodell und evolutionäres Entwicklungsmodell



Gegenüberstellung traditionelles Wasserfallmodell und evolutionäres Designmodell aus [Booch 95], Seiten 198 und 200

3 Grundbegriffe der objektorientierten Programmierung

Grundbegriffe bzw. Grundelemente der Objektorientierung

- Objekte (Objects) bzw. Instanzen (Instances)
- Klassen (Classes)
- Attribute (Attributes)
- Methoden (Methods)
- Vererbung (Inheritence)
- Komposition (composition)
- Aggregation (Aggregation)
- Assoziation (Association)
- Botschaften (Messages)
- Polymorphismus (Polymorphism) (siehe Abschnitt 4.5)

Wichtig ist es, *Klassen* von *Objekten* zu unterscheiden. Objekte könnte man mit Chips vergleichen, aus denen elektronische Schaltungen aufgebaut sind. Der Entwickler wählt diese Chips aus einem Katalog aus, in dem für jedes Chip ein Datenblatt enthalten ist, das detailliert beschreibt, welche Ergebnisse das Chip aus welchen Daten erzeugen kann. Dieses Datenblatt entspricht in der objektorientierten Programmierung einer *Klasse*. Welche Ergebnisse erzeugt werden können, ist in der Definition von *Methoden* festgelegt.

Das *Instantiieren* einer Klasse erzeugt Objekte: Das entspricht genau dem Kaufen eines oder mehrerer Chips entsprechend eines Datenblatts aus dem Katalog.

Der Entwickler muss nun die gekauften Chips miteinander vedrahten, um die geforderte Schaltung aufzubauen. In der objektorientierten Programmierung werden die Objekte über *Botschaften* miteinander verbunden. Jede Botschaft enthält einen Aufruf einer Methode, die in der Regel ein Ergebnis zur Folge hat.

Sowohl die Klassen als auch das Instantiieren und das Verbinden der Objekte über Botschaften müssen programmiert werden. Das sind zwei grundsätzlich verschiedene Programmieraufgaben! Da der Entwickler von elektronischen Schaltungen in der Regel seine Chips nicht selbst entwickelt, sondern sie aus einem Katalog auswählt und kauft, so stehen dem Programmierer Bibliotheken zur Auswahl von Klassen zur Verfügung. - Die Standard Template Library des ANSI-Standards wäre dafür ein Beispiel. Ein anderes Beispiel ist Qt - das ist eine Klassenbibliothek und Entwicklungsumgebung für die plattformübergreifende Programmierung graphischer Benutzeroberflächen (GUI) unter C++.

3.1 Objekte und Klassen

Begriffe

Objekte

- sind Elemente, die in einem Anwendungsbereich von Bedeutung sind.
- Beispiele für Objekte:
  - ein bestimmter Abschnitt in einem Text
  - Bilanz des Monats Oktober in einer Finanzbuchhaltung
- Benutzersicht: Objekt stellt ein bestimmtes Verhalten zur Verfügung (Beispiel: Aufzug, besitzt Funktionalität auf- und abwärts zu fahren sowie Informationen über die augenblickliche Position)
- Programmierersicht: Objekte als Teile einer Applikation, die zusammenarbeiten, um die gewünschte Funktionalität zur Verfügung zu stellen.
- Objekte sind aktiv, reagieren auf Botschaften, wissen *wie* sie zu reagieren haben.

Klasse

- Beschreibung einer Menge (nahezu) gleicher Objekte. Der einzige Unterschied besteht in den Werten der Attribute (siehe Abschnitt 3.2)
- Klasse beschreibt die Daten und Funktionen, die eine Menge von Objekten charakterisieren.
- Klasse enthält also den Prototyp eines Objektes.
- Jedes Objekt übernimmt die Attribute und das Verhalten, das in seiner Klasse definiert ist.

Instanz

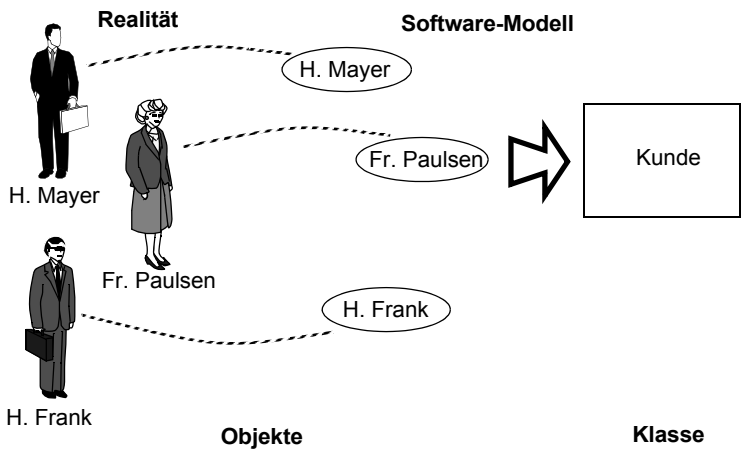
tatsächlich existierendes Objekt (Ausprägung einer Klasse)

Instantiierung

Vorgang des Erzeugens eines Objektes

Objekte und Klassen

Beispiel 1: (aus [Schäfer 94], Seite 11)



Beispiel 2:

- Klasse Auto

Daten:
- Typ:
- Leistung in kW:
- Anzahl der Scheinwerfer:
- momentane Geschwindigkeit:
- ...

Methoden:
- beschleunigen
- bremsen

- Objekt „Peters Auto“ (Instanz der Klasse Auto)

Daten:
- Typ: VW-Käfer
- Leistung in kW: 30
- Anzahl der Scheinwerfer: 2
- momentane Geschwindigkeit: 50
- ...

- Objekt „Mannis Auto“ (Instanz der Klasse Auto)

Daten:
- Typ: Opel-Manta
- Leistung in kW: 150
- Anzahl der Scheinwerfer: 6
- momentane Geschwindigkeit: 0
- ...

Zusammenfassung

Eine Klasse beschreibt ein Objekt, eine Instanz ist ein Objekt.

Klassen und Objekte in C++

Definition einer Klasse mit Schlüsselwort **`class`**, Instantiierung in Form von Variablendefinitionen.

Beispiel für eine Definition einer Klasse:

```
class Font {
// hier werden die Attribute und Methoden der Klasse Font deklariert
};
```

Beispiel für eine Instantiierung einer Klasse:

```
Font meinFont;
```

Darstellung von Klassen und Objekten:

3.2 Attribute und Methoden, Botschaften

Begriffe

Attribute

Beschreiben die Datenstruktur der Objekte einer Klasse, bestimmen den Zustand eines Objektes.

Methoden (auch Elementfunktionen)

Funktionen, die ein Objekt ausführen kann und die festlegen, wie ein Objekt zu reagieren hat, wenn es eine Botschaft erhält.

Botschaft (auch Nachricht)

- Botschaften dienen dem Aktivieren einer Methode (vergleichbar mit Funktionsaufruf in konventionellen Programmen).
- Objekte empfangen, bearbeiten und beantworten Nachrichten bei der Ausführung von Programmen.

Attribute, Methoden und Botschaften in C++

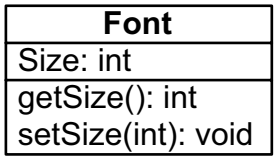
- Attribute/Methoden: Variablen und Funktionen, die innerhalb einer Klasse definiert sind.
- Nachrichten: Aufruf erfolgt über „.“-Operator (für Objekte) bzw. „->“-Operator (für Zeiger auf Objekte).

Beispiel

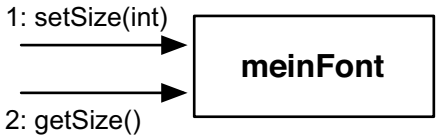
```
// Deklaration
class Font {
public:
   void setSize(int size); // Methode, nur Deklaration
   int getSize(void) {
      return Size;         // Methode
   };
private:
      int Size;            // Attribut
};

// Implementierung der Funktion setSize
void Font::setSize(int size) {
   Size = size;
};
```

```
// Zugriff
int main() {
  Font meinFont;        // Deklarieren eines Objektes vom Typ Font
  Font *meinFontZeiger; // Deklaration eines Pointers auf Font
  int s;

  meinFontZeiger = &meinFont;
  meinFont.setSize(10);          // Nachricht
  s = meinFontZeiger->getSize(); // Nachricht

  // ...
  return 0
}
```

Darstellung von Attributen und Methoden im Klassendiagramm:



Darstellung von Botschaften im Objektdiagramm:



Der Konstruktor: eine wichtige Methode

- Ein Konstruktor ist eine Methode, die automatisch aufgerufen wird, wenn ein Objekt generiert wird.
- Der Konstruktor ist eine Mitgliedsfunktion der Klasse. Sie hat den gleichen Namen wie die Klasse selbst und **keinen** Rückgabewert (nicht einmal **`void`**).
- Einzelheiten siehe Abschnitt 5.4.1

Beispiel:

```
class Font {
  public:
   Font(int s=12) {Size=s;}//Konstruktor mit Standard-Parameter
   // ohne Parameter muss das Objekt mit "Font meinFont(7);"
   // deklariert werden
   ...
   // weitere Methoden und Attribute wie oben
};
```

3.3 Vererbung

Begriffe

Vererbung (Inheritence, Generalization, auch Klassenhierarchie)

- Weitergabe von Eigenschaften von einer Klasse (Elternklasse) an eine hierarchisch untergeordnete Klasse (Kindklasse).
- Motivation:
  - Ausdrücken hierarchischer Beziehungen
  - Formulierung von Gemeinsamkeiten zwischen Klassen
  - Wiederverwendbarkeit
  - Reduzierung des Quellcodes
- Vererbung kann durch Überschreiben durchbrochen werden
  - Veränderung der Funktionen
  - Veränderung der Sichtbarkeit von Daten und Funktionen

Unterklasse (auch Subklasse, Kindklasse, abgeleitete Klasse)

Übernimmt (erbt) Attribute und Methoden von Elternklasse.

Oberklasse (auch Superklasse, Elternklasse)

Klasse, die in der Hierarchie höher angesiedelt ist.

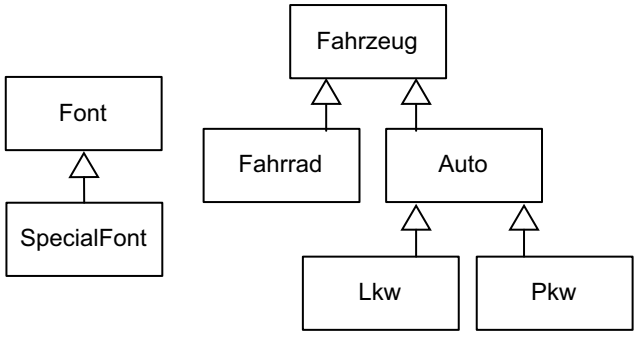
Einfachvererbung

Eine Klasse erbt von genau einer Oberklasse.

Mehrfachvererbung

Eine Klasse erbt von mehreren Oberklassen (siehe Abschnitt 4.3).

Darstellung der „erbt von“-Beziehung



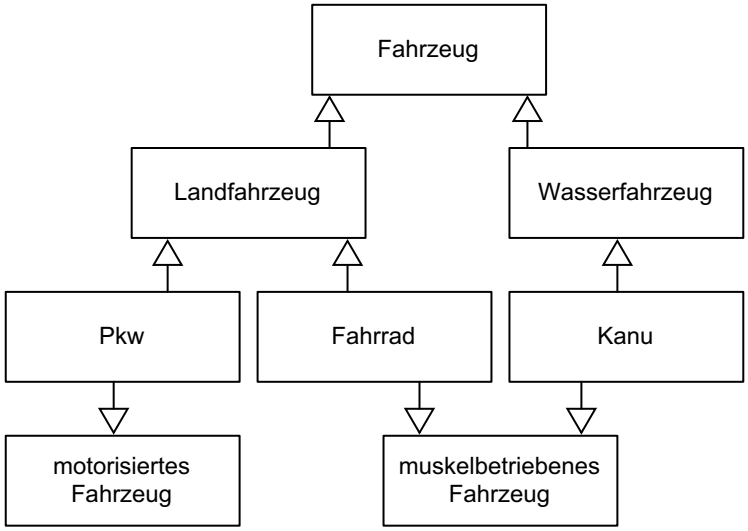
Vererbung in C++

```
enum FontType {normal, fett, kursiv};

class Font {
  public:
   void  setSize(int size);
   int   getSize()          {return Size;}
```

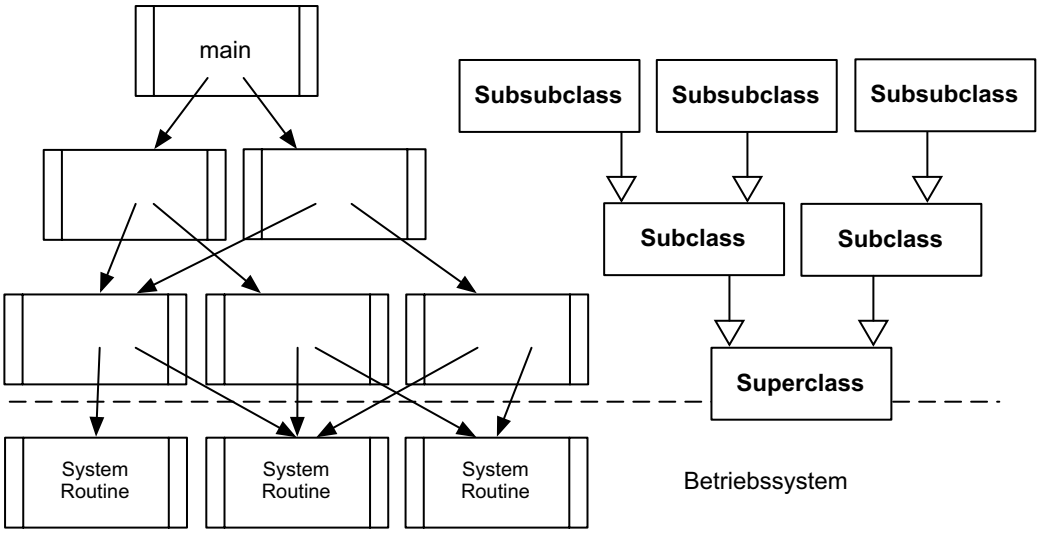
```
  private:
   int   Size;
};

class SpecialFont : public Font {
  public:
   FontType Type;
};
```

Die Klasse `SpecialFont` erbt die Attribute und Methoden der Klasse `Font`, sie besitzt folglich die Attribute `Size` und `Type` sowie die Methoden `setSize` und `getSize`.

Darstellung der Mehrfachvererbung



Vererbungsstruktur



Bei der Objektorientierten Programmierung steht die Welt Kopf gegenüber der alten prozeduralen Sichtweise. Das Programm `main` ist die Wurzel aus der andere Programme aufgerufen werden, die wiederum andere Programme aufrufen. Diese Hierarchie endet an der Grenze zum Betriebssystem.

Die letzten Aufrufe sind Systemaufrufe, entweder direkt aus dem Betriebssystem oder aus eingebundenen Bibliotheken.

Bei der Objektorientierten Programmierung ist auch die Grenze zum Betriebssystem vorhanden. (Objektorientierte Betriebssysteme gibt es noch nicht.) Die „Superklasse“ erbt die gesamte Funktionalität der Plattform mit ihrem Betriebssystem. Was sie nicht kann, kann auf diesem System nicht ausgeführt werden. Damit ist sie die Wurzel der Klassen-Hierarchie (gilt für die „strengen“ Sprachen Smalltalk, Objective-C). In C++ gibt es dagegen mehrere Superklassen; wir werden noch Qt kennenlernen. Die Subklassen stellen eine Spezialisierung dar. Dieser Baum endet dort, wo passende Klassen für eine Applikation zu finden sind. Sie werden hauptsächlich die Vererbung einsetzen, um vorgegebene Klassen aus der Qt-Bibliothek für Ihre Zwecke zu spezialisieren. Diese Spezialisierung ist die Grundidee für die Vererbungshierarchie. Wenn Sie diese Klassen angepasst haben, beginnt Ihre eigentliche Arbeit: die Klassen so miteinander in Beziehung zu setzen, dass die geforderte Applikation entsteht. Sei es, dass Sie die Klassen direkt (statisch) oder zur Laufzeit (dynamisch) zu einer Struktur verbinden, oder dass Sie Klassen über Botschaften miteinander Kommunizieren lassen.

3.4 Klassenbeziehungen

Für die Verwendung einer Klasse spielt es keine Rolle, ob sie etwas geerbt hat oder nicht, ob alle Methoden in ihr implementiert sind. Eine geerbte Methode unterscheidet sich nicht von einer Methode, die in der Klasse implementiert ist. Bei den Klassenbeziehungen geht es darum, welche Beziehungen zu anderen Klassen bestehen, welche Teile (Instanzen) von anderen Klassen genutzt werden. Man unterscheidet dabei die Komposition, die Aggregation und die Assoziation.

- eine Instanz einer Klasse enthält Instanzen anderer Klassen.
- Zusammenfassen von Objekten zu einem übergeordneten Objekt („... ist Bestandteil von ...“ oder „... ist in ... enthalten“).

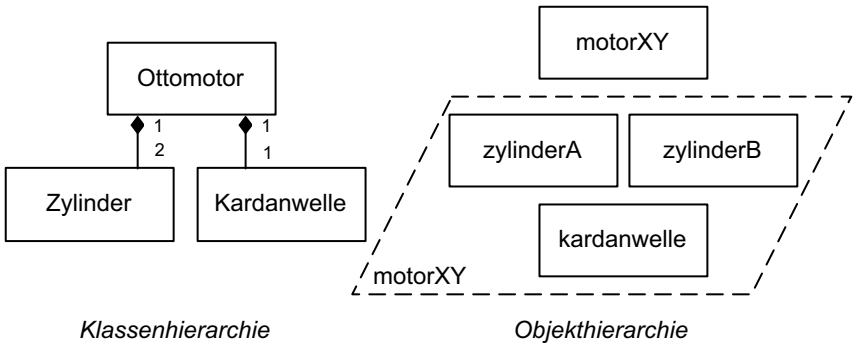
3.4.1 Komposition

Begriff

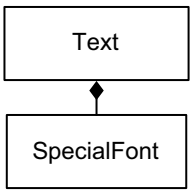
Komposition (auch „part of“-Beziehung, „has“-Beziehung, Enthalten-Relation)

Beispiele

Beispiel 1: Komposition (Composition)



Darstellung der Komposition („has“-Beziehung):



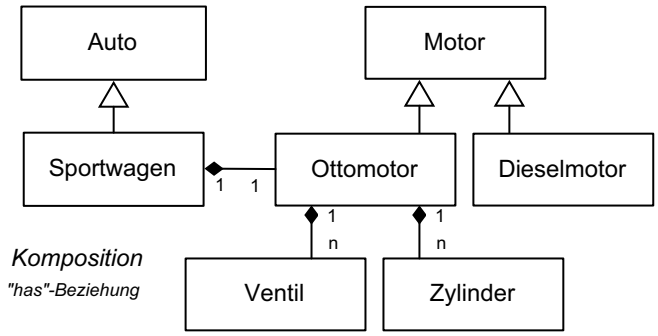
Aggregation in C++

Eine Instanz einer Klasse enthält Instanzen anderer Klassen.

Beispiel 2:

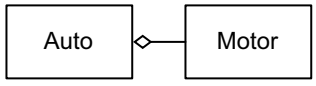
```
class Text {
   char myText[100];
   SpecialFont myFont; // Instantiierung der Klasse SpecialFont aus
                       // Abschnitt 3.3
}
```

Beispiel 3: Vererbung und Komposition



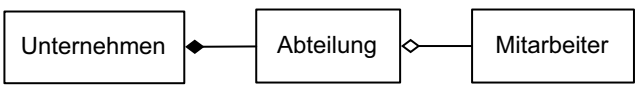
3.4.2 Aggregation

Bei der Aggregation wird die Instanz einer anderen Klasse hinzugefügt (aggregiert, „uses“-Beziehung). Das ist eigentlich kein Unterschied zur Komposition. Soll es auch nicht sein:



Aggregation

Mit diesem Auto lässt es sich genausogut Fahren wie dem Auto, das aus der Komposition hervorgegangen ist. Nur handelt es sich hier um einen Austauschmotor. Dieses Auto ist kein Neuwagen, bei dem der Motor mit dem Auto gemeinsam erzeugt (instantiiert) worden ist. Man kann auch die Lebenszeit betrachten. Wahrscheinlich wird der neue Motor länger als das Auto leben.

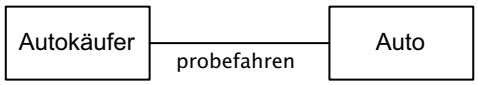


Komposition *Aggregation*

Bei diesem Beispiel wird der Unterschied deutlicher: Wenn das Unternehmen aufhört zu existieren, wird es auch die Abteilung nicht mehr geben. Der Mitarbeiter wird hoffentlich weiter leben.

3.4.3 Assoziation

Die Assoziation ist die schwächste Beziehung von Klassen untereinander.



Assoziation

Die Instanz der anderen Klasse existiert nur so lange, wie eine Funktion ausgeführt wird. Der Autokäufer muss das Auto nach der Probefahrt wieder zurückbringen. Es sei den, dass er einen Kaufvertrag über den gefahrenen Wagen abschließt. - Und das ist ein weiteres Beispiel für eine Assoziation. Der Kaufvertrag ist erfüllt, wenn das neue Auto übergeben worden ist.

Wir werden noch sehr viel genauer auf diese Beispiele eingehen, wenn wir das nötige Material für die Codierung haben. Die Unterschiede sind in der Initialisierung der Objekte beim Instantiieren zu finden. (siehe Abschnitt 4.4)

3.5 Beispielprogramme

Beispiel mit Standardausgabe und Kommentar

Ein erstes kleines Beispielprogramm kann so aussehen:

Beispiel 1:

```
#include <iostream>
void main()
{
  // Dies ist nur ein kleines Testprogramm.
  std::cout << "Hello world\n";
}
```

- Die Variable `cout` ist der Standardausgabe-Strom („Stream“). Mit `cout` wird Information am Bildschirm angezeigt. Dazu wird der <<-Operator verwendet.
- `cout` und der darauf bezogene Operator << sind in der Datei `<iostream>` definiert. Diese Datei gehört zur C++Standardbibliothek mit dem Standard-Namensraum `namespace std`. Man kann sich den Namen `std` und den Scope-Operator `::` vor `cout` sparen, wenn man vorher mit der `using`-Direktive `using namespace std;` den Standard-Namensraum aktiviert.
- Standardtypen können ohne weitere Formatierungsangaben (wie sie beispielsweise in C immer erforderlich sind) an `cout` übergeben werden (siehe auch Beispiel 2).
- Mit zwei vorwärts gerichteten Schrägstrichen `//` wird ein Kommentar eingeleitet, der sich bis zum Ende der Zeile erstreckt. Für mehrzeilige Kommentare ist das Zeichen entweder in jeder Zeile zu wiederholen, oder es sind die aus C bekannten Kommentarzeichen `/* ... */` zu verwenden.

Beispiel 2:

```
#include <iostream>
void main()
{
```

```
  int wert = 17;
  std::cout << "Der Wert ist " << wert << ".\n";
}
```

Die entsprechende Ausgabe lautet:

```
Der Wert ist 17.
```

Die Ausgabe kann jedoch auch zusätzlich formatiert werden. Einzelheiten dazu folgen in Abschnitt 5.9.

Das Gegenstück zu `cout` ist der Standardeingabe-Strom `cin`.

Beispiel 3:

```
#include <iostream>
void main()
{
  int wert;
  std::cout << "Geben Sie bitte einen Wert ein: ";
  std::cin >> wert;
  std::cout << "Der Wert ist " << wert << ".\n";
}
```

- Der >>-Operator für Eingabeströme ist anwendbar analog zum <<-Operator für Ausgabeströme.
- Auch hier ist standardmäßig keine Formatierungsangabe erforderlich.

Einzelheiten zu `cin` und `cout` folgen in Abschnitt 5.9.

4 Prinzipien der Objektorientierung

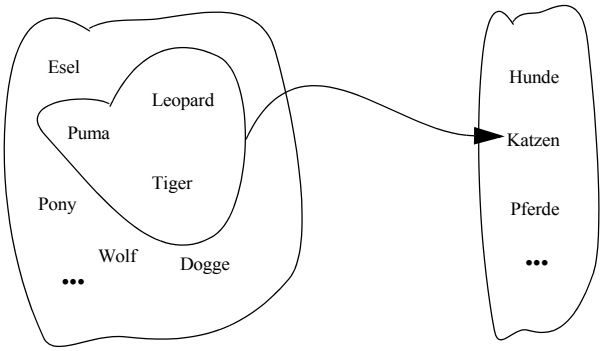
4.1 Abstraktion (Klassen und Objekte)

Begriff

Abstraktion

- grundlegendes Prinzip, um mit Komplexität umzugehen
- Konzentrieren auf das Wesentliche:
  - Erkennen von Gemeinsamkeiten zwischen verschiedenen Objekten.
  - Außer acht lassen von Unwichtigem.

Generalisierung



- [Schäfer 94] bezeichnet Abstraktion als ein sehr natürliches Konzept, mit dem selbst Kinder umgehen können. So stört es Kinder nicht, dass ihre Modell-Autos, Flugzeuge oder Schiffe keinen Motor haben, der wesentlicher Bestandteil des realen Objektes ist. Die Kinder wissen, dass der (komplexe) Motor ein wesentlicher Bestandteil des realen Objekts ist, er ist jedoch unwichtig für jene Abstraktion, mit der auf dem Fußboden des Kinderzimmers gespielt wird.
- In der Softwareentwicklung ist es erforderlich, Anwendungen in abstrakten Begriffen zu verstehen. Das für das Anwendungssystem wichtige Verhalten muss ausgehend von ein paar Objekten herausgefunden (abstrahiert) und in einer Klasse formuliert werden.
- Das Festlegen des richtigen Grades der Abstraktion ist das zentrale Problem beim Entwurf objektorientierter Systeme.

Abstraktion in C++

Das Prinzip der Abstraktion spiegelt sich in C++ beispielsweise durch die Bildung von Klassen als Abstraktion von Objekten wider (siehe Abschnitt 3.1).

4.2 Kapselung

4.2.1 Zugriffskontrollmechanismen

Begriff

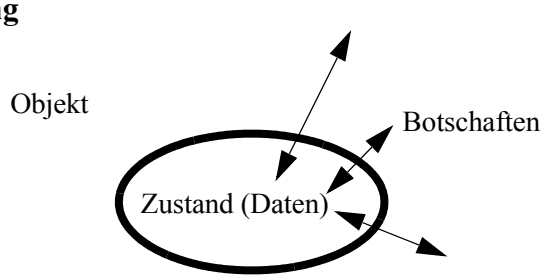
(Daten-)kapselung (Encapsulation, „Information Hiding“)

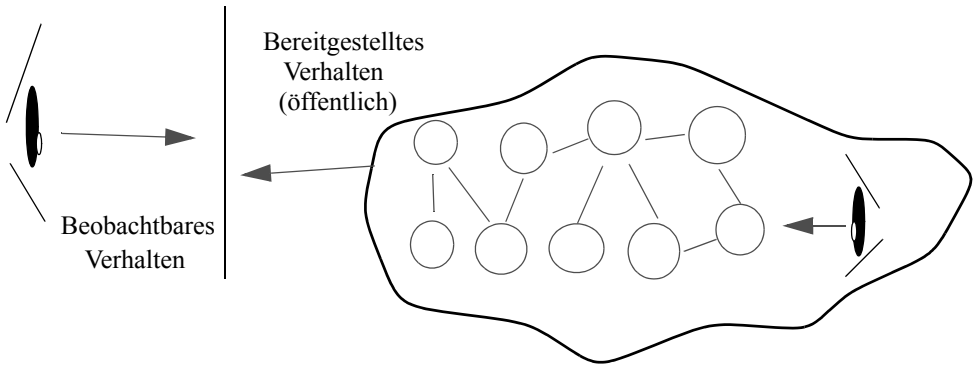
Zusammenfassung der Daten und Methoden eines Objektes so, dass auf die Daten nur mittels der Methoden zugegriffen werden kann.

Motivation

- interne Darstellung von Daten nicht nach außen hin sichtbar
- interne Darstellung von Daten unabhängig von anderen Objekten veränderbar
- Methoden nach außen sichtbar, deren Implementierung aber nicht

Prinzip der Datenkapselung





Kapselung in C++

Zugriffskontrolle

für Daten und Methoden mittels der Schlüsselwörter **`private`**, **`protected`** und **`public`**.

`private`: Zugriff nur innerhalb der Klasse

`protected`: Zugriff nur innerhalb der Klasse und von abgeleiteten Klassen

`public`: Zugriff auch von außerhalb möglich

Beispiel

```
class Font {
   public:
   int   getSize()          {return Size;}
   protected:
   void  setSize(int size)  {Size = size;}
   private:
   int   Size;
};
```

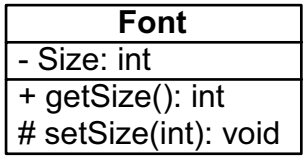
In diesem Beispiel können nur Funktionen innerhalb der Klasse `Font` (also die Methoden der Klasse) auf die Variable `Size` zugreifen.

Der Zugriff von außerhalb erfolgt statt dessen über die Methoden `setSize` (**`protected`**, d.h. verfügbar auch für Klassen, die von `Font` abgeleitet sind) und `getSize` (**`public`**, voll verfügbar auch außerhalb von `Font`).

Beispiel (Fortsetzung)

```
enum FontType {normal, fett, kursiv};

class SpecialFont : public Font {
public:
   FontType Type;
   void initialisiere();
};

void SpecialFont::initialisiere()
{
  Size = 15;    // Fehler! Size ist private
  setSize(25);  //ok, setSize ist protected und damit hier verfügbar
}

void
main()
{
  Font meinFont;
  SpecialFont meinSpecialFont;

  meinFont.Size = 15;             // Fehler! Size ist private
  meinFont.setSize(15);           // Fehler! setSize ist protected
  meinSpecialFont.setSize(15);      // Fehler! setSize ist protected
  std::cout << meinSpecialFont.getSize();  // ok, getSize ist public
}
```

Darstellung der Kapselung:



4.2.2 Zugriffsfunktionen und als `public` definierte Daten

Das Schreiben von öffentlichen Zugriffsfunktionen im Stil des obigen Beispiels (`setSize`, `getSize`) erscheint zunächst recht umständlich gegenüber der einfacheren Vorgehensweise, Daten gleich als **`public`** zu definieren.

Es gibt jedoch eine Reihe von wichtigen Vorteilen:

- Die Funktionen zum Setzen der Variablen können die Daten überprüfen und ungültige Daten von vornherein ausschließen (z.B. „31. Februar“).
  Funktionen, die die Daten nutzen (z.B. Ausgaberoutinen), können auf eine Überprüfung der Daten auf ungültige Werte verzichten und so wesentlich einfacher gestaltet werden.
- Die Kapselung der Daten in dieser Form vereinfacht eine evtl. später erforderliche Änderung der Implementierung der Datenstrukturen.
  Soll beispielsweise ein Datumswert, der zunächst in Form getrennter Integer- oder Enumwerte für Tag, Monat und Jahr gespeichert war, aus Speichergründen in die Bits eines einzigen Integerwertes umcodiert werden, so sind lediglich die Zugriffsfunktionen zu ändern.

Kurze Zugriffsfunktionen können als **`inline`** definiert werden (siehe Abschnitt 5.8). Dadurch ist der Zugriff praktisch genauso effizient wie ein direkter Zugriff auf die Daten selbst.

4.2.3 Private Vererbung

- Die Vererbung einer Klasse kann mit dem Schlüsselwort **`private`** deklariert werden.
- Damit sind alle als **`public`** oder **`protected`** deklarierten Daten und Mitgliedsfunktionen der Basisklasse private Elemente der abgeleiteten Klasse.
- Diese Daten und Mitgliedsfunktionen sind damit weder in weiter abgeleiteten Klassen, noch außerhalb der Klasse zugänglich.
- Die private Vererbung hat wenig praktische Bedeutung.
- Von der Bedeutung im Designprozess her hat die private Vererbung mehr Ähnlichkeit zur Aggregation (siehe Abschnitt 3.4.2) als zur Vererbung (Abschnitt 3.3).

Beispiel:

```
class SpecialFont : private Font {
public:
   FontType Type;
};

void
main()
{
  SpecialFont meinSpecialFont;

  meinSpecialFont.getSize(15);  // Fehler! nicht verfügbar, obwohl
                                // in Font als public definiert
}
```

4.2.4 Friend-Klassen und Friend-Funktionen in C++

Begriff:

Mit Hilfe des Schlüsselwortes **`friend`** kann einer anderen Klasse bzw. einer Funktion der Zugriff auf als **`private`** und **`protected`** deklarierte Attribute und Methoden der eigenen Klasse gewährt werden.

Beispiel 1: Friend-Klasse

```
class MeineKlasse {
friend class DeineKlasse;
private:
  int meineGeheimzahl;
};

class DeineKlasse {
public:
  void deineFunktion(MeineKlasse *mk)
          {mk->meineGeheimzahl++; /* Zugriff erlaubt!!! */}
};
```

Beispiel 2: Friend-Funktion

```
class MeineKlasse {
friend void oeffentlicheFunktion(MeineKlasse *mk);
private:
  int meineGeheimzahl;
};

void oeffentlicheFunktion(MeineKlasse *mk) {
  mk->meineGeheimzahl++; // Zugriff erlaubt!
}
```

Zusammenfassung:

- Friend-Klassen und Friend-Funktionen sollten äußerst sparsam und vorsichtig eingesetzt werden, da Sie dem Prinzip der Kapselung entgegenstehen.
- Sie sind nur sinnvoll, wenn zwei Klassen so eng kooperieren, dass der Zugriff über Zugriffsfunktionen zu ineffizient wäre.

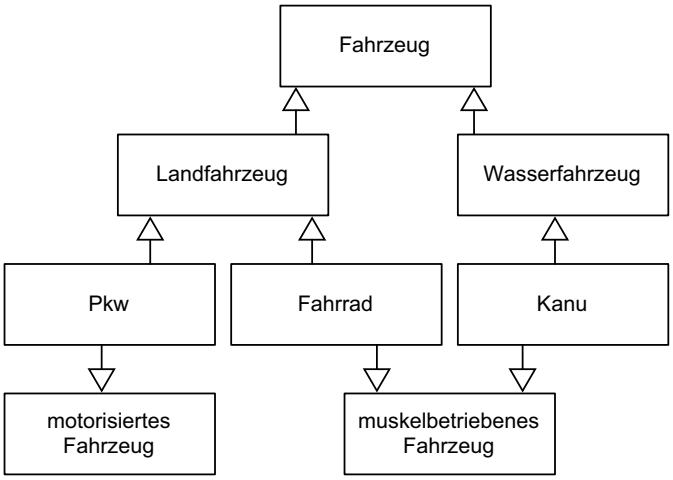
4.3 Mehrfachvererbung

Begriff

mehrfache Vererbung (Multiple Inheritance)

Eine Klasse erbt von mehreren Oberklassen.

Beispiel für Mehrfachvererbung



Mehrfachvererbung in C++

Im folgenden Beispiel sind vier Klassen A, B, C, und D enthalten. Mit denen werden sieben Versuche durchgeführt, um die Verhältnisse der Mehrfachvererbung aufzuzeigen. Es handelt sich um einfache, identische Klassen. Jede Klasse hat einen privaten Zustand (`i`, `j`), der mit `getZustand (void)` ausgegeben werden kann. Interessant ist, wo die Objekte jeweils im Speicher zu liegen kommen. Aus diesem Grund wird auch jeweils die Adresse `this` mit ausgegeben. `this` ist ein Pointer auf das instantiierte Objekt. Beispielsweise ist `this->i` die Variable `i`, die unter `private` in der Klasse abgelegt ist.(siehe auch: Der this-Zeiger auf Seite 65)

Beispiel

```
class A {
public:
   A (int i=1000, int j=2000) {
      this->i=i;
      this->j=j;
   };
   void getZustand (void);
private:
   int i;
   int j;
};

void A::getZustand (void) {
   std::cout << "Ich bin Klasse A mit Zustand"
   << " (i= "<< i << " j= " << j << " this = " << this << ")"
   << std::endl;
};

class B : public A {
public:
   B (int i=101, int j=102) : A(111, 112) {
      this->i=i;
      this->j=j;
   };
```

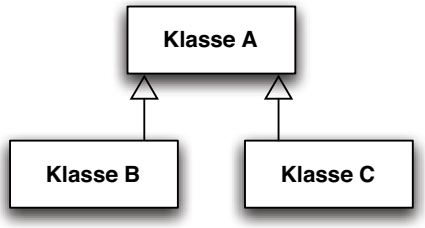
```
   void getZustand (void);
private:
   int i;
   int j;
};

void B::getZustand (void) {
   std::cout << "Ich bin Klasse B mit Zustand"
   << " (i= "<< i << " j= " << j << " this = " << this << ")"
   << std::endl;
   A::getZustand ();
};

class C : public A {
public:
   C (int i=201, int j=202) : A(211, 212) {
      this->i=i;
      this->j=j;
   };
   void getZustand (void);
private:
   int i;
   int j;
};

void C::getZustand (void) {
   std::cout << "Ich bin Klasse C mit Zustand"
   << " (i= "<< i << " j= " << j << " this = " << this << ")"
   << std::endl;
   A::getZustand ();
};

class D : public B, public C {
public:
   D (int i=301, int j=302) : B(311, 312), C(321, 322) {
      this->i=i;
      this->j=j;
   };
   void getZustand (void);
private:
   int i;
   int j;
};

void D::getZustand (void) {
   std::cout << "Ich bin Klasse D mit Zustand"
   << " (i= "<< i << " j= " << j << " this = " << this << ")"
   << std::endl;
```

```
    B::getZustand ();
    C::getZustand ();
    A::getZustand ();
};
```

4.3.1 Einfache und mehrfache Vererbung

Erster Versuch: einfache Vererbung

Es werden die beiden Klassen B und C instantiiert. Beide sind von der Klasse A abgeleitet:

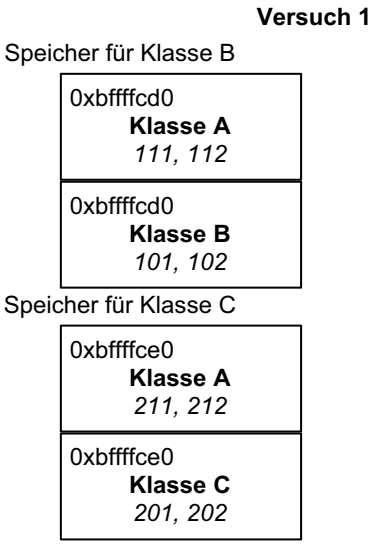


Programm

```
B b;
b.getZustand();
C c;
c.getZustand ();
```

Programm-Ausgabe

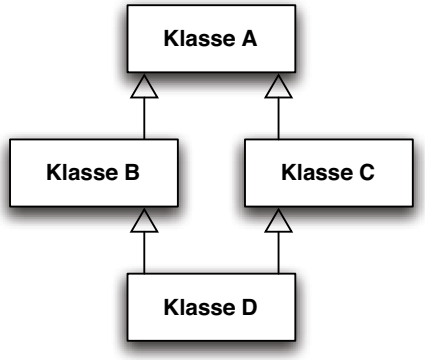
```
Ich bin Klasse B mit Zustand (i= 101 j= 102
this = 0xbfffffcd0)
Ich bin Klasse A mit Zustand (i= 111 j= 112
this = 0xbfffffcd0)
Ich bin Klasse C mit Zustand (i= 201 j= 202
this = 0xbfffffce0)
Ich bin Klasse A mit Zustand (i= 211 j= 212
this = 0xbfffffce0)
Hello, World!
```



Beide Klassen haben ihren eigenen Speicherbereich. Weil sie von der Klasse A abgeleitet sind, ist auch jeweils eine Instanz von A enthalten.

Zweiter Versuch: mehrfache Vererbung

Im zweiten Versuch wird nun die Klasse D instantiiert, die sowohl von der Klasse B als auch von der Klasse C erbt.



Programm

```
D d;
d.getZustand ();
```

Compiler-Meldung:

```
main.cpp:81:
internal compiler error:
in build_base_path, at cp/class.c:286
```

Source Code

```
// Zeile 81 ist in class D:
   A::getZustand ();
```

Für den Compiler stellt die Mehrfachvererbung offensichtlich ein größeres Problem dar. Zum Glück gibt er noch die Zeile an, in der für ihn das Problem liegt. Aber es ist nicht nur für den Compiler ein Problem, sondern auch für uns! Es existieren nämlich zwei Wege für die Vererbung. Die Klasse D erbt nicht nur von den Klassen B und C, sondern implizit auch von der Klasse A - und das eben doppelt!

Dritter Versuch: auflösen von Doppeldeutigkeit bei Mehrfachvererbung

Wir machen die Zeile 81 einfach zu einem Kommentar!

Source Code

```
// So könnte man die Doppeldeutigkeit von A::getZustand (); auflö-
sen:
   D d;
   d.getZustand ();
   d.B::getZustand ();
   d.C::getZustand ();
```

Der Weg für die Vererbung zur Klasse A muss eindeutig angegeben werden. Dazu dient uns der Bezugsoperator (scope operator), entweder `B::` für den Weg über die Klasse B oder `C::` für den Weg über C.(siehe auch Abschnitt 5.12.3)

Vierter Versuch: Speicherbelegung bei Mehrfachvererbung

Wir belassen es dabei, weil uns ohne das Erben von A die Klassen C und D ausreichende Informationen geben. Sie erben ja auch von der Klasse A.

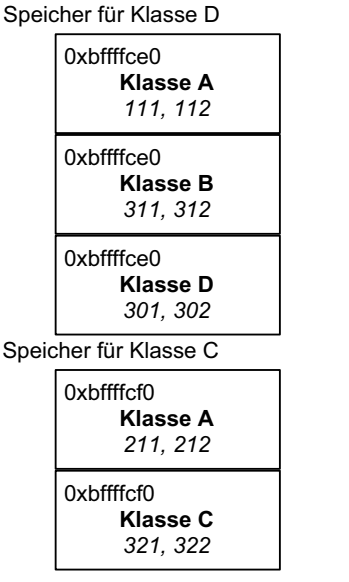
Programm

```
D d;
d.getZustand ();
```

Programm-Ausgabe

```
Ich bin Klasse D mit Zustand (i= 301 j= 302
this = 0xbffffce0)
Ich bin Klasse B mit Zustand (i= 311 j= 312
this = 0xbffffce0)
Ich bin Klasse A mit Zustand (i= 111 j= 112
this = 0xbffffce0)
Ich bin Klasse C mit Zustand (i= 321 j= 322
this = 0xbffffcf0)
Ich bin Klasse A mit Zustand (i= 211 j= 212
this = 0xbffffcf0)
Hello, World!
```

Versuch 4

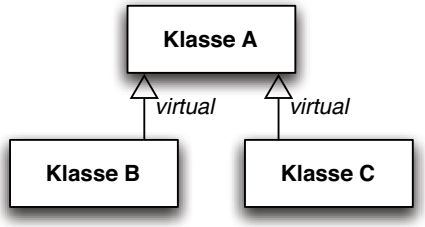


An der Speicherbelegung ist deutlich zu sehen, dass das Problem nicht in den Wegen begründet ist, sondern es gibt zwei Instanzen von der Klasse A mit unterschiedlichen Attributen. Wir haben verschiedene Zustände! Und darin liegt das eigentliche Problem begründet. Der Programmierer muss sich im Klaren sein, was er eigentlich mit dieser Konstruktion erreichen will.

4.3.2 Virtuelle Klassen

Fünfter Versuch: Verwaltung virtueller Klassen

Wir vergessen im Moment die Mehrfachvererbung und konzentrieren uns auf eine Abhilfe, Mehrdeutigkeiten in Form von mehreren Instanzen einer Klasse aufzuheben. Die Abhilfe ist die Verwaltung virtueller Klassen. Durch die folgende Vererbung wird die Klasse A zu einer virtuellen Basisklasse.



Source Code

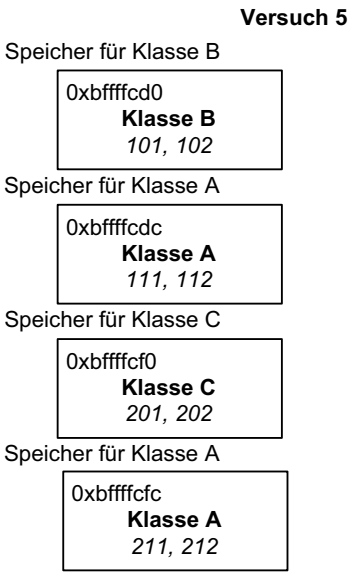
```
class B : public virtual A {
class C : public virtual A {
```

Programm

```
B b;
b.getZustand();
C c;
c.getZustand ();
```

Programm-Ausgabe

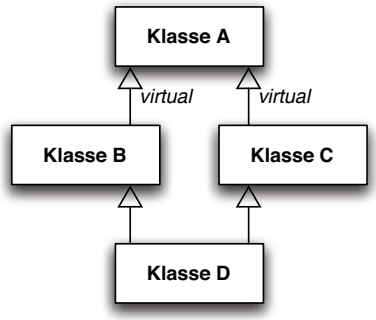
```
Ich bin Klasse B mit Zustand (i= 101 j= 102
this = 0xbfffffcd0))
Ich bin Klasse A mit Zustand (i= 111 j= 112
this = 0xbfffffcdc)
Ich bin Klasse C mit Zustand (i= 201 j= 202
this = 0xbfffffcf0)
Ich bin Klasse A mit Zustand (i= 211 j= 212
this = 0xbfffffcfc)
Hello, World!
```



Auffallend an der Speicherbelegung ist, dass keine zusammenhängende Speicherbereiche mehr existieren. Der Compiler hat nämlich eine sogenannte VTable angelegt, mit der die Objekte zur Laufzeit verwaltet werden. (Kann einen nicht unerheblichen Aufwand zur Laufzeit bedeuten. Sie sind aber bei virtuellen Methoden unerlässlich. siehe Abschnitt 4.5.2)

Sechster Versuch: virtuelle Klassen

Nun kehren wir zu unserer Mehrfachvererbung zurück. Wir instantiieren die Klasse D, die nun virtuell über C und D die Klasse A erbt. Damit sind alle Doppeldeutigkeiten aufgehoben.

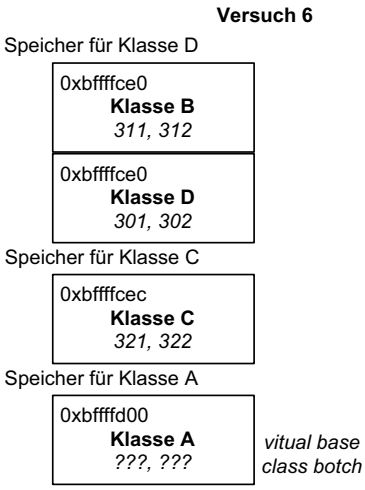


Programm

```
D d;
d.getZustand ();
```

Programm-Ausgabe

```
Ich bin Klasse D mit Zustand (i= 301 j= 302
this = 0xbfffffce0)
Ich bin Klasse B mit Zustand (i= 311 j= 312
this = 0xbfffffce0)
Ich bin Klasse A mit Zustand (i= 1000 j= 2000
this = 0xbfffffd00)
Ich bin Klasse C mit Zustand (i= 321 j= 322
this = 0xbfffffcec)
Ich bin Klasse A mit Zustand (i= 1000 j= 2000
this = 0xbfffffd00)
Hello, World!
ACHTUNG! GDB: Error: virtual base class botch
(botch = Murks :-) )
```



Unser Ziel ist nun (fast) erreicht. Es existiert nur noch eine Instanz von der Klasse A! Dem normalen Programmablauf ist nichts anzumerken, aber der Debugger kritisiert den Zustand der virtuellen Basisklasse. Der von der Klasse A gemeldeten Zustand ist überhaupt nicht plausibel. Er entspricht zwar dem in der Klassendefinition festgelegten Standardwert, aber die Klasse B oder C sollte ihn mit ihrem Konstruktor ändern. Dem Zustand 1000, 2000 ist wirklich nicht zu trauen.

Siebter Versuch: virtuelle Klassen initialisieren

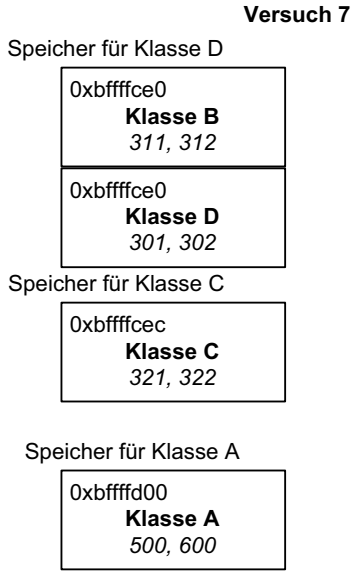
Da jetzt nur eine einzige Instanz der virtuellen Basisklasse existiert, auch für die Zwischenklassen, die aber offensichtlich auf die Initialisierung dieser Basisklasse mit ihren Konstruktoren keinen Einfluss nehmen können, ist der Programmierer gezwungen, mit einem einzigen Konstruktor alle Objekte zu initialisieren.

Source Code

```
Konstruktor von D:
   D (int i=301, int j=302) : A(500,600), B(311, 312), C(321, 322) {
Der Konstruktor muss eindeutige Verhältnisse schaffen: Basisklasse
initialisieren!!!
```

Programmausgabe:

```
Ich bin Klasse D mit Zustand (i= 301 j= 302
this = 0xbfffffce0)
Ich bin Klasse B mit Zustand (i= 311 j= 312
this = 0xbfffffce0)
Ich bin Klasse A mit Zustand (i= 500 j= 600
this = 0xbfffffd00)
Ich bin Klasse C mit Zustand (i= 321 j= 322
this = 0xbfffffcec)
Ich bin Klasse A mit Zustand (i= 500 j= 600
this = 0xbfffffd00)
Hello, World!
```



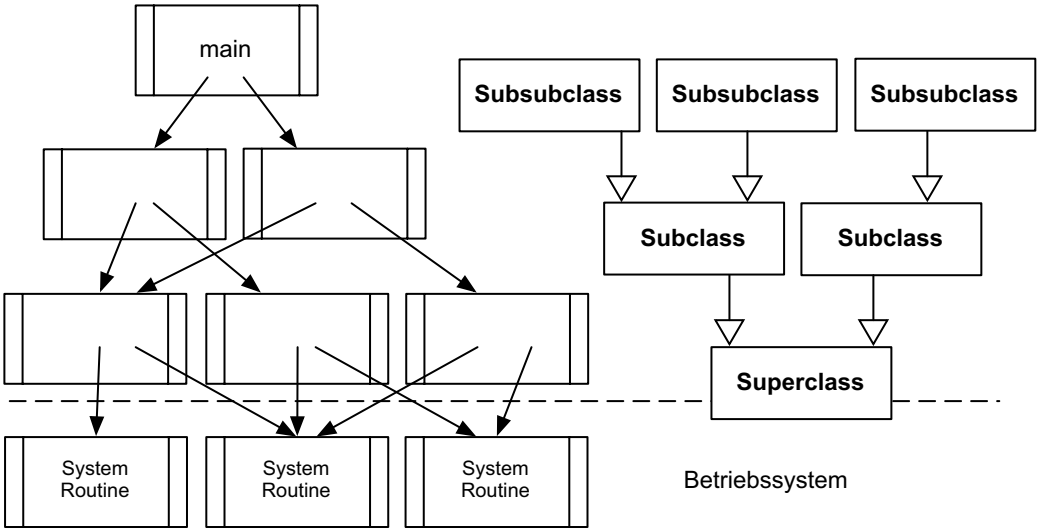
Dass der Programmierer von einem einzigen Punkt aus die Initialisierung vornehmen muss, ist ein großer Nachteil! Er muss ja auf alle Details der oberen Klassen eingehen, was er normalerweise nicht tut, weil ja jede Klasse für sich abgeschlossen ist. Er braucht lediglich mit seinem Konstruktor die anderen an der Vererbungslinie anzustossen. Hier muss er aber so tun, als wenn nur noch eine

Klasse existiert. Das dürfte ein Grund dafür sein, dass andere Sprachen keine Mehrfachvererbung (wie Java, C#) kennen.

4.4 Klassenbeziehungen

4.4.1 Vererbung

Über Vererbung soll hier nicht mehr viel gesagt werden. Sie ist in den verschiedensten Formen behandelt worden.

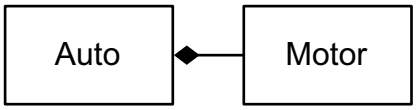


Prozeduren Hierarchie *Klassen Hierarchie*

Es sei nur nochmal an die prinzipielle Vererbungsstruktur erinnert. Ihre Struktur ist ein Baum, der gegenüber der alten prozeduralen Sichtweise auf dem Kopf steht. Die generelle Idee, die zu dieser Struktur führt, ist die Spezialisierung. Eine abgeleitete Klasse passt die Klasse, von der sie erbt, an die speziellen Verhältnisse an. Die eigentliche Struktur der zu entwickelnden Applikation kommt durch das Inbeziehungssetzen der Klassen zustande. Das entspricht dem Layout in der Elektronik.

4.4.2 Komposition

Bei der Komposition enthält die Instanz von anderen Klassen Instanzen.



Komposition

Es folgt der vollständige Source-Text für dieses Beispiel.

- Gleich in der ersten Zeile der Datei `Auto.h` wird der `Motor engebautMotor` instantiiert.

```
./*
  *   Auto.h
  *   Komposition
  *
  *   Created by Peter Krauss on Thu Mar 25 2004.
  *   Copyright (c) 2004 TUM. All rights reserved.
```

```
 *
 */
#include <string>
#include "Motor.h"

class Auto {
public:
   Motor eingebautMotor; //Motor wird erzeugt
   Auto (std::string initFabrikat = "?", std::string initType ="?",
      std::string initArt = "?", int initKw = 9999);
      // Konstruktor Deklaration mit Std-Werten

   void setFabrikat (std::string n);
   std::string getFabrikat (void);

   void setType (std::string n);
   std::string getType (void);

private:
   std::string fabrikat, type;
};
/*
 *  Auto.cpp
 *  Komposition
 *
 *  Created by Peter Krauss on Thu Mar 25 2004.
 *  Copyright (c) 2004 TUM. All rights reserved.
 *
 */
#include "Auto.h"

Auto::Auto (std::string initFabrikat, std::string initType,
      std::string initArt, int initKW)
   : eingebautMotor(initArt, initKW),
      fabrikat(initFabrikat), type(initType) {}; // Konstruktor

void Auto::setFabrikat (std::string n){
   fabrikat = n;
};
std::string Auto::getFabrikat (void){
   return fabrikat;
};

void Auto::setType (std::string n){
   type = n;
};
std::string Auto::getType (void){
   return type;
```

```
};
/*
 *  Motor.h
 *  Komposition
 *
 *  Created by Peter Krauss on Thu Mar 25 2004.
 *  Copyright (c) 2004 TUM. All rights reserved.
 *
 */
#include <string>

class Motor {
public:
   Motor (std::string initArt = "?", int initKw = 88);
      // Konstruktor Deklaration mit Std-Werten

   void setArt (std::string n);
   std::string getArt (void);

   void setKW (int n);
   int getKW (void);

private:
   std::string art;
   int kw;
};
/*
 *  Motor.cpp
 *  Komposition
 *
 *  Created by Peter Krauss on Thu Mar 25 2004.
 *  Copyright (c) 2004 TUM. All rights reserved.
 *
 */

#include "Motor.h"

Motor::Motor (std::string initArt, int initKw)
   : art(initArt), kw(initKw) {}; // Konstruktor

void Motor::setArt (std::string n){
   art = n;
};
std::string Motor::getArt (void){
   return art;
};

void Motor::setKW (int n){
```

```
   kw = n;
};
int Motor::getKW (void){
   return kw;
};
/*
 *  main.cpp
 *  Komposition
 *
 *  Created by Peter Krauss on Thu Mar 25 2004.
 *  Copyright (c) 2004 TUM. All rights reserved.
 *
 */
#include <iostream>
#include "Auto.h"

int main (int argc, char * const argv[]) {
   Auto meinAuto;//Instanziierung vom Auto

   std::cout << "Adresse von meinAuto " << &meinAuto << std::endl;
      // Adresse von meinAuto 0xbffffc30
   std::cout << "Adresse von eingebautMotor "
      << &meinAuto.eingebautMotor << std::endl << std::endl;
//Adresse von eingebautMotor 0xbffffc30 // Adressen sind gleich!!!!

   std::string fabrikat = meinAuto.getFabrikat();
   std::string type = meinAuto.getType();
   std::string art = meinAuto.eingebautMotor.getArt();
   int kw = meinAuto.eingebautMotor.getKW();

    std::cout << "Auto Fabrikat = " << fabrikat  << " type = " <<
type << std::endl;
     // Auto Fabrikat = ? type = ?
    std::cout << "Motor Art = " << art << " KW = " << kw <<
std::endl << std::endl;
     // Motor Art = ? KW = 9999

   meinAuto.setFabrikat("Alfa Romeo");
   meinAuto.setType("GTV");
   meinAuto.eingebautMotor.setArt("V 6");
   meinAuto.eingebautMotor.setKW(110);

   fabrikat = meinAuto.getFabrikat();
   type = meinAuto.getType();
   art = meinAuto.eingebautMotor.getArt();
   kw = meinAuto.eingebautMotor.getKW();

   std::cout << "Auto Fabrikat = " << fabrikat  << " type = "
            << type << std::endl;
```

```
      // Auto Fabrikat = Alfa Romeo type = GTV
   std::cout << "Motor Art = " << art << " KW = " << kw
             << std::endl;
      // Motor Art = V 6 KW = 110

   return 0;
}
```

Programm-Ausgabe

```
Adresse von meinAuto 0xbffffc30
Adresse von eingebautMotor 0xbffffc30

Auto Fabrikat = ? type = ?
Motor Art = ? KW = 9999

Auto Fabrikat = Alfa Romeo type = GTV
Motor Art = V 6 KW = 110

Komposition has exited with status 0.
```

Besonders interessant sind in der Programm-Ausgabe die Adressen der Instanzen `meinAuto` und `eingebautMotor`. Sie sind gleich! Durch die Komposition entsteht eine einzige Einheit. - Den gleichen Effekt konnten wir bei der einfachen Vererbung (Abschnitt 4.3.1) beobachten.

Ein besonderes Augenmerk ist auf den Konstruktor von der Klasse Auto zu richten.

Source-Code

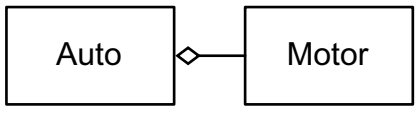
```
Auto::Auto (std::string initFabrikat, std::string initType,
      std::string initArt, int initKW)
   : eingebautMotor(initArt, initKW),
      fabrikat(initFabrikat), type(initType) {}; // Konstruktor
```

- Es wird das Auto vollständig (Auto und gleichzeitig Motor) initialisiert, und zwar so als wenn eine normale Klasse vorliegen würde.

Das ist das Kennzeichen Komposition!

4.4.3 Aggregation

Auch bei der Aggregation enthält die Instanz von anderen Klassen Instanzen. Der Unterschied wird bei den Konstruktoren zu sehen sein.



Aggregation

Es folgt der vollständige Source-Text für dieses Beispiel.

- Hier wird im Unterschied zur Komposition kein `Motor engebautMotor` instantiiert. Der Motor taucht erst im Konstruktor als Referenz `Motor & austauschMotor` auf.

```
/*
 *  Auto.h
 *  Aggregation
 *
 *  Created by Peter Krauss on Thu Mar 25 2004.
 *  Copyright (c) 2004 TUM. All rights reserved.
 *
 */
#include <string>
#include "Motor.h"

class Auto {
   public:
      Auto (Motor & austauschMotor, std::string initFabrikat = "?",
         std::string initType = "?",
         std::string initArt = "?", int initKw = NULL);
         // Konstruktor Deklaration

      void setFabrikat (std::string n);
      std::string getFabrikat (void);

      void setType (std::string n);
      std::string getType (void);
      Motor &eingebautMotor;// Rerenz vom Motor

   private:
      std::string fabrikat, type;
};
/*
 *  Auto.cpp
 *  Aggregation
 *
 *  Created by Peter Krauss on Thu Mar 25 2004.
 *  Copyright (c) 2004 TUM. All rights reserved.
 *
 */
#include "Auto.h"

Auto::Auto (Motor & austauschMotor,
      std::string initFabrikat, std::string initType,
      std::string initArt, int initKW)
   : eingebautMotor(austauschMotor),
      fabrikat(initFabrikat), type(initType) {};

void Auto::setFabrikat (std::string n){
   fabrikat = n;
};
std::string Auto::getFabrikat (void){
```

```
   return fabrikat;
};

void Auto::setType (std::string n){
   type = n;
};
std::string Auto::getType (void){
   return type;
};
/*
 *  Motor.h
 *  Aggregation
 *
 *  Created by Peter Krauss on Thu Mar 25 2004.
 *  Copyright (c) 2004 TUM. All rights reserved.
 *
 */
#include <string>

class Motor {
public:
   Motor (std::string initArt = "?",
      int initKw = 9999);
      // Konstruktor Deklaration mit Std-Werten

   void setArt (std::string n);
   std::string getArt (void);

   void setKW (int n);
   int getKW (void);

private:
   std::string art;
   int kw;
};
/*
 *  Motor.cpp
 *  Aggregation
 *
 *  Created by Peter Krauss on Thu Mar 25 2004.
 *  Copyright (c) 2004 TUM. All rights reserved.
 *
 */
#include "Motor.h"

Motor::Motor (std::string initArt, int initKw)
   :  art(initArt), kw(initKw) {}; //Konstruktor
```

```
void Motor::setArt (std::string n){
   art = n;
};
std::string Motor::getArt (void){
   return art;
};

void Motor::setKW (int n){
   kw = n;
};
int Motor::getKW (void){
   return kw;
};
/*
 *  main.cpp
 *  Aggregation
 *
 *  Created by Peter Krauss on Thu Mar 25 2004.
 *  Copyright (c) 2004 TUM. All rights reserved.
 *
 */
#include <iostream>
#include "Auto.h"


int main (int argc, char * const argv[]) {
   Motor austauschMotor("zweiTakter");
   Auto meinAuto(austauschMotor);//Instanziierung von Auto

   std::cout << "Adresse von meinAuto " << &meinAuto << "\n";
      // Adresse von meinAuto 0xbffffc40
   std::cout << "Adresse von eingebautMotor "
      << &meinAuto.eingebautMotor << std::endl << std::endl;
      // Adresse von eingebautMotor 0xbffffc10
      // Adressen verschieden!!!

   std::string artAustausch = austauschMotor.getArt();
   int kwAustausch = austauschMotor.getKW();
   std::cout << "MotorAustauschArt = " << artAustausch
      << " MotorAustauschKW = " << kwAustausch << std::endl
      << std::endl;
      // MotorAustauschArt = zweiTakter MotorAustauschKW = 9999
      // Das ist der Beweis: Motor existiert neben Auto!!!

   meinAuto.setFabrikat("Alfa Romeo");
   meinAuto.setType("GTV");
   meinAuto.eingebautMotor.setKW(110);
   meinAuto.eingebautMotor.setArt("V 6");
```

```
    std::string fabrikat = meinAuto.getFabrikat();
    std::string type = meinAuto.getType();
    std::string art = meinAuto.eingebautMotor.getArt();
    int kw = meinAuto.eingebautMotor.getKW();

    std::cout << "Auto Fabrikat = " << fabrikat  << " type = "
              << type << std::endl;
       // Auto Fabrikat = Alfa Romeo type = GTV
    std::cout << "Motor Art = " << art << " KW = " << kw << std::endl
              << std::endl;
       // Motor Art = V 6 KW = 110

    artAustausch = austauschMotor.getArt();
    kwAustausch = austauschMotor.getKW();
    std::cout << "MotorAustauschArt = " << artAustausch
            << " MotorAustauschKW = " << kwAustausch << std::endl;
       // MotorAustauschArt = V 6 MotorAustauschKW = 110

    return 0;
}
```

Programm-Ausgabe

```
Adresse von meinAuto 0xbffffc40
Adresse von eingebautMotor 0xbffffc10

MotorAustauschArt = ? MotorAustauschKW = 9999

Auto Fabrikat = Alfa Romeo type = GTV
Motor Art = V 6 KW = 110

MotorAustauschArt = V 6 MotorAustauschKW = 110

Aggregation has exited with status 0.
```

In der Programmausgabe sind wieder die Adressen der beiden Instanzen `meinAuto` und `eingebautMotor` interessant. Sie sind verschieden!

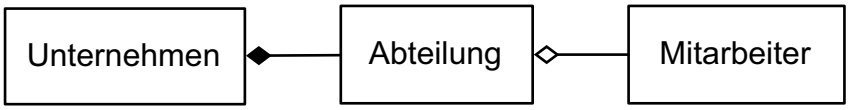
Die Ursache dafür ist an dem Konstruktor zu erkennen:

Source-Code

```
Auto::Auto (Motor & austauschMotor,
      std::string initFabrikat, std::string initType,
      std::string initArt, int initKW)
   : eingebautMotor(austauschMotor),
      fabrikat(initFabrikat), type(initType) {};
```

- Es wird eine Referenz `Motor &` `austauschMotor` übergeben. Dieser `austauschMotor` wird zwar gleichzeitig initialisiert, bleibt aber eine eigene Instanz (es sind verschiedene Adressen). Stirbt das Auto, bleibt der Motor bestehen!

An dem folgenden Bild ist der Unterschied zwischen Komposition und Aggregation zu erkennen:

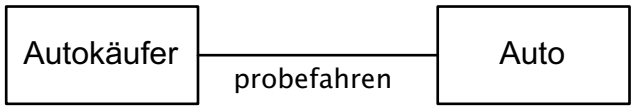


Komposition *Aggregation*

Wenn das Unternehmen stirbt, existiert auch die Abteilung nicht mehr. Der Mitarbeiter wird aber weiter leben!

4.4.4 Assoziation

Die Assoziation ist die schwächste Beziehung von Klassen untereinander.



Assoziation

Die Instanz der anderen Klasse existiert nur so lange, wie eine Funktion mit dieser Instanz ausgeführt wird.

Es folgt der vollständige Source-Text für dieses Beispiel.

- Hier taucht die andere Instanz in keinem Konstruktor auf. Sie wird erst in der Funktion `probeFahren()` bzw. `kaufenAuto()` als `probeAuto` bzw. `kaufAuto` instantiiert.

```
/*
 *  Autokaeufer.h
 *  Assoziation
 *
 *  Created by Peter Krauss on Sun Mar 28 2004.
 *  Copyright (c) 2004 TUM. All rights reserved.
 *
 */

#include "Auto.h"

class Autokaeufer {
public:
   std::string probeFahren();
   Auto kaufenAuto(std::string initFabrikat, std::string initType,
            std::string initArt, int initKW);
};
/*
 *  Autokaeufer.cpp
 *  Assoziation
 *
 *  Created by Peter Krauss on Sun Mar 28 2004.
 *  Copyright (c) 2004 TUM. All rights reserved.
 */
```

```
#include "Autokaeufer.h"

std::string Autokaeufer::probeFahren() {
   Auto probeAuto;
   std::string fabrikat = probeAuto.getFabrikat();
   std::string type = probeAuto.getType();
   std::string art = probeAuto.eingebautMotor.getArt();
   std::string kw = probeAuto.eingebautMotor.getKW();

   return ("Probefahrt mit " +fabrikat +" " +type +" mit Motor "
      +art +"/" + kw  +"KW");
};

Auto Autokaeufer::kaufenAuto(std::string initFabrikat, std::string
initType, std::string initArt, int initKW) {

   Auto kaufAuto(initFabrikat, initType, initArt, initKW);
   return (kaufAuto);
};
/*
 *  Auto.h
 *  Assoziation
 *
 *  Created by Peter Krauss on Sun Mar 28 2004.
 *  Copyright (c) 2004 TUM. All rights reserved.
 *
 */
#include "Motor.h"

class Auto {
public:
   Motor eingebautMotor; //Motor wird erzeugt
   Auto (std::string initFabrikat = "Alfa Romeo",
      std::string initType = "GTV",
         std::string initArt = "V 6", int initKw = 122);
   // Konstruktor Deklaration mit Std-Werten

   std::string getFabrikat (void);
   std::string getType (void);

   std::string probeFahren (void);

private:
   std::string fabrikat, type;
};
/*
 *  Auto.cpp
 *  Assoziation
```

```
 *
 *  Created by Peter Krauss on Thu Mar 28 2004.
 *  Copyright (c) 2004 TUM. All rights reserved.
 *
 */
#include "Auto.h"

Auto::Auto (std::string initFabrikat, std::string initType,
      std::string initArt, int initKW)
   : eingebautMotor(initArt, initKW),
      fabrikat(initFabrikat), type(initType) {}; // Konstruktor

std::string Auto::getFabrikat (void){
   return fabrikat;
};
std::string Auto::getType (void){
   return type;
};

/*
 *  Motor.h
 *  Assoziation
 *
 *  Created by Peter Krauss on Sun Mar 28 2004.
 *  Copyright (c) 2004 TUM. All rights reserved.
 *
 */
#include <string>

class Motor {
public:
   Motor (std::string initArt = "?", int initKw = 88);
      // Konstruktor Deklaration mit Std-Werten

   std::string getArt (void);
   std::string getKW (void);

private:
   std::string art;
   int kw;
};
/*
 *  Motor.cpp
 *  Assoziation
 *
 *  Created by Peter Krauss on Sun Mar 28 2004.
 *  Copyright (c) 2004 TUM. All rights reserved.
 */
```

```
#include <sstream>
#include "Motor.h"

Motor::Motor (std::string initArt, int initKw)
   : art(initArt), kw(initKw) {}; // Konstruktor

std::string Motor::getArt (void){
   return art;
};
std::string Motor::getKW (void){
   std::ostringstream KW; // wandeln int nach string
   KW << kw;
   return KW.str();
};
/*
 *  main.cpp
 *  Assoziation
 *
 *  Created by Peter Krauss on Thu Mar 28 2004.
 *  Copyright (c) 2004 TUM. All rights reserved.
 */
#include <iostream>
#include "Autokaeufer.h"

int main (int argc, char * const argv[]) {
   Autokaeufer hans;
   std::string probe = hans.probeFahren();
   std:: << probe << std::endl;
   // Probefahrt mit Alfa Romeo GTV mit Motor V 6/122KW

   Auto neuesAuto=hans.kaufenAuto("BMW", "700", "12 Zylinder", 300);
   // Jetzt ist hans stolzer Besitzer!!
   std::string fabrikat = neuesAuto.getFabrikat();
   std::string type = neuesAuto.getType();
   std::string art = neuesAuto.eingebautMotor.getArt();
   std::string kw = neuesAuto.eingebautMotor.getKW();

   std::cout << "Auto Fabrikat = " << fabrikat << " type = " << type
            << std::endl;
   // Auto Fabrikat = BMW type = 700
   std::cout << "Motor Art = " << art << " KW = " << kw <<std::endl;
   // Motor Art = 12 Zylinder KW = 300
   return 0;
}
```

Programm-Ausgabe

```
Probefahrt mit Alfa Romeo GTV mit Motor V 6/122KW
Auto Fabrikat = BMW type = 700
Motor Art = 12 Zylinder KW = 300
```

Die erste Ausgabe bezieht sich auf die Funktion `probeFahren()`. Es wurde gleich eine zweite Funktion `kaufenAuto()` implementiert, auf die sich die zweite Ausgabe bezieht. Das ist das Auto, das gekauft wurde.

Source-Code

```
std::string Autokaeufer::probeFahren() {
   Auto probeAuto;
...
}

Auto Autokaeufer::kaufenAuto(std::string initFabrikat, std::string
initType, std::string initArt, int initKW) {

   Auto kaufAuto(initFabrikat, initType, initArt, initKW);
   return (kaufAuto);
};
```

- Prinzipiell hätte es auch anders herum funktioniert. Die Funktion `probeFahren()` hätte auch in der Klasse Auto sein können. Das ist Grund, warum im Sinnbild kein Pfeil angegeben ist.

Das ist das besondere Kennzeichen der Assoziation, dass die Beziehung zu einer anderen Klasse nur während der Funktion, die die Assoziation zu der anderen Klasse darstellt, besteht und auch die Instanz als lokales Objekt nur während dieser Zeit existiert.

4.5 Polymorphismus (dynamisches Binden)

Begriff

Polymorphismus

- Fähigkeit einer Variablen, Objekte verschiedenen Typs zu enthalten.
- Bietet die Möglichkeit, verschiedene Objekte in gleicher Weise anzusprechen. Objekte können auf gleiche Botschaften verschiedenartig reagieren, wenn sie verschiedenen Klassen angehören.

4.5.1 Typ-Bindung

Die Zeiger auf Klassen haben in C++ eine interessante Eigenschaft:

Ein Zeiger, der auf Objekte einer bestimmten Klasse definiert ist, darf auch auf Objekte ihrer abgeleiteten Klassen zeigen.

Um das herauszufinden, benutzen wir wieder die Programme, die wir für die Mehrfachvererbung auf Seite 22 verwendet haben.

```
class A {
public:
   A (int i=800, int j=900) {
      this->i=i;
      this->j=j;
   };
```

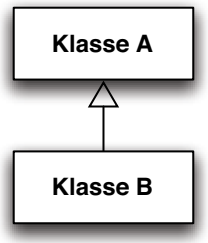
```
   void getZustand (void);
private:
   int i;
   int j;
};

void A::getZustand (void) {
   std::cout << "Ich bin Klasse A mit Zustand"
   << " (i= "<< i << " j= " << j << " this = " << this << ")"
   << std::endl;
};

class B : public A {
public:
   B (int i=101, int j=102) : A(111, 112) {
      this->i=i;
      this->j=j;
   };
void getZustand (void);
private:
   int i;
   int j;
};

void B::getZustand (void) {
   std::cout << "Ich bin Klasse B mit Zustand"
   << " (i= "<< i << " j= " << j << " this = " << this << ")"
   << std::endl;
   A::getZustand ();
};
```

4.5.2 Zeiger auf Klassen

Erster Versuch

Programm

```
A a;
B b;
a.getZustand();
b.getZustand();
```



Programm-Ausgabe

```
Ich bin Klasse A mit Zustand (i= 800 j= 900 this = 0xbffffcf0)
Ich bin Klasse B mit Zustand (i= 101 j= 102 this = 0xbffffd00)
Ich bin Klasse A mit Zustand (i= 111 j= 112 this = 0xbffffd00)
```

Das kommt uns bekannt vor: das Objekt der Klasse `B` enthält auch eine Instanz der Klasse `A`. Beide haben dieselbe Adresse.

Zweiter Versuch

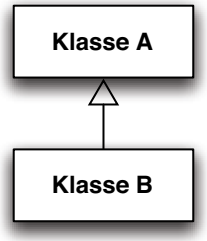
Jetzt werden Zeiger eingeführt, die auf diese Klassen zeigen. Über einen Zeiger soll bestimmt werden, von wem der Zustand ausgegeben werden soll.

Sourcecode

```
void gibMalZustand(A *aZeiger) {
   aZeiger->getZustand();
};
```

Programm

```
A a;
B b;
gibMalZustand(&a);
gibMalZustand(&b);
```



Programm-Ausgabe

```
Ich bin Klasse A mit Zustand (i= 800 j= 900 this = 0xbffffcf0)
Ich bin Klasse A mit Zustand (i= 111 j= 112 this = 0xbffffd00)
Hello, World!
```

Wieso gibt der Compiler keinen Fehler aus? Die Adresse `&b` ist doch nicht vom Typ `A`, den wir in der Funktion `gibMalZustand` definiert haben. Das ist der Beweis der obigen Aussage: `B` ist eine Sub-Klasse von `A`.

Wieso meldet sich aber dann zweimal die Klasse `A`? Wir haben doch Adresse `&b` angegeben! Eigentlich kein Wunder (aus der Sicht des Compilers): das Objekt `b` der Klasse `B` enthält ja eine Instanz von der Klasse `A` (gleiche Adressen). Trotzdem: der Programmierer kann damit nicht zufrieden sein. Wenn er auf `b` zeigt, dann will er auch `b` und nichts anderes.

Dritter Versuch

Jetzt machen wir den Gegenversuch.

Sourcecode

```
void gibMalZustand(B *bZeiger) {
   bZeiger->getZustand();
};
```

Programm

```
A a;
B b;
gibMalZustand(&a);
gibMalZustand(&b);
```

Compiler-Meldung

```
invalid conversion from `A*' to `B*'
```

Also hat vorher der Compiler eine Umwandlung von B* nach A* erfolgreich durchgeführt. Man könnte den Compiler mit einem `static_cast` veranlassen die Umwandlung vorzunehmen (wird in Abschnitt 5.6, Seite 70 behandelt):

```
gibMalZustand(static_cast<B*>(&a));
```

4.5.3 Virtuelle Methoden

Vierter Versuch

Uns hatte ja schon vorher das Wunderwort `virtual` geholfen. Versuchen wir es damit!

Sourcecode

```
void gibMalZustand(A *aZeiger) { // alter Zustand Typ A*
...
virtual void getZustand (void); // in Klasse A
```

Programm

```
A a;
B b;
gibMalZustand(&a);
gibMalZustand(&b);
```

Programm-Ausgabe

```
Ich bin Klasse A mit Zustand (i= 800 j= 900 this = 0xbffffce0)
Ich bin Klasse B mit Zustand (i= 101 j= 102 this = 0xbffffcf0)
Ich bin Klasse A mit Zustand (i= 111 j= 112 this = 0xbffffcf0)
Hello, World!
```

Jetzt funktioniert es, wie gewollt! Der Unterschied besteht darin: Im ersten Versuch hat eine Klassen-Typ-Bindung (frühe Bindung) stattgefunden. Die Funktion `gibMalZustand` hat als Parameter-Typ `A*` definiert. Folglich hat der Compiler eine Typ-Wandlung von `B*` nach `A*` durch geführt.

Mit dem Zusatz `virtual` wird eine Verwaltung der Objekte zur Laufzeit installiert (VTable). Man spricht nun von einer Objekt-Typ-Bindung (späte Bindung). Die Verantwortung liegt nun wirklich bei den Objekten, die ja erst existent sind, wenn sie erzeugt worden sind - also zur Laufzeit.

Implizit ist die Klasse `A` durch die virtuelle Methode `getZustand` zu einer virtuellen Basisklasse geworden.

Fünfter Versuch

Die Klasse `A` ist noch immer nicht vollständig virtuell, von ihr können noch immer ganz real Objekte instantiiert werden. Zu einer wirklichen virtuellen, nämlich abstrakten Klasse, wird sie wenn man die virtuelle Methode auf null setzt.

Sourcecode

```
virtual void getZustand (void) = 0;
```

Programm

```
A a;
B b;
gibMalZustand(&a);
gibMalZustand(&b);
```

Compiler-Meldung

```
error: because the following virtual functions are abstract:
error: virtual void A::getZustand()
error: cannot declare variable `a' to be of type `A'
```

Das bedeutet, dass es sich bei der Klasse `A` um eine abstrakte Klasse handelt, von der keine Objekte erzeugt werden können

4.5.4 Bedeutung von abstrakten Methoden und Klassen

Sechster Versuch

Weil nun die Klasse A abstrakt ist, werden wir auch keine Objekte mehr von ihr erzeugen

Programm

```
// A a;
   B b;
// gibMalZustand(&a);
   gibMalZustand(&b);
```

Programm-Ausgabe

```
Ich bin Klasse B mit Zustand (i= 101 j= 102 this = 0xbffffcf0)
Ich bin Klasse A mit Zustand (i= 111 j= 112 this = 0xbffffcf0)
Hello, World!
```

Die Programmausgabe zeigt, dass nach wie vor die Klasse `A` als Instanz existent ist, sie wird von der Klasse `B` geerbt. Selbst die abstrakte Methode `getZustand()` ist noch vorhanden und kann ausgeführt werden. Sie meldet sich in der Programm-Ausgabe.

Siebter Versuch

Wir gehen noch einen Schritt weiter: wenn die abstrakte Methode existent ist, können wir sie auch direkt ansprechen.

Programm

```
B b;
b.A::getZustand(); // Aufruf der abstrakten Methode
```

Programm-Ausgabe

```
Ich bin Klasse A mit Zustand (i= 111 j= 112 this = 0xbffffcf0)
Hello, World!
```

Also kann es mit „abstrakt“ und „virtuell“ nicht ganz so ernst sein.

Achter Versuch

Dieser Versuch zeigt, dass abstrakte und virtuelle Methoden doch ihre Berechtigung haben. Wir löschen die Methode `getZustand()` in der Klasse B.

Compiler-Meldung

```
error: because the following virtual functions are abstract:
error: virtual void A::getZustand()
error: cannot declare variable `b' to be of type `B'
```

Das bedeutet:

Eine abstrakte Methode muss in einer abgeleiteten Klasse implementiert werden!

Was uns nicht daran hindert, die abstrakte Methode zu benutzen. Eine abstrakte Methode ist also nicht nur eine Schablone, sondern im Sinne der Abstraktion der Objektorientierten Programmierung werden alle Dinge, die den abgeleiteten Klassen gemein sind, in ihr implementiert. Nur ist sie ohne die spezifischen Ergänzungen in den abgeleiteten Klassen unvollständig.

Von abstrakten Klassen können keine Objekte instantiiert werden.

Eine Klasse wird zur abstrakten Klasse, wenn sie eine rein virtuelle Funktion enthält, also eine virtuelle Funktion, die mit = 0 deklariert, aber nicht definiert ist.

4.5.5 Polymorphismus, praktisches Beispiel

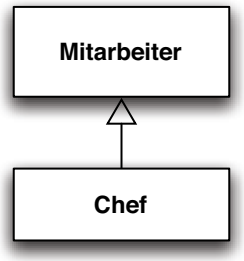
Im folgenden Beispiel gibt es eine Klasse für Mitarbeiter und eine Klasse für den Chef. Der einzige Unterschied besteht darin, dass der Chef mehr verdient als ein normaler Mitarbeiter. Deswegen erbt er von der Mitarbeiterklasse.

Beispiel: Klasse Mitarbeiter

```
class Mitarbeiter {
public:
   std::string vollerName() {
      return (this->vorname + " " + this->nachname);
   };
   int gehaltBerechnen(int stunden) {
      return stunden * 50;
   };

   std::string vorname;
   std::string nachname;
};
```

Beispiel: Klasse Chef

```
class Chef : public Mitarbeiter {
public:
   int gehaltBerechnen(int stunden) {
      return stunden * 150;
   }
};
```



Hauptprogramm

Für die Mitarbeiter wird ein Array angelegt:

```
int main () {
   Mitarbeiter *mitarbeiter[2];

   mitarbeiter[0] = new Mitarbeiter();
   mitarbeiter[0]->vorname = "Hans";
   mitarbeiter[0]->nachname = "Meier";
```

Eine einfache Funktion einen Mitarbeiter zu bearbeiten:

```
   std::cout << mitarbeiter[0]->vollerName() <<
      " verdient in 100 Stunden " <<
      mitarbeiter[0]->gehaltBerechnen(100) <<
      " Euro" << std::endl;
```

Der Chef wird in die Reihe der Mitarbeiter aufgenommen:

```
   mitarbeiter[1] = new Chef();
   mitarbeiter[1]->vorname = "Klaus";
   mitarbeiter[1]->nachname = "Schmidt";
```

Seine Bearbeitung unterscheidet sich nicht von einem Mitarbeiter:

```
    std::cout << mitarbeiter[1]->vollerName() <<
       " verdient in 100 Stunden " <<
       mitarbeiter[1]->gehaltBerechnen(100) <<
       " Euro" << std::endl;

    delete mitarbeiter[0];
    delete mitarbeiter[1];
    return 0;
}
```

Programm-Ausgabe

```
Hans Meier verdient in 100 Stunden 5000 Euro
Klaus Schmidt verdient in 100 Stunden 5000 Euro
Hello, World!
```

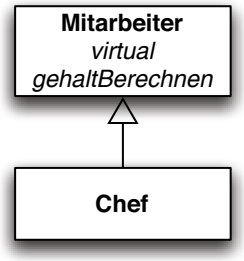
- Der Chef verdient genau soviel wie ein Mitarbeiter!

Dieses Problem ist uns schon aus dem zweiten Versuch in 4.5.2 geläufig. Er ist zwar der Chef, erbt aber von den Mitarbeitern, so dass eine frühe **Klassen-Bindung** stattfindet. Der Compiler wandelt den `Chef*` zum `Mitarbeiter*`.

Wir brauchen eine späte **Objekt-Bindung** zur Laufzeit! Das Objekt Chef muss berücksichtigt werden nicht die Klasse Mitarbeiter. Abhilfe bringt eine virtuelle Methode in der Klasse Mitarbeiter:

Source-Code

```
    virtual int gehaltBerechnen(int stunden) {
```



So weit, so gut: der Compiler ist nicht einverstanden:

Compiler-Meldung

```
main.cpp:3: `class Mitarbeiter' has virtual
functions but non-virtual destructor
main.cpp:17: `class Chef' has virtual func-
tions but non-virtual destructor
```

4.5.6 Virtuelle Destruktoren

Das Problem besteht darin, dass die Objekte mit `new` dynamisch erzeugt worden sind. Im Abschnitt Destruktoren auf Seite 57 wird gesagt, dass nur der Destruktor der Basisklasse aufgerufen wird, nicht jedoch der Destruktor von abgeleiteten Klassen, selbst wenn der Zeiger auf ein Objekt der abgeleiteten Klasse zeigt. Um das zu vermeiden, muss mindestens der Destruktor der Basisklasse `virtual` deklariert werden.

Source-Code

```
    virtual ~Mitarbeiter() {};
    virtual ~Chef() {};
```

Unser Problem ist damit behoben:

Programm-Ausgabe

```
Hans Meier verdient in 100 Stunden 5000 Euro
Klaus Schmidt verdient in 100 Stunden 15000 Euro
```

Es gibt sogar mal hilfreiche Meldungen des Compilers, die man auch sofort versteht! Wegen der Angabe `virtual` beim Destruktor wird er in die Liste VTable der virtuellen Methoden aufgenommen und zur Laufzeit (dynamisch) verwaltet.

4.6 Abstrakte und konkrete Klassen

Begriffe

abstrakte Klasse

- Klasse, für die keine Objekte instantiierbar sind
- dient als Muster für abzuleitende Unterklassen

konkrete Klasse

- Klasse, die Instanzen haben kann
- zur Unterscheidung von abstrakten Klassen

Abstrakte und konkrete Klassen in C++

Beispiel:

siehe Abschnitt 4.3.2

4.7 Generische/parametrierte Klassen

Begriff:

Generische Klassen (parametrierte Klassen)

- Generische Klassen beschreiben eine ganze Familie von einander sehr ähnlichen Klassen.
- Ausgehend von einer generischen Klasse müssen zunächst richtige Klassen instantiiert werden (abhängig von Parametern), dann können Objekte der instantiierten Klasse instantiiert werden.
- Einsetzbar z.B. zur Implementierung von Listen oder Arrays, die beliebige Objekte enthalten können.

Generische Klassen in C++:

- Generische Klassen finden sich in C++ in Form von Templates. Dies sind Rahmen für eine Familie von Klassen.
- Templates haben eine gewisse Ähnlichkeit zu Makros. Sie bringen keine Laufzeitnachteile mit sich.
- Templates sind nicht verfügbar in älteren Implementierungen von C++.

Beispiel:

```
template<class T> class Vektor {
  T *v;
  int groesse;
public:
```

```
  Vektor(int g) {
    if (g <= 0) error("falsche Vektorgroesse");
    v = new T[groesse=g]; //reserviere Speicherplatz fuer g Elemente
                          // vom Typ T
  }
  T &operator[](int i);     //Operatordeklaration vgl. Abschnitt 5.5
  int getGroesse() {return groesse;}
  // ...
};

//Definition des Vektors:
main() {
  Vektor<int> v1(100);   //Vektor von 100 Integer-Werten
  Vektor<complex> v2(50);//Vektor von 50 Werten eines Typs complex,
                         //der z.B. als Klasse definiert werden kann
  // ...
  v1[i] = 17;
  // ...
}
```

Weitere Eigenschaften von Templates:

- Neben Klassen-Templates gibt es auch Funktions-Templates, z.B.

```
template<class T> void sort(Vektor<T>&);
```

- Das Schlüsselwort class gibt hier an, dass T als Parameter für einen Typ steht (z.B. **`int`**, **`float`** oder ein benutzerdefinierter Typ).
- Templates für andere konstante Ausdrücke sind ebenfalls erlaubt, z.B. ein Template für einen Integer-Wert.

```
template<int groesse> class KurzerString {
  // ...
private:
  char inhalt[groesse];
}

main() {
  KurzerString<10> meinString;
  // ...
}
```

5 Weitere Sprachelemente in C++

5.1 Überladen von Funktionen und Methoden

Begriff:

Überladene Funktionen (Overloaded Functions) und Methoden entstehen durch die Vergabe eines Funktionsnamens an mehrere verwandte Funktionen bzw. an mehrere Methoden, die unterschiedliche Parameter haben müssen.

Beispiel 1:

```
int wurzel(int i);
float wurzel(float f);
double wurzel(double f);
```

Alle drei Funktionen haben denselben Namen. Wird die Funktion `wurzel` aufgerufen, so sucht der Compiler auf Basis der Anzahl und des Typs der Parameter die richtige Funktion aus. Der Funktionsname `wurzel` ist also überladen.

```
void main() {
  int ia, ib;
  float fa, fb;
  ia = wurzel(ib);
  fa = wurzel(fb);
}
```

Es ist jedoch nicht möglich, zwei Funktionen zu deklarieren, die sich ausschließlich im Typ des Rückgabewertes unterscheiden, da der Compiler nur die Parameterlisten zur Unterscheidung von überladenen Funktionen heranzieht.

Beispiel 2:

```
int search(char *key);
char *search(char *name);  // Fehler! Gleiche Parameterliste.
```

Hinweis:

Es ist nicht sinnvoll, Funktionen, die nichts miteinander zu tun haben, den gleichen Funktionsnamen zu geben!

5.2 Funktionen/Methoden mit Default-Argumenten

Begriff:

In C++-Funktions-/Methodenprototypen können Standardwerte für die Parameter angegeben werden (*Default-Argumente*).
Diese Standardwerte werden dann verwendet, wenn beim Aufruf der Funktion keine entsprechenden Parameter übergeben werden.

Beispiel:

```
void meineFunktion(int i = 5, double d = 3.23);
```

Hier sind z.B. folgende Aufrufe zulässig:

```
meineFunktion(12, 5.31);// überschreibt beide Default-Argumente
meineFunktion(4); // als Wert für d wird 3.23 eingesetzt,
                  // entspricht meineFunktion(4, 3.23);
meineFunktion;    // entspricht meineFunktion(5, 3.23);
```

Unzulässig sind jedoch beispielsweise:

```
meineFunktion(, 3.5);         // unzulässiger Aufruf
void andereFunktion(int i=5, double d);// unzulässige Deklaration
```

Man kann also die Liste der Parameter nur lückenlos verkürzen.

```
void andereFunktion(int i, double d=3.23);// zulässige Deklaration!
```

5.3 Referenzen

Eine Referenz ist ein weiterer Name für eine bereits vorhandene Variable. Definiert wird eine Referenz etwa wie folgt:

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

`a` ist eine gewöhnliche `int`–Variable und `b` ist eine `int`–Referenz (kenntlich gemacht durch das vorangestellte `&` - Referenzoperator). Eine Referenz muss bei ihrer Definition sofort initialisiert werden mit einer Variablen (oder einem anderen Ausdruck, der einen Speicherbereich vom entsprechenden Typ identifiziert!) und die Referenz ist ein weiterer Name für die bei der Initialisierung angegebene Variable (bzw. den angegebenen Speicherbereich!).

5.3.1 Referenzen als Alias-Bezeichner

In obigem Beispiel sind `a` und `b` zwei Namen für eine und dieselbe Variable (wobei `a` die eigentliche Variable und `b` nur ein anderer Name für diese ist!).

Diese Referenz kann wie die Variable verwendet werden:

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

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

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

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

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

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

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

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

Ebenso unzulässig sind Adress–Variablen auf Referenzen:

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

Eine Referenz auf eine Adress–Variable hingegen ist möglich

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

Angewendet werden Referenzen hauptsächlich im Zusammenhang mit Funktionen!

5.3.2 Referenzen als Funktionsparameter

Wird eine Funktion wie folgt definiert:

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

und wie folgt aufgerufen:

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

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

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

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

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

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

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

- über Adressen (wie in C):

```
void fkt(int *);  //Deklaration
fkt(&i);          //Aufruf
```

- über Referenzen:

```
void fkt(int &);  //Deklaration
fkt(i);           //Aufruf
```

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

```
void fkt(int); //Deklaration
fkt(i);        //Aufruf
```

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

5.3.3 Referenzen als Funktionsergebnisse

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

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

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

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

Vernünftige Anwendungen sind:

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

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

oder

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

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

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

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

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

- in Zusammenhang mit In–/Dekrementierung:

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

  das Funktionsergebnis wird inkrementiert!

- in Zusammenhang mit Adressen:

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

  die Adresse des Funktionsergebnisses wird einer Adress–Variablen zugewiesen!

  – in Zusammenhang mit einer Zuweisung:

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

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

5.3.4 Zusammenfassung

Eine Referenz ist keine Kopie der Variablen, sondern dieselbe Variable unter einem anderen Namen.

Folgende Aktionen können nicht mit Referenzen selbst durchgeführt werden, sie wirken automatisch auf das referenzierte Objekt:

- Verwenden eines Zeigers auf die Referenz
- Verwenden ihrer Adresse
- Vergleichen oder Zuordnen von Werten
- Arithmetische Berechnungen durchführen
- Modifizieren

Beispiele

- In C gibt es zwei Möglichkeiten, Variablen als Parameter an eine Funktion zu übergeben: Übergabe der Variablen selbst oder Übergabe als Zeiger.
- In C++ besteht zusätzlich die Möglichkeit, eine Referenz zu übergeben, die Funktion erhält also einen Alias-Bezeichner der Original-Variablen.

```
// Parameterübergabe an Funktionen
#include <iostream>

// eine grosse struct
struct bigone {
  int nummer;
  char text[1000];
} bo = {123, "Dies ist eine GROSSE struct"};

// Drei Funktionen mit der struct als Parameter:
void varfunc(bigone v1);             // Übergabe als Variable
void zgrfunc(const bigone *z1);      // Übergabe als Zeiger
void reffunc(const bigone &r1);      // Übergabe als Referenz

int main() {
  varfunc(bo);    // Übergabe der Variablen selbst
  zgrfunc(&bo);   // Übergabe der Adresse der Variablen
```

```
  reffunc(bo);    // Übergabe einer Referenz auf die Variable
}

// Übergabe als Variable
void varfunc(bigone v1) {
  std::cout << '\n' << v1.nummer;
  std::cout << '\n' << v1.text;
}

// Übergabe als Zeiger
void zgrfunc(const bigone *z1) {
  std::cout << '\n' << z1->nummer;     // Zeiger-Schreibweise
  std::cout << '\n' << z1->text;
}

// Übergabe als Referenz
void reffunc(const bigone &r1) {
  std::cout << '\n' << r1.nummer;      // Referenz-Schreibweise
  std::cout << '\n' << r1.text;
}
```

- Kleine Variablen (Standardtypen wie **`int`**, **`float`**, ...) werden am effizientesten selbst als Parameter übergeben.

5.4 Konstruktoren/Destruktoren und dynamische Speicherverwaltung

5.4.1 Konstruktoren/Destruktoren

Konstruktoren

- Ein Problem von C-Strukturen ist die Tatsache, dass sie keinen definierten Initialisierungszustand haben.
- Wird in C Speicherplatz für eine Struktur belegt (in Form einer Variablendefinition oder mit `malloc`), so müssen die Werte der **`struct`** explizit festgelegt werden.
- In C++-Klassen gibt es hierfür Konstruktoren.
- Ein Konstruktor ist eine Funktion, die automatisch aufgerufen wird, wenn ein Objekt generiert wird.
- Der Konstruktor ist eine Mitgliedsfunktion einer Klasse. Sie muss den gleichen Namen haben wie die Klasse selbst.

Beispiel:

```
class Font {
   public:
   int   Size;
   Font(int s) {Size = s;}    // Konstruktor
   ~Font() {}                 // Destruktor, siehe unten
```

```
    ...
};

main() {
  Font meinFont(10);      //Deklarieren eines Objektes der Klasse
Font
  Font myotherFont = 10;  // bei einem Parameter möglich
  ...
}
```

Bei der Deklaration der Variablen wird der Konstruktor aufgerufen und ausgeführt, dabei können Parameter übergeben werden. Der Variablen `Size` wird hier der übergebene Wert `10` zugewiesen.

Eigenschaften von Konstruktoren:

- Die Initialisierung von Werten mit Hilfe des Konstruktors ist vom Compiler effizienter zu realisieren als die nachträgliche Zuweisung von Werten.
- Der Konstruktor kann bei entsprechender Definition die Gültigkeit von Werten vor der Zuweisung überprüfen.
- Ein Konstruktor hat **keinen** Rückgabewert (nicht einmal **`void`**).
- Ein Konstruktor kann mit Default-Argumenten deklariert werden (siehe Abschnitt 5.2).
- Eine Klasse kann mehrere Konstruktoren haben, es ist also möglich den Konstruktor durch verschiedene Parameterlisten zu überladen.
- Der Compiler generiert:
  - einen Default-Konstruktor ohne Parameter, der aber keine Initialisierung vornimmt (siehe Abschnitt 5.4.2)
  - einen Default-Copy-Konstruktor (siehe Abschnitt 5.4.5)

Destruktoren

- Der Destruktor ist das Gegenstück zum Konstruktor.
- Der Name wird aus dem Klassennamen und einer vorangestellten Tilde (~) gebildet.
- Der Destruktor führt notwendige „Aufräumarbeiten“ durch, bevor der Speicher des Objektes freigegeben wird. Beispielsweise muss dynamisch zugeteilter Speicher explizit wieder freigegeben werden.
- Der Speicher der Daten innerhalb der Klasse wird automatisch freigegeben.
- Destruktoren können nicht überladen werden, es gibt genau einen Destruktor pro Klasse.
- Beim Aufruf von Destruktoren für dynamisch allozierte Objekte kann ein Problem auftreten:
  - Wird **`delete`** auf einen Zeiger auf eine Basisklasse angewandt, wird nur der Destruktor der Basisklasse aufgerufen, nicht jedoch der Destruktor von abgeleiteten Klassen, selbst wenn der Zeiger auf ein Objekt der abgeleiteten Klasse zeigt.
  - Um dies zu vermeiden, muß der Destruktor der Basisklasse als **`virtual`** deklariert werden, dann wird in jedem Fall der richtige Destruktor aufgerufen.

- kein Problem, wenn der auto_ptr aus der Standardbibliothek verwendet wird (siehe Abschnitt 5.4.9)

Beispiel für Destruktor siehe oben.

5.4.2 Standard-Konstruktor (default constructor)

Der Standard–Konstruktor hat keine Parameter:

```
A::A();
```

- Er sorgt dafür, dass für das Objekt ausreichend Speicher reserviert wird — der Speicherinhalt wird nicht weiter behandelt.
- Er wird immer implizit vom System dann aufgerufen, wenn ein neues A–Objekt ohne weitere Angaben erzeugt werden soll:

```
A a,b,c;       // 3-mal parameterloser Konstruktor
A aFeld[100];  // 100-mal
```

- Alternativ wäre auch folgende Schreibweise möglich:

```
A a();      // lieber nicht!
            // Compiler-Fehlermeldung bei Methoden-Aufruf
```

5.4.3 Konstruktoren, selbstgeschrieben

Hat eine Klasse einen selbstdefinerten, parameterbehafteten Konstruktor, so gibt es nicht mehr den ansonsten vom System automatisch bereitgestellten parameterlosen Konstruktor (default constructor)! Stellt der Klassenentwickler also selbst keinen parameterlosen Konstruktor zusätzlich zur Verfügung, können keine „Standard–Objekte“ mehr erzeugt werden!

```
Bruch a;       // FEHLER: kein parameterloser Konstruktor vorhanden
Bruch feld[10];// FEHLER: kein parameterloser Konstruktor vorhanden
...
```

Möchte der Klassen–Entwickler, dass Standard–Objekte (ohne explizites Argument) seiner Klasse erzeugt werden können, so muss er dann selber dafür sorgen, dass ein ohne Argumente aufrufbarer Konstruktor definiert wird.

Möglichkeiten hierzu:

- Er deklariert und definiert den parameterlosen Konstruktor, ggf. mit leerem Anweisungsteil:

```
class Bruch {
private:
   ...
public:
   ...
   Bruch() // parameterloser Konstruktor
      { } // leerer Anweisungsteil ist OK!
...
};
```

- oder er versieht einen parameterbehafteten Konstruktor so mit Defaultwerten, dass er auch ohne Argumente aufrufbar ist:

```
class Bruch {
private:
   int zaehler;
```

```
   int nenner;
public:
   ...
   Bruch(int z = 0, int n = 1){
      zaehler = z;
      nenner = n;
   }
   ...
};
...
Bruch c(3,4);  // Zaehler 3, Nenner 4
Bruch b(7);    // Zaehler 7, Nenner 1
Bruch a;       // Zaehler 0, Nenner 1
               // a() macht wieder Probleme!
               //ganz verkürzte Liste ist nicht mehr da, nicht leer!
...
```

5.4.4 Konstruktoren, Initialisierungslisten

Hat ein Objekt einer Klasse B eine Komponente einer anderen Klasse A, so wird standardmäßig bei jedem Konstruktor für B, bevor der Anweisungsteil des B–Konstruktors durchgeführt wird, zunächst der parameterlose Konstruktor für die A–Komponente aufgerufen! Insbesondere muss ein parameterloser A–Konstruktor verfügbar sein!

```
class A {
...
public:
   A (int); // einziger Konstruktor fuer A!
...
};
class B {
private:
   A a_komp1; // erste A-Komponente
   A a_komp2; // zweite A-Komponente
...
public:
   B(int i)
// an dieser Stelle wird versucht, fuer die A-Komponenten jeweils
// den parameterlosen A-Konstruktor aufzurufen.
// Da es diesen nicht gibt, meldet der Compiler einen FEHLER!
};
```

Abhilfe bietet hier eine sogenannte Initialisierungsliste: Bei der Implementierung des B–Konstruktors kann hinter der Parameterliste, vor dem Anweisungsteil ein Doppelpunkt und anschließend eine Liste von Konstruktoraufrufen für eventuelle Komponenten aufgeführt sein, etwa:

```
B(int i) : a_komp1(i), a_komp2(2*i+5)// Initialisierungsliste
{ ... }
```

Hier wird für die A–Komponente `a_komp1` der Konstruktor `A(int)` mit dem Argument `i` (vom B–Konstruktor stammender Parameter) aufgerufen und für die zweite A–Komponente `a_komp2` derselbe Konstruktor, aber mit Argument `2*i+5`. (Der parameterlose Konstruktor für `A` wird also nicht mehr benötigt)

5.4.5 Der Copy-Konstruktor

Begriff:

Copy-Konstruktoren sind Konstruktoren, die eine Referenz auf ein Objekt des gleichen Typs als Parameter haben.

Eigenschaften:

- Copy-Konstruktoren werden aufgerufen, wenn ein Objekt mit dem Wert eines anderen Objektes initialisiert wird.
- Copy-Konstruktoren werden bei der Initialisierung entweder über einen Funktionsaufruf oder über den Operator = aufgerufen.
- Copy-Konstruktoren werden außerdem implizit aufgerufen, wenn ein Objekt als Parameter oder als Rückgabewert eines Funktionsaufrufs verwendet wird.
- Der Copy-Konstruktor wird automatisch vom Compiler generiert und kopiert alle Elemente des vorhandenen Objektes in das neu zu initialisierende Objekt.
- Wenn dies zu Problemen führt, z.B. dann, wenn das Objekt Zeiger enthält, muss ein eigener Copy-Konstruktor (also ein Konstruktor mit einer Referenz auf dieselbe Klasse als Parameter) definiert werden.

Standard–Copy–Konstruktor (default copy constructor)

Der Standard-Copy-Konstruktor ist definiert:

```
A::A(const A &);
```

und sorgt dafür, dass ein neues Objekt der Klasse `A` als Kopie eines vorhandenen Objektes der gleichen Klasse erzeugt werden kann. Dieser Konstruktor sorgt dafür dass

- zunächst ausreichend Speicherplatz für das neue Objekt zur Verfügung gestellt wird (wie beim parameterlosen Konstruktor). Vorsicht bei Arrays - siehe Beispiel 2.
- darüberhinaus erhält jede Datenkomponente des neuen Objektes den gleichen Wert, wie die entsprechende Komponente des vorhandenen Objektes (bei Standardtypen als Komponenten wird byteweise kopiert, bei Objekten als Komponenten wird für diese wiederum der zum Komponententyp gehörende Copy– Konstruktor aufgerufen).

Beispiel 1: Standard–Copy–Konstruktor

```
class Font {
public:
  ...
  Font(int s);   // erster Konstruktor
};

main() {
  Font meinFont(10);       // Aufruf des ersten Konstruktors
  Font font1(meinFont);    // Aufruf des Copy-Konstruktors
  Font font2 = meinFont;   // Aufruf des Copy-Konstruktors
  ...
}
```

Beispiel 2: Definition eines eigenen Copy-Konstruktors

```
class Font {
public:
  Font(int s);    // erster Konstruktor
  Font(Font &f);  // eigener Copy-Konstruktor
private:
  int Size;
  char *Bezeichnung;
  ...
};

Font::Font(Font &f) {
  Size = f.Size;
  // Kopieren der Bezeichnung
  Bezeichnung = new char[strlen(f.Bezeichnung) + 1];
  strcpy (Bezeichnung, f.Bezeichnung);
}
```

Beispiel 3: Weitere Aufrufe des Copy-Konstruktors

```
// Funktion mit Klasse als Parameter, Copy-Konstruktor wird beim
// Aufruf dieser Funktion aufgerufen:
void meineFunktion(Font meinFont) {
  ...
}
// Funktion mit Klasse als Rückgabetyp, Copy-Konstruktor wird beim
// Rücksprung von dieser Funktion aufgerufen:
Font andereFunktion() {
  Font tmp(10);
  return tmp;
}
```

5.4.6 Konstruktoren für abgeleitete Klassen

Konstruktoren für abgeleitete Klassen können auf Konstruktoren der Basisklasse zurückgreifen.

Beispiel:

```
enum fontType {normal, fett, kursiv};

class SpecialFont : public Font {
public:
   SpecialFont(int s, fontType ft);
   fontType Type;
   ...
};

SpecialFont::SpecialFont(int s, fontType ft) : Font(s) {
   type = ft;
};
```

5.4.7 Konstruktoren für Objekte innerhalb von Klassen (Member Initializer)

Enthält eine Klasse Objekte als Mitglieder (Enthalten-Relation, vgl. Abschnitt 3.4), so wird ein Initialisierer für die Klasse der enthaltenen Objekte benötigt. Der Initialisierer ist eine Initialisierungsliste im Konstruktor, die im Abschnitt 5.4.4 beschrieben worden ist. (Beispiele in Abschnitte Komposition auf Seite 29 und Aggregation auf Seite 33.)

Beispiel (`Font` vgl. 5.4.1**):**

```
class Text {
  Text(char *txt, int font_groesse); // Konstruktor
  Font textFont1, textFont2;
  char *inhalt;
  // ...
}
Text::Text(char *txt, int font_groesse)
    : textFont1 (font_groesse), textFont2 (12) {
  inhalt = txt;
}
```

Der Konstruktor der Klasse Text ruft also seinerseits den Konstruktor der Klasse Font auf. Alternativ kann die Größe des Fonts nachträglich angegeben werden:

```
Text::Text(char *txt, int font_groesse) {
  inhalt = txt;
  textFont1.setSize(font_groesse);
  textFont2.setSize(12);
}
```

Dabei wird jedoch der Default-Konstruktor für `Font` verwendet, der die Größe nicht initialisiert und erst anschließend die Funktion `Font::setSize()` aufruft.

Hinweis:

Wären `textFont1` oder `textFont2` konstante Objekte (Vgl. Abschnitt 5.12.4), so wäre diese Vorgehensweise nicht möglich.

5.4.8 Dynamische Speicherverwaltung mit new/delete

Der `new` Operator

Anstatt der aus C bekannten Funktion `malloc` zur dynamischen Zuteilung von Speicher wird in C++ der Operator **`new`** verwendet.

Beispiel 1: C

```
Font *fontZeiger;
fontZeiger = (Font *)malloc(sizeof(Font));
```

Beispiel 2: C++

```
Font *fontZeiger1, *fontZeiger2;
fontZeiger1 = new Font;           // Default-Konstruktor
fontZeiger2 = new Font(10);       // Konstruktor aus 5.4.1
```

Vorteile von `new`:

- Ein explizites „Casten“ (Typumwandlung) des zugeteilten Speichers ist nicht erforderlich. So ist auch eine Typprüfung des Zeigers möglich.

- **new** stellt selbständig die nötige Speichergröße fest. Die Verwendung der `sizeof`-Funktion ist unnötig.
- **new** ruft automatisch den passenden Konstruktor auf, das Objekt kann so nach Wunsch initialisiert werden.
- Die Syntax für **new** ist wesentlich übersichtlicher als die von `malloc`.

Weitere Eigenschaften von `new`:

- Falls **new** den angeforderten Speicher nicht zuteilen kann, gibt es den Wert `0` zurück. Hinweis: Der Null-Zeiger in C++ hat den Wert `0`. `#define NULL 0`; in C: `#define NULL ((void*)0)`.
- **new** kann auch für Standardtypen (z.B. **int**) und Arrays (Syntax siehe Beispiel 3!) verwendet werden. Auch mehrdimensionale Arrays sind möglich, solange die Dimensionen konstant sind (bis auf die erste).

Beispiel 3:

```
// integer:
int *int_zeiger;
int_zeiger = new int;

// array:
int laenge
char *char_zeiger;
laenge = 10;
char_zeiger = new char[laenge];

// mehrdimensionaler array
int (*matrix)[10];
int groesse;
groesse = 8;
matrix = new int[groesse][10];
```

Der `delete` Operator

- **delete** ist das Gegenstück zu **new**.
- **delete** ruft automatisch den Destruktor auf und gibt anschließend den Speicher des Objektes frei.
- Wichtig:
  - **delete** darf nur für Zeiger aufgerufen werden, die mit **new** belegt wurden.
  - Ein mehrfacher Aufruf von **delete** für den gleichen Speicherbereich ist unzulässig und kann zum Absturz führen (schwer zu entdeckender Fehler!!!).
  - Der Aufruf von **delete** für Null-Zeiger ist dagegen ungefährlich.

Beispiel 4: (vgl. Beispiel 2)

```
delete fontZeiger1;  // hier wird der Destruktor von Font aufgerufen
delete fontZeiger2;  // hier wird der Destruktor von Font aufgerufen
```

Die Syntax für den Aufruf von **delete** für Arrays sollte besonders beachtet werden:

Beispiel 5: (vgl. Beispiel 3)

```
delete int_zeiger;
delete [] char_zeiger;
delete [] matrix;
```

Klassen mit Zeigerelementen

Auf das Problemfeld „Zeigerelemente in Klassen“ wird in den Übungsaufgaben eingegangen.

5.4.9 Smart-Pointer auto_ptr (C++Standardbibliothek)

Die Templateklasse `auto_ptr` der C++Standardbibliothek ist eine Form eines Smart-Pointers. Als Smart-Pointer bezeichnet man Klassen, die Zeiger auf dynamisch erstellte Objekte verwalten. Die Intelligenz eines Smart-Pointers liegt darin, dass er sich komplett um das Memory-Management kümmert. Dazu gehören Dinge wie das Verhindern von Dangling-Pointern (also Zeigern die auf nicht mehr existierende Objekte zeigen) und die automatische Speicher- bzw. Resourcenfreigabe für das referenzierte Objekt. Damit sich ein Smart-Pointer wie ein normaler Zeiger verhalten kann, muss er die Schnittstelle eines Zeigers besitzen. Diese besteht mindestens aus dem Dereferenzierungsoperator (operator`*`) und dem Indirektionsoperator (operator`->`). Da es für die Umsetzung des Memory-Managements unzählige Möglichkeiten gibt (Besitztransfer, Referenzzählung usw.), existieren auch unzählige verschiedene Arten von Smart-Pointer-Klassen. - Hier soll es nun nur um die Standard C++ `auto_ptr` gehen.

Grundsätzlich haben `auto_ptr` die folgenden Vorteile gegenüber normalen Pointern:

- automatische Freigabe von Speicher und Resourcen
- automatische Initialisierung. Entweder mit einem Heapobjekt oder mit NULL
- korrektes Freigabe von Speicher und Resourcen auch im Falle von Exceptions

Beispiel: Zeiger auf Festpunktzahl

```
std::auto_ptr<int> festpunktZeiger(new int);
*festpunktZeiger = 1;
std::cout << *festpunktZeiger << std::endl;
```

- Im folgenden Bespiel lässt sich verfolgen, wie `auto_ptr` die Klasse `X` verwaltet. In Abhängigkeit einer Zufallszahl wird eine Fehlersituation (Exception) simuliert. Das setzt allerdings Kenntnisse über die Fehlerbehandlung in C++ voraus (wird in Kapitel 5.10 behandelt).

Verwaltung einer Klasse mit auto_ptr

```
class X {
public:
   X() {std::cout << "X Standard-Konstruktor" << std::endl;}
   X(const X& re) {std::cout << "X Copy-Konstruktor" << std::endl;}
   ~X() {std::cout << "X Destruktor" << std::endl;}
   void MightThrow() {
      if (rand() % 2)
         throw int(42);
   }
};
int main (int argc, char * const argv[]) {

   srand(time(NULL));
```

```
    try {
       std::auto_ptr<X> Pointer(new X);
       Pointer->MightThrow();
    }
    catch(...){ // echte ... catched alles!
       std::cout << "Holla! Eine Exception!" << std::endl;
    }
    std::cout << "Hello, World!\n";
    return 0;
}
```

5.4.10 Der this-Zeiger

Innerhalb einer Member–Funktion kann auf jede Komponente des aktuellen Objektes - das Objekt, für welches die Member–Funktion aufgerufen wurde - zugegriffen werden. Innerhalb einer Member–Funktion zu einer Klasse `A` gibt es automatisch einen konstanten Zeiger auf ein Objekt vom Typ `A` - und dieser Zeiger hat den Namen `this`.

- Die exakte Definition dieses Zeigers müsste also lauten:

```
c * const this;
```

Dieser Zeiger zeigt innerhalb der Member–Funktion auf das aktuelle Objekt, und da der Zeiger `const` ist, kann dieser Zeiger auch nicht geändert werden (das, worauf der Zeiger zeigt - also das aktuelle Objekt - kann über den Zeiger sehr wohl verändert werden!).

In einer Member–Funktion könnte man über diesen `this`–Zeiger auf Komponenten (Daten oder Funktionen) des aktuellen Objektes zugreifen:

```
...this->Komponentenname...
```

(Hier würde es der reine Komponentenname auch tun!) über diesen `this`–Zeiger hat man aber auch Zugriff auf das ganze aktuelle Objekt, etwa, um das aktuelle Objekt als Funktionsergebnis zurückzugeben

```
class A {
   ...
public:
   A& A_fkt(...); // A-Member-Funktion, welche Referenz auf
                  // ein A zurückgibt
   ...
};
```

- Definition dieser Funktion

```
A& A::A_fkt(...){
   ...
   return *this; // gebe aktuelles Objekt zurück
}
```

- oder eine andere Funktion mit dem aktuellen Objekt (oder seiner Adresse) als Argument aufzurufen:

```
class A {
   ...
public:
   void A_fkt(void);
   ...
```

```
};
...
```

- globale Funktion

```
void glob_fkt(A *a){
   ...
}
```

- Definition der A-Member-Funktion

```
void A::A_fkt(void){
   ...
   glob_fkt(this);   // rufe globale Funktion mit Adresse des
                     // aktuellen Objektes als Argument auf
   ...
}
```

Statische Methoden (vgl. Abschnitt 5.12.6) müssen ohne `this` auskommen.

5.5 Überladen von Operatoren

Operatoren, wie z.B. +, -, < oder = sind zunächst nur für Standardtypen definiert und können nicht auf benutzerdefinierte Typen wie Klassen angewandt werden.

Beispiel:

```
class String {
 char *inhalt;
 int laenge;
 // ...
}

main() {
 String str_a, str_b;
 // ... (Belegen von String str_a mit Inhalt)

 str_b = str_a; // Dies macht hier keinen Sinn, der Operator "="
                // ist nicht definiert für String.

 if(str_b < str_a) { /* ... */ };
 // Ebenfalls nicht möglich, da "<" nicht definiert ist für String
}
```

In C++ können solche Operatoren für eigene Datentypen definiert werden. Sie erhalten eine zusätzliche Bedeutung, die abhängig ist vom Typ des Parameters, werden also überladen.

Eigenschaften:

- Folgende Operatoren können überladen werden:

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

- Es können keine eigenen zusätzlichen Operatoren gebildet werden.
- Verschiedene Operatoren können auf einen oder auf mehrere Operanden angewandt werden (z.B. steht „-“ mit zwei Operanden für die Subtraktion, mit einem Operanden für die Negierung des Wertes), andere Operatoren, wie z.B. „~“ sind für einen Operanden anwendbar und können nicht für zwei Operanden definiert werden.
- Die Reihenfolge der Auswertung von Operatoren (z.B. Punkt-vor-Strich) kann nicht umdefiniert werden.
- Die Operatoren

  ```
  .   .*   ::   ?:
  ```

  können nicht überladen werden.
- Operatoren sollten nur überladen werden, wenn die Bedeutung intuitiv klar ist, ansonsten sind Funktionen, die per Funktionsnamen aufgerufen werden, klarer. Der Aufruf

  ```
  menge1 && menge2
  ```

  für Objekte einer selbstdefinierten Klasse Menge könnte z.B. sowohl als Schnittmenge als auch als Vereinigungsmenge aufgefaßt werden und wäre hier nicht klar.

5.5.1 Überladen innerhalb einer Klasse

Der erste Parameter ist immer ein Objekt der Klasse, in der überladen wird.

Formate

```
• Rückgabetyp operatorZeichen (Parameter)
        // für: Objekt Zeichen Parameter Infix-Operator
• Rückgabetyp operatorZeichen ()
        // für: Objekt Zeichen Präfix-Operator
• Rückgabetyp operatorZeichen (int)
        // für: Objekt Zeichen Postfix-Operator;
           // (int) nur für Unterscheidung
• Rückgabetyp operator[] (Parameter)
        // für: Objekt [Parameter] Index-Operator
```

Beispiel 1: Operatoren, die innerhalb von Klassen deklariert sind

```
class MeinString {
public:
   char *inhalt;
   int laenge;
   void operator=(const MeinString &zweiterString);
   int operator<(const MeinString &zweiterString) const;
   ...
}

void MeinString::operator=(const MeinString &zweiterString) {
   laenge = zweiterString.laenge;
   delete [] inhalt;
   inhalt = new char[laenge+1];
   strcpy(inhalt, zweiterString.inhalt);
}
```

```
int MeinString::operator< (const MeinString &zweiterString) const {
   if (strcmp(inhalt, zweiterString.inhalt) < 0)
      return 1;
   else
      return 0;
}
main() {
   MeinString str_a, str_b;

   ...// (Belegen von MeinString str_a mit Inhalt)

   str_b = str_a;                          // Jetzt ok!
   if(str_b < str_a) { /* ... */ };  // ok
}
```

5.5.2 Überladen außerhalb einer Klasse

Hier muss auch der erste Parameter angeben werden.

Formate

- `Rückgabetyp operatorZeichen (Parameter1, Parameter2)`
  `// für: Parameter1 Zeichen Parameter2 Infix-Operator`
- `Rückgabetyp operatorZeichen(Parameter)`
  `// für: Zeichen Parameter Präfix-Operator`
- `Rückgabetyp operatorZeichen (Parameter, int)`
  `// für: Parameter Zeichen Postfix-Operator`
  `// (int) nur zur Unterscheidung`

Beispiel 2: Operatoren, die außerhalb von Klassen deklariert sind

```
ostream& operator<< (ostream &os, const MeinString &meinString) {
   return os << meinString.inhalt;
}
main() {
   MeinString str_a, str_b;

   ... // (Belegen von MeinString str_a und str_b mit Inhalt)

   std::cout << str_a << str_b;
}
```

5.5.3 Funktionsobjekte (Funktoren)

Wenn ein Objekt so wie eine Funktion aufgerufen werden soll, muss der Funktionsaufruf-Operator `()` überladen werden. Das Objekt heißt dann „Funktionsobjekt“ oder „Funktor“. Der Vorteil liegt darin, dass diese „Funktion“ alles das machen kann, was sonst eine Methode macht. Nur ist die Schreibweise sehr viel einfacher; dass es sich dabei um ein Objekt handelt bleibt verborgen. (Der Funktionsaufruf–Operator `()` kann auch nur als nicht statische Member–Funktion überladen werden.)

Der Funktionsaufruf–Operator `()`

Der erste Operand für diesen Operator ist das aktuelle Element der Klasse, für welchen er definiert wird. Er kann kein, ein oder mehrere weitere Operanden oder besser Argumente haben.

Hier ist eine Deklaration eines Funktionsaufruf–Operator mit einem Argument vom `Typ2` und dem Ergebnis von `Typ1`:

```
class A {
   ...
public:
Typ1 operator() (Typ2);
   ...
};
```

und die passende Definition dieser Funktion:

```
Typ1 A::operator() (Typ2 arg){
   // hier steht, was gemacht werden soll
}
```

Der Aufruf (wie eine Funktion):

```
A a;
int b;
...
b = a(7);
...
```

wird vom Compiler umgesetzt zu:

```
a.operator() (7);
```

d.h. vom Objekt `a` wird die Operatorfunktion `operator()` mit dem Argument `7` aufgerufen. Dieser etwas umständliche Aufruf `a.operator()(7)` der Operatorfunktion `operator()` ist syntaktisch in C++ ebenfalls erlaubt!

Der Funktionsaufruf-Operator kann mit unterschiedlicher Signatur überladen werden — Defaultparameter sind hier auch erlaubt:

```
class A {
   ...
public:
      // ein double Argument, Standardparameter 1.0
   int operator() (double = 1.0);
      // zwei Argumente, eins int, eins char
   int operator() (int, char);
      // zwei Argumente, eins int, eins char,
      // aber konstante Member-Funktion
   int operator() (int, char) const;
      // ein Argument, Typ char *
   int operator() (char*);
      // ein Argument, Typ const char*
   int operator() (const char*);
   ...
};
```

Anwendung

```
A a;
const A b;
char s[10];
double x;
...
a(x);       // a.operator() (x),
            // Aufruf von operator() (double = 1.0);
a();        // a.operator() (),
            // Aufruf von operator() (double = 1.0);
a(7,'c');   // a.operator() (7,'c'),
            // Aufruf von operator() (int, char);
b(7,'c');   // b.operator() (7,'c'),
            // Aufruf von operator() (int, char) const;
a(s);       // a.operator() (s),
            // Aufruf von operator() (char*);
a("hallo"); // a.operator() ("hallo"),
            // Aufruf von operator() (const char*);
...
```

Zusammenfassung:

- Operatoren können innerhalb (siehe Beispiel 1) oder außerhalb (siehe Beispiel 2) von Klassen deklariert werden.
- Innerhalb von Klassen deklarierte Operatoren haben als ersten Operanden das aktuelle Objekt. Der zweite Operand ist ggf. als Parameter anzugeben.
- Bei der Deklaration von Operatoren außerhalb von Klassen sind alle Operanden als Parameter anzugeben.
- Operatoren können einen Rückgabewert besitzen.
  - Dieser bezieht sich auf das Ergebnis des gesamten Ausdrucks (siehe Rückgabewert `int` für „<"-Operator im Beispiel 1).
  - Der Rückgabewert **sollten** denselben Typ wie der erste Operand (siehe `ostream&` für „<<"-Operator im Beispiel 2). Damit ist die Verkettung von Operatoranweisungen möglich.
  - Der Rückgabewert **könnte** `void` sein („="-Operator im Beispiel 1 ist ein schlechtes Beispiel).
  - Konvertierungsoperatoren (siehe Abschnitt 5.7) benötigen nicht die Angabe eines Rückgabetyps. Der Typ des Rückgabewertes entspricht dem Operatornamen.
- Weitere Beispiele siehe Abschnitte 5.9.1 und 5.9.2.

5.6 Konvertierung allgemein (casten)

Mit dem ANSI Standard wurde ein neu Cast-Syntax eingeführt. Statt des bisherigen Cast-Operators `(Typ)`, der fast alles in jedes konvertiert auch - konstante in nicht-konstante Typen -, stehen jetzt vier verschiedene Cast-Operatoren mit abgestufter Wirkungsweise zur Verfügung:

```
static_cast<Typ> (ausdruck)
dynamic_cat<Typ> (ausdruck)
const_cast<Typ> (ausdruck)
reinterpret_cast<Typ> (ausdruck)
```

Ergebnis ist jeweils der nach Typ konvertierte Wert des Ausdrucks. Die von den alten Casts abweichende und etwas umständliche Syntax soll es einfacher machen, mit einem Editor in der Programmquelle nach solchen zu suchen.

5.6.1 static_cast

Dieser Cast erlaubt Konversionen zwischen elementaren Datentypen sowie zwischen Klassen, die durch öffentliche, nicht-virtuelle Ableitung auseinander hervorgehen. Die Konstantheit eines Typs lässt sich durch diesen Cast nicht verändern.

Beispiel für eine Konversion von `int` nach `float`:

```
int festpunkt = 14;
float wert = static_cast<float>(festpunkt);
```

Erster Versuch: Klassenbindung bei mehrfacher Vererbung

Es wird ein Programm mit drei Klassen `A, B` und `C`, die jeweils mit `getZustand()` ihren Zustand (`i, j, this`) ausgeben können, verwendet.

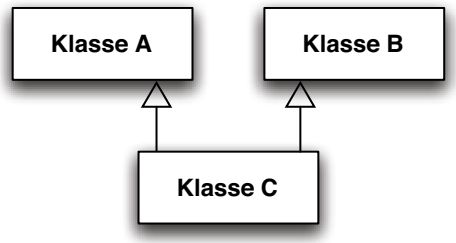
```
class A {
public:
   A (int i=1000, int j=2000) {
      this->i=i;
      this->j=j;
   };
   void getZustand (void);
private:
   int i;
   int j;
};

void A::getZustand (void) {
   std::cout << "Ich bin Klasse A mit Zustand"
   << " (i= "<< i << " j= " << j << " this = " << this << ")"
   << std::endl;
   };

class B {
public:
   B (int i=101, int j=102) {
      this->i=i;
      this->j=j;
   };
   void getZustand (void);
private:
   int i;
   int j;
};
```

```
void B::getZustand (void) {
   std::cout << "Ich bin Klasse B mit Zustand"
   << " (i= "<< i << " j= " << j << " this = " << this << ")"
   << std::endl;
};

class C : public A, public B {
public:
   C (int i=201, int j=202) : A(211, 212), B(221, 222) {
      this->i=i;
      this->j=j;
   };
   void getZustand (void);
private:
   int i;
   int j;
};

void C::getZustand (void) {
   std::cout << "Ich bin Klasse C mit Zustand"
   << " (i= "<< i << " j= " << j << " this = " << this << ")"
   << std::endl;
};
```

In diesem Versuch soll gezeigt werden, wie in der Instanz der Klasse `C` zu den Instanzen der anderen Klassen, die `C` durch Vererbung enthält, navigiert werden kann. Leitfaden dafür ist die Klassen-Bindung (frühe Bindung).

Source-Code

```
class A {};
class B {};
class C : public A, public B {};
```

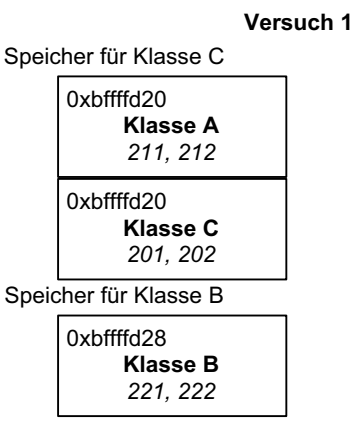


Programm

```
C c;
std::cout << "Adresse C = " << &c <<
std::endl;
A * aZeiger = &c;
aZeiger->getZustand();
B * bZeiger = &c;
bZeiger->getZustand();
C * cZeiger = &c;
cZeiger->getZustand();
```

Programm-Ausgabe

```
Adresse C = 0xbffffd20
Ich bin Klasse A mit Zustand (i= 211 j= 212
this = 0xbffffd20)
Ich bin Klasse B mit Zustand (i= 221 j= 222
this = 0xbffffd28)
Ich bin Klasse C mit Zustand (i= 201 j= 202
this = 0xbffffd20)
```



Wir haben hier eine Instanz der Klasse `C` mit der Adresse `0xbffffd20`. Sie ist quasi das Eingangstor zu allen enthaltenen Instanzen, die `C` durch Vererbung erhalten hat. Deswegen haben alle Zeiger diesen Wert - sie unterscheiden sich aber in ihrem Typ. Obwohl alle dieselbe Adresse haben, rufen sie doch, entsprechend ihres Klassentyps, verschiedene Methoden auf, nämlich `getZustand` in ihrer Klasse (Klassen-Bindung, frühe Bindung). `bZeiger` ist eine Ausnahme, seine Adresse wird durch einen impliziten `upcast` gewandelt (siehe nächster Versuch).

Zweiter Versuch: Impliziter upcast bei mehrfacher Vererbung

Programm

```
C * cZeiger = new C;
A * aZeiger = cZeiger;
B * bZeiger = cZeiger;
```

Mit diesem Programm erreichen wir denselben Effekt wie im ersten Versuch. Es wird ein `cZeiger` erzeugt, der auf eine Instanz der Klasse `C` zeigt und den anderen Zeigern zugewiesen wird.

upcast

- Casten von einer abgeleiteten zu einer Basisklasse (in der Klassenhierarchie von *unten* nach *oben*)

Impliziter upcast

Bei den Zuweisungen findet nun eine implizite Typ-Umwandlung (`cast`, hier `upcast`) statt. Der Typ `C*` von

`cZeiger` wird in den Typ `A*` für `aZeiger` bzw. in den Typ `B*` für `bZeiger` gewandelt.

Dritter Versuch: Expliziter downcast

Programm

```
C * cZeiger = new C;
A * aZeiger = cZeiger;
B * bZeiger = cZeiger;
cZeiger=aZeiger;
```

Compiler-Meldung

```
invalid conversion from `A*' to `C*'
```

downcast

- Casten von einer Klasse zu einer von dieser Klasse abgeleiteten Klasse (in der Klassenhierarchie von *oben* nach *unten*)

Der Compiler kann *implizit* nur einen `upcast` durchführen. Aus diesem Grund müssen wir *explizit* den `downcast` durchführen.

Programm

```
cZeiger=static_cast<C*>(aZeiger);
```

Vierter Versuch: upcast bei mehrfacher Vererbung mit gemeinsamer Wurzel

Wir benutzen das schon in Abschnitt 4.3.1 verwendete Programm, dort hatten wir das Problem, dass die gemeinsame Wurzel (Basisklasse) bedingt durch Mehrfachvererbung doppelt existiert.

```
class A {
public:
   A (int i=1000, int j=2000) {
      this->i=i;
      this->j=j;
   };
   void getZustand (void);
private:
   int i, j;
};

void A::getZustand (void) {
   std::cout << "Ich bin Klasse A mit Zustand"
   << " (i= "<< i << " j= " << j << " this = " << this << ")"
   << std::endl;
};


class B : public A{
public:
   B (int i=101, int j=102) : A(111, 112){
      this->i=i;
   this->j=j;
   };
   void getZustand (void);
private:
   int i, j;
};

void B::getZustand (void) {
   std::cout << "Ich bin Klasse B mit Zustand"
   << " (i= "<< i << " j= " << j << " this = " << this << ")"
   << std::endl;
};

class C : public A {
public:
   C (int i=201, int j=202) : A(211, 212) {
      this->i=i;
      this->j=j;
   };
   void getZustand (void);
```

```
private:
   int i, j;
};

void C::getZustand (void) {
   std::cout << "Ich bin Klasse C mit Zustand"
   << " (i= "<< i << " j= " << j << " this = " << this << ")"
   << std::endl;
};

class D : public B, public C {
public:
   D (int i=301, int j=302) : B(311, 312), C(321, 322) {
      this->i=i;
      this->j=j;
   };
   void getZustand (void);
   operator char * ();
   std::string textZustand (void);
private:
   int i, j;
};

void D::getZustand (void) {
   std::cout << "Ich bin Klasse D mit Zustand"
   << " (i= "<< i << " j= " << j << " this = " << this << ")"
   << std::endl;
};
```

Source-Code

```
class A {};
class B : public A {};
class C : public A {};
class D : public B, public C {};
```

Programm

```
D * dZeiger = new D;
A * aZeiger = dZeiger;
B * bZeiger = dZeiger;
C * cZeiger = dZeiger;
```

Compiler-Meldung

```
error: `A' is an ambiguous base of `D'
bei Zeile: A * aZeiger = dZeiger;
```

Dritter Versuch: Expliziter upcast bei mehrfacher Vererbung mit gemeinsamer Wurzel

Wir müssen dem Compiler einen eindeutigen Weg angeben. Das geht nur von Klasse `B` nach `A`, oder von Klasse `C` nach `A`. Dazu ist ein expliziter upcast von `B*` nach `A*` bzw. von `C*` nach `A*` notwendig.

Programm

```
    D * dZeiger = new D;
// A * aZeiger = dZeiger;
    B * bZeiger = dZeiger;
    C * cZeiger = dZeiger;

    A * aZeiger;
    aZeiger=static_cast<A*>(bZeiger);
    aZeiger=static_cast<A*>(cZeiger);
```

Nun können wir uns wieder die Zustände ausgeben lassen.

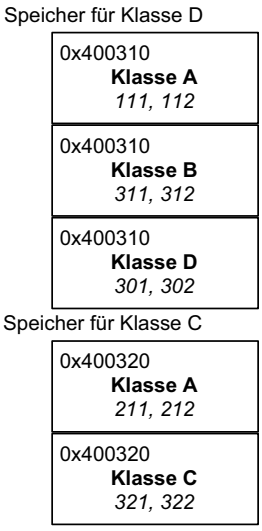
Programm

```
D * dZeiger = new D;
dZeiger->getZustand();
B * bZeiger = dZeiger;
bZeiger->getZustand();
C * cZeiger = dZeiger;
cZeiger->getZustand();

A * aZeiger;
aZeiger=static_cast<A*>(bZeiger);
aZeiger->getZustand();
aZeiger=static_cast<A*>(cZeiger);
aZeiger->getZustand();
```

Programm-Ausgabe

```
Ich bin Klasse D mit Zustand (i= 301 j= 302
this = 0x400310)
Ich bin Klasse B mit Zustand (i= 311 j= 312
this = 0x400310)
Ich bin Klasse C mit Zustand (i= 321 j= 322
this = 0x400320)
Ich bin Klasse A mit Zustand (i= 111 j= 112
this = 0x400310)
Ich bin Klasse A mit Zustand (i= 211 j= 212
this = 0x400320)
```

Versuch 4



Zusammenfassung: Navigieren durch Klassenhierarchien

Mit Hilfe des Castens können wir ausgehend von der Adresse einer Instanz alle Instanzen erreichen, die von anderen Klassen enthalten sind. Mittels der Klassen-Bindung (frühe Bindung) kann der Compiler implizit `upcasts` durchführen. `downcasts` müssen explizit mit dem Cast-Operator vom Programmierer vorgenommen werden.

5.6.2 dynamic_cast

Hiermit werden Zeiger oder Referenzen auf miteinander verwandte polymorphe Klassen - **und nur solche** - ineinander konvertiert. Im Gegensatz zum statischen Cast wird dieser zur Laufzeit ausgeführt. Beim Konvertieren von Zeigern und Referenzen bekommt man eine Rückmeldung darüber,

ob der angegebene Typ dem dynamischen Typ des Objekts entspricht. Bei Zeigern wird der Wert 0 zurückgegeben, wenn die Konversion unzulässig ist, bei Referenzen wird eine Exception (Abschnitt 5.10) ausgelöst. Ein weiterer Unterschied zu normalen Casts besteht darin, dass sich hiermit auch eine Konversion von einer virtuellen Basisklasse in eine abgeleitete durchführen lässt.

Erster Versuch: cast bei polymorphen Klassen

Wir wandeln unser Programm vom letztem Versuch in den virtuellen Zustand (wie wir es schon im Abschnitt 4.3.2 gemacht haben).

Source-Code

Klasse `A` wird zur virtuellen Basisklasse

```
virtual ~A(){};// virtueller Destruktur, bitte nie vergessen!
virtual void getZustand (void);// virtuelle funktion
```

bei Klasse `B`: `virtual` hinzufügen

```
class B : virtual public A {
```

bei Klasse `C`: `virtual` hinzufügen

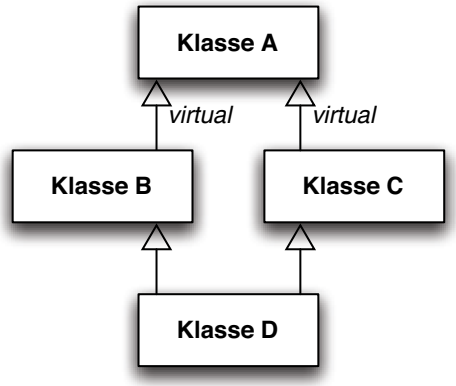
```
class C : virtual public A {
```

Klasse `D`: Konstruktor um Initialisierung für Klasse `A` Erweitern

```
D (int i=301, int j=302) : A(800, 900), B(311, 312), C(321, 322) {
```

Programm

```
D * dZeiger = new D;
dZeiger->getZustand();
B * bZeiger = dZeiger;
bZeiger->getZustand();
C * cZeiger = dZeiger;
cZeiger->getZustand();

A * aZeiger;
aZeiger=static_cast<A*>(bZeiger);
aZeiger->getZustand();
aZeiger=static_cast<A*>(cZeiger);
aZeiger->getZustand();
```



Programm-Ausgabe

```
Ich bin Klasse D mit Zustand (i= 301 j= 302 this = 0x400310)
Ich bin Klasse D mit Zustand (i= 301 j= 302 this = 0x400310)
Ich bin Klasse D mit Zustand (i= 301 j= 302 this = 0x400310)
Ich bin Klasse D mit Zustand (i= 301 j= 302 this = 0x400310)
Ich bin Klasse D mit Zustand (i= 301 j= 302 this = 0x400310)
```

Alle unsere Bemühungen aus den letzten Versuchen sind dahin! Es wird nur die Funktion get von Klasse `D` angesprochen, weil ja allen Zeigern der `dZeiger` zugewiesen wurde. - Das wollten wir in Abschnitt 4.3.2 ja erreichen, dass wirklich das gemacht wird, auf das gezeigt wird.

Eigenartig ist das schon: die Zeiger haben ja nicht alle identische Adressen!

Programm

```
D * dZeiger = new D;
```

```
std::cout << "dZeiger = " << dZeiger << std::endl;
A * aZeiger = dZeiger;
std::cout << "aZeiger = " << aZeiger << std::endl;
B * bZeiger = dZeiger;
std::cout << "bZeiger = " << bZeiger << std::endl;
C * cZeiger = dZeiger;
std::cout << "cZeiger = " << cZeiger << std::endl;
```

Programm-Ausgabe

```
dZeiger = 0x400310
aZeiger = 0x400330
bZeiger = 0x400310
cZeiger = 0x40031c
```

Es muss also in der VTable zu jedem Zeiger auch der Typ gespeichert sein, wohin er zeigt. - Das ist der *dynamische* Typ oder *Objekt-Typ* (im Gegensatz zum letztem Kapitel, wo der *statische* Typ oder *Klassen-Typ* bestimmte, was gemacht werden soll).

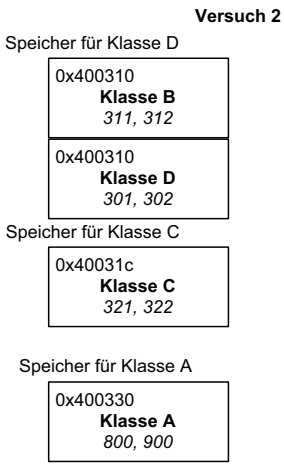
Zweiter Versuch: downcast bei polymorphen Klassen

Programm

```
A * aZeiger = new D;
std::cout << "aZeiger = " << aZeiger << std::endl;
aZeiger->A::getZustand();
B * bZeiger = dynamic_cast<B*>(aZeiger);
std::cout << "bZeiger = " << bZeiger << std::endl;
bZeiger->B::getZustand();
C * cZeiger = dynamic_cast<C*>(aZeiger);
std::cout << "cZeiger = " << cZeiger << std::endl;
cZeiger->C::getZustand();
D * dZeiger = dynamic_cast<D*>(aZeiger);
std::cout << "dZeiger = " << dZeiger << std::endl;
dZeiger->D::getZustand();
```

Programm-Ausgabe

```
aZeiger = 0x400330
Ich bin Klasse A mit Zustand (i= 800 j= 900
this = 0x400330)
bZeiger = 0x400310
Ich bin Klasse B mit Zustand (i= 311 j= 312
this = 0x400310)
cZeiger = 0x40031c
Ich bin Klasse C mit Zustand (i= 321 j= 322
this = 0x40031c)
dZeiger = 0x400310
Ich bin Klasse D mit Zustand (i= 301 j= 302
this = 0x400310)
```



Beim Navigieren durch eine polymorphe Klassenhierarchie wird empfohlen, alle Zeiger von der Basisklasse mit `downcasts` abzuleiten. Denn ein `upcast` wird immer (auch beim `dynamic_cast`) vom Compiler *statisch* durchgeführt!!!

Auffallend ist, dass man immer die Klassenzugehörigkeit der virtuellen Funktion `getZustand` angeben muss. Wird sie weggelassen, wird die Methode in der Klasse `D` aufgerufen.

Programm

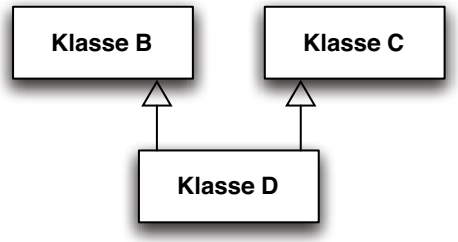
```
dZeiger->A::getZustand();
dZeiger->B::getZustand();
dZeiger->C::getZustand();
dZeiger->D::getZustand();
```

Mit diesem Programm kommt man zu demselben Ergebnis. Also war das Erzeugen der Zeiger mittels `dynamic_cast` nicht zwingend! Es wird sowieso empfohlen möglichst ohne Casts auszukommen.

Dritter Versuch: crosscast bei polymorphen Klassen

Source-Code

```
class B {
   ..
   virtual ~B(){};
}
class C {}
class D : public B, public C
```



Programm

```
   B * bZeiger = new D;
   C * cZeiger = dynamic_cast<C*>(bZeiger);
```

Ein `crosscast` ist nur eingeschränkt möglich. Beide Klassen müssen einen gemeinsamen Erben haben (hier: class `D`) und das umzuwandelnde Objekt muss direkt oder indirekt seinen Typ haben (hier: `bZeiger` ist vom Typ `D`).

5.6.3 Absicherung eines dynamic_cast

Da es sich ja generell um eine späte Bindung handelt, die erst zur Laufzeit ausgewertet wird, hat der Compiler wenig Chancen Fehler im vorhinein festzustellen. Aus diesem Grund kann der `dynamic_cast` im Fehlerfall reagieren. Der Compiler legt für jede polymorphe Klasse ein Objekt der Klasse `type_info` an, die zum RTTI (Run-time type identification) gehört.

Bei Zeigern wird der Wert 0 zurückgegeben, wenn die Umwandlung unzulässig ist.

Erster Versuch: dynamic_cast mit Zeigern

Programm

```
A a;
C * cZeiger = dynamic_cast<C*>(&a);
if (cZeiger == 0) {
   std::cout << "cast schiefgegangen!";
   return 0;
}
std::cout << "cZeiger = " << cZeiger << std::endl;

cZeiger->C::getZustand();
```

Compiler-Meldung

```
dynamic_cast of `A a' to `class C*' can never succeed
```

Programm-Ausgabe

```
cast schiefgegangen!
cast has exited with status 0.
```

Zweiter Versuch: dynamic_cast mit Referenzen

Bei Referenzen wird eine Fehlerklasse `bad_cast` geworfen, die mit einem `catch` aufgefangen werden muss. Es handelt sich dabei um eine Ausnahmebehandlung (Exception), die in Abschnitt 5.10 genau behandelt wird.

Programm

```
A * aZeiger = new D;
{
   try {
      B & refB = dynamic_cast<B&>(*aZeiger);
      refB.B::getZustand();
      A a;
      C & refC = dynamic_cast<C&>(a);
      refC.C::getZustand();
   }
   catch (std::bad_cast) {
      std::cout << "cast schiefgegangen!";
      return 0;
   }
}
```

Compiler-Meldung

```
dynamic_cast of `A a' to `class C&' can never succeed
```

Programm-Ausgabe

```
Ich bin Klasse B mit Zustand (i= 311 j= 312 this = 0x400310)
cast schiefgegangen!
cast has exited with status 0.
```

5.6.4 const_cast

Dieser Cast dient ausschließlich dazu, die Konstantheit eines Typs zu entfernen. Es handelt sich dabei um einen schwerwiegenden und potentiell fehlerträchtigen Vorgang. Die neue Cast-Syntax macht im Programm sehr deutlich, was geschieht, während der alte Cast das quasi nebenbei erledigt, ohne dass es beim Lesen der Quelle ins Auge fällt. Es gibt nunmehr eine klare Aufgabenteilung: nur `const_cast` und niemand sonst ist für die Entfernung der Konstantheit zuständig; `const_cast` tut dies und sonst nichts. Eine gleichzeitige Typ- und Konstantheitsänderung muss in zwei Stufen erfolgen.

In Einzelfällen ist dieser Cast doch ganz nützlich. Beispielsweise liefert die Methode `std::c_str() const` eine konstante Zeichenfolge. Entweder man kopiert diesen C-String in einen anderen, oder man entfernt mit dem Cast das `const`.

```
char alter_C_Text[];
std::string neuer_Text;
```

```
...
strcpy(alter_C_Text, neuer_Text.c_str());
D::operator char * () {
std::string textString:
...
return const_cast<char *>(textString.c_str());
};
```

5.6.5 reinterpret_cast

Dieser Operator ist der undifferenzierteste der neuen Casts. Er konvertiert beliebige Zeigertypen ineinander, unabhängig vom Verwandtschaftsgrad, und konvertiert alle Zeiger in ganzzahligen Typen bzw. umgekehrt. Ableitungshierarchien werden nicht berücksichtigt, d.h. der Zeigerwert ändert sich bei der Konversion *nicht*. Abgesehen von dieser Eigenschaft und davon, dass er die Konstantheit nicht entfernt, entspricht er dem alten Cast `(X*)a`.

`reinterpret_cast` setzt die Typprüfung weitgehend außer Kraft und ist daher potentiell gefährlich. Er sollte nicht leichtfertig, sondern nur für spezielle Dinge genutzt werden, die sich anders nicht bewerkstelligen lassen, und man sollte im Auge behalten, dass dergleichen meist nicht portabel ist. Alledings braucht man den Cast auch für den Aufruf von C-Funktionen

```
void writeFloat(int fd, float x) {
write(fd, reinterpret_cast<const char*>(&x), sizeof x);
}
```

5.7 Konvertierung zwischen Klassen

- Implizite Typkonvertierungen werden in folgenden Situationen vom Compiler automatisch durchgeführt, sofern keine explizite Typkonvertierung („Cast“) angegeben wird:
  - Bei der Zuweisung von Werten.
  - Bei arithmetischen Operationen.
  - Bei der Parameterübergabe an Funktionen.
  - Bei der Rückgabe von Werten durch Funktionen.
- Die dabei verwendeten Konvertierungsfunktionen müssen für Konvertierungen zwischen Klassen und zwischen Klassen und Standardtypen vom Benutzer definiert werden.
- Der Compiler kann Standardkonvertierungen und benutzerdefinierte Konvertierungen gemeinsam ausführen. Beispielsweise kann er die Konvertierung von **`long`** zu einer benutzerdefinierten Klasse `klasse` durchführen, falls eine Konvertierung von **`int`** nach `klasse` definiert wurde (**`long`** -> **`int`** -> `klasse`).

Konvertierung durch Konstruktoren

Konstruktoren, für die nur ein Parameter angegeben werden muss, werden als Konvertierungsfunktion vom Typ des Parameters zum Typ der Klasse verwendet.

Beispiel:

```
class Komplex {
public:
```

```
   Komplex(float realteil, float imaginaerteil = 0.0);
         // Konvertierungsfunktion von float nach complex!
   ...
};

Komplex c(2.0);    // hier wird der Konstruktor Komplex c(2.0, 0.0)
                   // aufgerufen
c = 3.0;          // gleichbedeutend mit c = Komplex(3.0) und
                   // c = Komplex(3.0, 0.0)
```

Hier wird der Konstruktor für ein temporäres Objekt aufgerufen, und dieses Objekt dann zugewiesen.

```
c = 3;
```

Hier wird eine Standardkonvertierung von **`int`** nach **`float`** durchgeführt und dann die o.g. Zuweisung mit Konvertierung nach `Komplex` ausgeführt.

Konvertierungsoperatoren

Konvertierungsoperatoren für die Umwandlung von einer Klasse in eine andere Klasse oder einen Standardtypen können explizit definiert werden.

- Ziel: Umwandlung eines Objekts einer Klasse in einen anderen Typ (Konvertierungskonstruktoren wandeln umgekehrt ein Objekt eines ändern Typs in ein Objekt einer Klasse um)

```
Syntax: operator Typ ()
```

- Konvertierungsoperatoren
  - sind immer Methoden,
  - haben keinen Rückgabetyp
  - liefern trotz fehlenden Rückgabetyps Wert zurück.
  - haben keine Parameterliste.

Beispiel:

```
class MeineKlasse {
public:
   ...
   operator float() const;
   operator AndereKlasse() const;
   int meinWert;
};
MeineKlasse::operator float() const {
   return (float)meinWert;
}

MeineKlasse::operator AndereKlasse() const {
   ...
}
```

- Anwendung

```
MeineKlasse m;
AndereKlasse a;
float f;
```

```
// 4 gleichwertige Aufrufmöglichkeiten:
f = m.operator float();
f = float(m);
f = (float) m;
f = m;

// auch der folgende Aufruf ist nun moeglich:
a = m;
```

Sind z.B. bei einer Zuweisung mehrere alternative Wege der Typkonvertierung möglich, meldet der Compiler einen Fehler, z.B. falls in `MeineKlasse` und in `AndereKlasse` jeweils eine Konvertierung von `MeineKlasse` nach `AndereKlasse` definiert wird.

5.8 Inline-Funktionen

- Eine mit dem Schlüsselwort **`inline`** deklarierte Funktion wird vom Compiler als Kopie an der Stelle des Aufrufs ins Programm eingefügt. Für jeden Aufruf der Funktion besteht also eine Kopie des Programmcodes der Funktion.
- So kann der Overhead von Funktionsaufrufen (z.B. Laden von Parametern auf den Stack) zur Laufzeit vermieden werden. Andererseits nimmt jedoch der Umfang des kompilierten Programmcodes stark zu, wenn die Funktion groß ist oder sehr häufig aufgerufen wird.
- Im Gegensatz zu Makros (wie sie auch in C verfügbar sind) findet bei **`inline`**-Funktionen eine Typüberprüfung statt. Soll die Funktion mit Parametern verschiedener Typen funktionieren (wie Makros), so muss sie überladen werden (Vergleiche Abschnitt 5.1).
- Darüberhinaus werden die teilweise lästigen Nebenwirkungen von Makros vermieden.

Beispiel 1:

```
// Makros und inline-Funktionen
#define MAX(A, B) ((A) > (B) ? (A) : (B))

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

void main()
{
  int i, x=23, y=45;
  i = MAX(x++, y++); // lästige Makro-Nebenwirkung
                     // (Überlegen Sie selbst, welche!)
  int v=23, w=45;
  i = max(v++, w++); // funktioniert wie gewünscht
};
```

Die **`inline`**-Deklaration kann auch implizit erfolgen, indem die Funktionsdefinition in der Klassendefinition erfolgt:

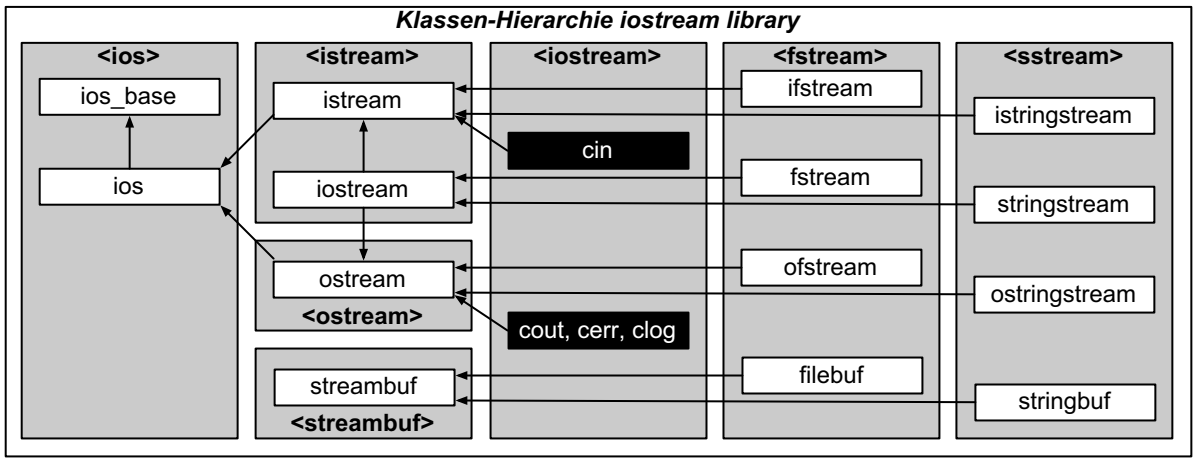
Beispiel 2:

```
class MeineKlasse {
public:
```

```
 void print() {std::cout << i << ''; } // Implizit inline deklariert
private:
 int i;
};
```

5.9 Ein-/Ausgabe in C++ (Streams)

- Die Sprache C++ selbst stellt keine Ein-/Ausgabemechanismen bereit. Diese werden einfach und elegant innerhalb der Sprache umgesetzt.
- Die Standard-Ein-/Ausgabebibliothek auf Basis von Streams (Strömen) in `<iostream>` bietet eine typsichere, flexible und effiziente Methode für Standardtypen und Erweiterungsmöglichkeiten für benutzerdefinierte Typen.
- Eine minimale Einführung in die Ein-/Ausgabemechanismen von C++ erfolgte bereits in den Beispielprogrammen in Abschnitt 3.5.
- Ältere Implementierungen von C++ benutzen die Bibliothek `<iostream.h>`, die inzwischen von der mächtigeren Bibliothek `<iostream>` abgelöst worden ist, jedoch auf vielen Systemen noch aus Kompatibilitätsgründen existiert.



5.9.1 Ausgabe

Zur Ausgabe wird einem Ausgabe-Stream (z.B. `cout`) mit dem <<-Operator („put to“-Operator) ein auszugebender Wert übergeben.

Der <<-Operator muss für den Typ des auszugebenden Wertes definiert sein.

Ausgabe-Streams

Vordefinierte Ausgabe-Streams sind zunächst der Standardausgabe-Stream `cout`, der Standardfehlerausgabe-Stream (ungepuffert) `cerr` und der Standardfehlerausgabe-Stream (gepuffert, Protokoll) `clog`.

Ausgabe von eingebauten Standardtypen

In der Klasse `ostream` ist der <<-Operator für die eingebauten Standardtypen definiert.

```
class ostream : virtual public ios {
```

```
   ...
public:
   ostream& operator<< (const char*);  // Strings
   ostream& operator<< (char);
   ostream& operator<< (short);
   ostream& operator<< (int);
   ostream& operator<< (long);
   ostream& operator<< (double);
   ostream& operator<< (const void*);  // Zeiger
   // weitere Definitionen, z.B. für unsigned-Typen
   ...
};
```

Die Definition mit Rückgabewert vom Typ `ostream&` ist für die Verkettung von Ausgabekommandos gemäß dem Beispiel

```
cerr << "x = " << x;
```

erforderlich.

Ausgabe benutzerdefinierter Typen

Für benutzerdefinierte Typen kann der <<-Operator nach folgendem Schema erweitert werden:

```
ostream& operator<< (ostream &s, Typ Parameter)
{
   s << <was immer Sie hier ausgeben möchten>;
   return s;
};
```

Eine Modifikation der Header-Datei `<iostream>` ist hierzu nicht erforderlich.

Beispiel:

```
class komplex {
   double re, im;
public:
   friend ostream& operator<< (ostream &s, komplex z);
   ...
};

ostream& operator<< (ostream &s, komplex z) {
   return s << ´(´ << z.re << ´,´ << z.im << ´)´;
         // Ausgabe im Format (1.0,2.3)
};
```

Benutzung:

```
void main() {
   ... (Definition eines Objektes x vom Typ komplex)
   std::cout << x << "\n";
};
```

5.9.2 Eingabe

Die Eingabe erfolgt analog zur Ausgabe mit dem >>-Operator („get from“-Operator). Dieser ist definiert in der Klasse `istream`.

Eingabe von Standardtypen

Der >>-Operator ist ebenfalls für Standardtypen vordefiniert.

Beispiel:

```
void main() {
   int i;
   std::cout << "Bitte geben Sie eine Zahl ein: ";
   std::cin >> i;
   std::cout << "Die eingegebene Zahl lautet " << i << "\n";
}
```

Statusabfrage von Streams

Um Fehler insbesondere bei der Eingabe komplexerer Datentypen zu vermeiden, kann der Status von Streams abgefragt werden.

- Dazu sind in der Klasse `ios` (die Basisklasse von `ostream` und `istream`) verschiedene Funktionen definiert.

```
bool eof() const;    // end-of-file entdeckt
bool fail() const;   // Fehler in der Verarbeitung
bool bad() const;    // Stream buffer beschaedigt
bool good() const;   // kein Fehler, Abwesenheit der anderen Bits
```

- Alternativ steht die Methode `rdstate()` zur Verfügung. Deren Rückgabewerte sind in `ios` definiert:

```
std::ios::goodbit
std::ios::eofbit
std::ios::failbit
std::ios::badbit
```

Der Status eines Streams kann z.B. folgendermaßen getestet werden:

```
if(std::cin.rdstate() & std::ios::goodbit)
   // letzte Operation war erfolgreich
if(std::cin.rdstate() & std::ios::eofbit)
   // Ende des Puffers
if(std::cin.rdstate() & std::ios::failbit:
   // irgendein Formatfehler, wahrscheinlich nichts Ernstes
if(std::cin.rdstate() & std::ios::badbit:
   // cin hat moeglicherweise ein Zeichen verloren
```

Eingabe benutzerdefinierter Typen

Die Definition des Eingabeoperators für eigene Typen erfolgt so:

```
istream& operator>>(istream& s, Typ &Parameter) {
   s >> <lokale Variable>;
   // Kopieren von der lokalen Variablen in den Parameter.
   // Behandlung von Fehlerfällen und Ausnahmen
   return s;
}
```

Beispiel:

Eingabeoperator für komplexe Zahlen für drei mögliche Eingabeformate:

```
(re,im)   also z.B.: (1.2,3.7)
(re)      also z.B.: (5.0)
re        also z.B.: 2.8
```

Code:

```
class Komplex {
public
   Komplex(double realpart, double impart) {re=realpart; im=impart;}
private:
   double re, im;
};

istream& operator>>(istream& s, komplex& z) {
   double realpart = 0, impart = 0;
   char c = 0;

   s >> c;
   if (c == ´(´) {
      s >> realpart >> c;
      if (c == ´,´) s >> impart >> c;
      if (c != ´)´) s.clear(std::ios::badbit);
                        //löschen und nur badbit setzen
   }
   else {
      s.putback(c);
      s >> realpart;
   }
   if (s) {                // Status abfragen!
      z = komplex(realpart, impart);
   }
   return s;
}
```

Mit diesem Codestück werden bereits zahlreiche potentielle Fehleingaben ausgeschlossen, eine Verfeinerung ist nach Bedarf möglich.

5.9.3 Formatierung

Funktionen zur Manipulation von Streams

Die Klasse `ios` (Oberklasse von `ostream` und `istream`) stellt zahlreiche Funktionen für die Manipulation der Ein- und Ausgabe zur Verfügung, auf die hier nicht detailliert eingegangen werden kann.

Angesprochen seien die folgenden:

- `tie()` z.B. zum Aneinanderbinden von `cin` und `cout` (zeitliche Abstimmung)
- `flush()` zum Entleeren des Puffers des entsprechenden Streams

- `width()` zum Festlegen der Feldlänge der folgenden Zahlen- oder String-Ausgabe
- `fill()` zum Festlegen des verwendeten Füllzeichens
- `flags()` zum Abfragen des Format-Status von `ios` (z.B. linksbündig, rechtsbündig, dezimal, oktal, hexadezimal, ...)
- `setf()` zum Setzen des Format-Status von `ios`

Manipulator-Notation

Solche Operationen können mittels der sog. Manipulator-Notation z.T. auch direkt in den Stream eingefügt werden. Dazu ist das Einfügen von `<iomanip>` erforderlich.

So werden beispielsweise folgende Schreibweisen möglich:

- Entleeren des Puffers von `cout` mit `flush()`:

```
std::cout << x << flush << y << flush;
```

- Konsumieren von „whitespace"-Zeichen vor dem Leseversuch von `x`:

```
std::cin >> ws >> x;
```

(Hinweis: Das „Fressen" von whitespace-Zeichen wird bei >> ohnehin vorgenommen, diese `ws`-Anwendung ist hier also überflüssig!)

- Ausgabe von „\n" und `flush()`:

```
std::cout << "Hallo" << endl;
```

- Ausgabe von Zahlen in Hexadezimal- oder Oktalformat:

```
std::cout << hex << 1234 << ' ' << oct << 1234 << endl;
```

Positionierung innerhalb von Streams

Im Hinblick auf die Verwendung von Streams auch für Dateizugriffe sind für `istream` und `ostream` verschiedene Funktionen zur Manipulation vorhanden.

`ostream`:

`flush()` entleert den Puffer in den Ausgabestrom
`seekp()` positioniert den `ostream` für Schreiboperationen

`istream`:

`peek()` untersucht das nächste Zeichen, ohne es aus dem Strom zu nehmen
`putback()` schreibt ein „ungewolltes" Zeichen in den Stream zurück
`seekg()` positioniert den `istream` für Leseoperationen

5.9.4 Dateien und Streams

Zur Ein-/Ausgabe von/in Dateien ist `<fstream>` erforderlich.

Öffnen von Dateien

- Eine Datei wird zur Ausgabe geöffnet, indem ein Objekt der Klasse `ofstream` (output file stream) erzeugt wird, als Parameter für den Konstruktor ist der Dateiname anzugeben.
- Eingabedateien werden durch Erzeugen eines Objektes vom Typ `ifstream` (input file stream) geöffnet.

- Durch die Angabe eines zweiten Arguments beim Öffnen der Datei sind alternative Modi möglich.
  Mögliche Werte sind:

`std::ios::in`	Öffnen für Lesezugriff
`std::ios::out`	Öffnen für Schreibzugriff
`std::ios::ate`	Öffnen und Dateiende suchen
`std::ios::app`	Anfügen am Dateiende
`std::ios::trunc`	Datei auf Länge 0 kürzen
`std::ios::nocreate`	nur öffnen, wenn Datei existiert
`std::ios::noreplace`	nur öffnen, wenn Datei nicht existiert

- Die gelisteten Modi können OR-verknüpft werden (z.B. Lese- und Schreibzugriff)

Beispiele:

```
std::ofstream Ausgabedatei("test1.txt");
std::ifstream Eingabedatei("test2.txt");
std::ofstream meineDatei("test3.txt",
                  std::ios::out | std::ios::nocreate);
std::fstream nocheineDatei("test4.txt",
                  std::ios::in | std::ios::out);
```

Datei Ein-/Ausgabe und Statusabfrage

- Auf Objekte der Klasse `ofstream` können alle `ostream`-Operationen angewandt werden, auf `ifstream`-Objekte alls `istream`-Operationen, d.h. Ein- und Ausgabe erfolgen vor allem mit den Operatoren << und >>.
- Auf ein Objekt der Klasse `fstream` können `istream`- und `ostream`-Operationen angewandt werden, da sie von iostream abgeleitet wurde, welche von `istream` *und* `ostream` abgeleitet ist (Mehrfachvererbung!). Ein `iostream` besitzt unterschiedliche Positionen für das Schreiben und Lesen, die mit seekp und seekg manipuliert werden können.
- Gleich nach dem Öffnen der Datei und auch zu späteren Zeitpunkten sind Statusabfragen sinnvoll.

Beispiele:

```
if (meineDatei.bad()) {
   ...
}
if (EingabeDatei.eof()) {
   ...
}
```

Datei schließen

Schließen einer Datei kann explizit mit `close()` erfolgen, mit Aufruf des Destruktors erfolgt das Schließen der Datei jedoch ohnehin automatisch.

Beispiel:

```
meineDatei.close();
```

5.9.5 C++Strings und Streams

C++Strings und ihre Methoden zur Manipulation sind in `<string>` definiert. Sie dürfen nicht mit mit den C-Strings aus `<string.h>` verwechselt werden.

- Der Weg in die neue Welt ist einfach:

```
char alter_C_Text[] = "ganz alter Text aus der C Welt";
string neuer_Text = alter_C_Text;
```

- Zurück in die alte Welt geht es nur über die Methode `string::c_str() const`:

```
neuer_Text = "ein neuer c Text";
strcpy(alter_C_Text, neuer_Text.c_str());
```

- Strings können genauso wie Dateien als Streams angesprochen werden. Dazu werden die in `<sstream>` deklarierten Klassen `istringstream` (Lesen), `ostringstream` (Schreiben) und `stringstream` (Lesen und Schreiben) verwendet.

```
int wert1 = 14, wert2 = 0, pos;
std::string text1, text2, text3;
std::stringstream   sstr;
```

- `stringstream` kann analog zu `cin` bzw. `cout` benutzt werden:

```
sstr << wert1;
sstr << " mV" << std::ends;
```

- Aus dem `stringstream` kann mittels der Methode `str()` die enthaltene Zeichenkette gewonnen werden:

```
std::cout << "1.  " <<  "Der Wert ist " << sstr.str() << std::endl;
```

- Die Klasse `string` kann sehr einfach benutzt werden. Folgende Operatoren sind überladen: +, +=, ==, !=, <, <=, >, >=, <<, >>. So können Strings z.B. einfach aneinandergehängt werden:

```
text1 = "Der Wert ist ";
text1 += sstr.str();
std::cout << "2.  " << text1 << std::endl;
```

- Strings werden mit `assign()` kopiert:

```
text2.assign(text1);
std::cout << "3.  " << text2 << std::endl;
```

- Pattern matching:

```
pos = text2.find("mV");
std::cout << "4.  mV steht an Position " << pos << std::endl;
```

- Strings werden mittels `append()` aneinandergehängt

```
text3.assign("Der Wert ist ");
```

```
text3.append(sstr.str());
std::cout << "5.   " << text3 << std::endl;
```

- Ein Stream kann auch wieder ausgelesen werden, aber ACHTUNG: whitespaces werden überlesen!

```
sstr >> wert2 >> text3;
std::cout << "6.   " << wert2 << text3 << std::endl;
```

- Auch Manipulatoren (z.B. `hex` und `endl`) können benutzt werden

```
wert2 = 50;
sstr.clear();     //eof Bit ist gesetzt, muss gelöscht werden!
sstr << " Dezimal: " << wert2 << "  hexadezimal " << std::hex <<
wert2 << std::ends;
std::cout << "7.   " << sstr.str() << std::endl;
```

- Ein interner „Dateizeiger“ kann positioniert werden:

```
sstr.clear();
sstr.seekp(3);
sstr << "sonstwas   " << std::ends;
std::cout << "8.   " << sstr.str() << std::endl;
```

- Das ist die Ausgabe obiger Beispiele:

```
1.   Der Wert ist 14 mV
2.   Der Wert ist 14 mV
3.   Der Wert ist 14 mV
4.   mV steht an Position 16
5.   Der Wert ist 14 mV
6.   14mV
7.   14 mV Dezimal: 50 hexadezimal 32
8.   14 sonstwas   : 50 hexadezimal 32
```

5.9.6 Zeichen in Strings

Um einzelne Zeichen in C++Strings verarbeiten zu können, ist ein sogenannter Iterator vom Typ `string::iterator` in der Klasse `string` definiert. Er hat ähnliche Eigenschaften wie ein Zeiger:

- mittels des Operators `*` kann man über einen Iterator auf Zeichen eines C++Strings zugreifen
- mittels des Operators ++ kann man den Iterator erhöhen, so dass er auf das nächste Zeichen des C++Strings „zeigt“
- Iteratoren kann man mittels ==, `!=`, <, ... vergleichen

Wie ein Zeiger muss auch ein Iterator „initialisiert“ werden, damit er nicht „irgendwohin“ in die Gegend „zeigt“, sondern zunächst mal auf das erste Element eines Strings. Hierzu gibt es die String-Member-Funktionen:

`string::iterator string::begin()`
liefert „Iteratorposition“ des ersten Zeichens im String

`string::iterator string::end()`
liefert „Iteratorposition“ eins hinter dem letzten Zeichen des Strings

Mittels dieses Iterator-Typs und dieser Funktionen können dann auch sehr einfach Schleifen über alle Zeichen des C++Strings formuliert werden (man beachte: das Element an der Position `begin()` gehört zum String, das mit der Position `end()` gerade nicht mehr - insbesondere ist der `string s` leer, wenn die durch `s.begin()` und `s.end()` gelieferten Iteratorpositionen gleich sind):

```
string s;
...
for (string::iterator iter = s.begin(); iter < s.end(), ++iter) {
      // *iter ist das aktuelle Zeichen des Strings
   ... // mach was mit *iter!
}
```

- Genaueres zu Iteratoren ist im Kapitel 5.11 „STL“ zu finden.

5.9.7 Konvertieren mit Strings

Stringstreams eignen sich sehr gut, einen Typ in einen anderen umzuwandeln.

```
std::stringstream zwischen;
zwischen << eingabe;
zwischen >> ausgabe;
```

Alle Typen für die die Operatoren << und >> definiert sind, können so ineinander umgewandelt werden. Es bietet sich dafür an, eine Schablone (Template) in einem Headerfile zu erstellen.

```
#ifndef Schablone_h
#define Schablone_h
#include <sstream>
template <class Eingabe, class Ausgabe>
void convert(const Eingabe & eingabe, Ausgabe & ausgabe) {
   std::stringstream zwischen;
   zwischen << std::fixed << eingabe;
            //fixed: 2.0e+01 würde sonst nicht gewandelt werden
   zwischen >> ausgabe;
}
#endif
```

Mit dieser Schablone können wir nun alles wandeln, auch beispielsweise unsere komplexe Zahl `Komplex`, weil für diese Klasse beide Operatoren << und >> definiert worden sind!

```
#include "schablone.h"
Komplex z;
convert("(8,3)",z);
```

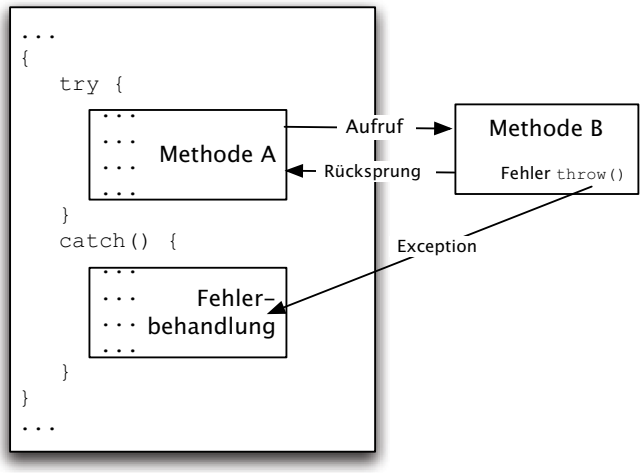
Mit Strings können wir den >> Operator für unsere Klasse Komplex sehr viel eleganter codieren. Wir lesen mit der Funktion `getline()` die ganze Zeile ein und suchen uns darin die einzelnen Strings, abgetrennt durch die Klammern und das Komma, zusammen.

```
std::istream & operator >> (std::istream & s, Komplex & z) {
   double realpart = 0, impart = 0;
   std::string zeile;
```

```
   std::stringstream part_string;
   std::string::size_type pos_anf, pos_end, pos_komma;

   getline(s, zeile);// ganze Zeile einlesen

   pos_anf = zeile.find('(');
   pos_komma = zeile.find(',');
   pos_end = zeile.find(')');

   if ((pos_anf != std::string::npos) && (pos_end !=
std::string::npos)){
      // linke und rechte Klammer gefunden
      if (pos_komma != std::string::npos) {
         // Komma gefunden
         part_string.str(zeile.substr(pos_anf+1, pos_komma-1));
         if (!(part_string >> realpart)) s.clear(std::ios::badbit);

         part_string.clear();
         part_string.str(zeile.substr(pos_komma+1, pos_end-1));
         if (!(part_string >> impart)) s.clear(std::ios::badbit);
      }
      else {
         // kein Komma zwischen Klammern, nur realpart
         part_string.str(zeile.substr(pos_anf+1, pos_end-1));
         if (!(part_string >> realpart)) s.clear(std::ios::badbit);
      }
   }
   else {
      if ((pos_anf == std::string::npos)
            && (pos_end == std::string::npos)
            && (pos_komma == std::string::npos)) {
         // keine Klammern, kein Komma
         part_string.str(zeile);
         if (!(part_string >> realpart)) s.clear(std::ios::badbit);
      }
      else {
         s.clear(std::ios::badbit);
            // eine einzige Klammer oder ein Komma
      }
   }
   if (!s)std::cout << "Eingabefehler" << std::endl;
   z = ComplexNumber(realpart, impart);
   return s;
}
```

5.10 Ausnahmebehandlung

C++ hält einen umfangreichen Mechanismus zur Ausnahmebehandlung (Exception Handling) bereit. Das Grundprinzip ist dabei die Trennung von „normalem Code“ und „Fehlerbehandlung“.



Hierbei startet eine Funktion mittels **`throw`** eine Ausnahme, sobald sie einen nicht selbst zu bereinigenden Fehler entdeckt (z.B. Verletzung der zulässigen Index-Werte eines Arrays). Mittels **`catch`** fängt eine Funktion die „geworfene“ Fehlerklasse auf und behandelt die Ausnahme.

Das Besondere von throw ist, dass es einen sogenannten Prozess `stack unwinding` startet. Da alle Blöcke und Funktionen, die im `try`-Block aktiv sind, auf einem Stack liegen, werden sie so verlassen als wenn jeweils ein Return-Befehl ausgeführt wird. Sind lokale Objekte vorhanden, werden ihre Destruktoren aufgerufen. Bevor also der `catch`-Block betreten wird, ist alles ordentlich aufgeräumt worden.

Beispiel Rationale Zahlen

```
class Rational : public Paar {
   public:
      Rational(int z, int n) : wert1(z), wert2(n) {
         if (wert2 == 0) throw NennerIstNull();
         }
      class NennerIstNull {}; // Fehlerklasse
      ...
};
```

- Dies ist ein besonderer Fall! Der Konstruktor hat keinen Rückgabewert, so dass man einen Fehler feststellen könnte. Mit `throw` ist das nun möglich!

```
int z, n;
...// beliebige Anweisungen
{
   try {
      std::cin >> z >> n;
      Rational r(z, n);
      std::cout << "Eingabe ergab Bruch: " << r;
   }
   catch(Rational::NennerIstNull){
      ...
      std::cerr << "Nenner darf nicht null sein!" << std::endl;
```

```
    }
    catch(...) {// echte ... bedeutet "nicht näher spezifiziert"
       std::cout << "ein nicht erkannter Fehler!" << std::endl;
    }
}
std::out << "Hello World" << std::endl;
      // nach dem try-catch-Block geht es ganz normal weiter
```

- `throw` erzeugt Fehlerobjekte, die von `catch` aufgefangen und weiter bearbeitet werden. Es sind mehrere `catch`-Blöcke zu einem `try`-Block möglich, die so auf verschiedene Fehler unterschiedlich reagieren.

5.10.1 Exception-Hierarchien

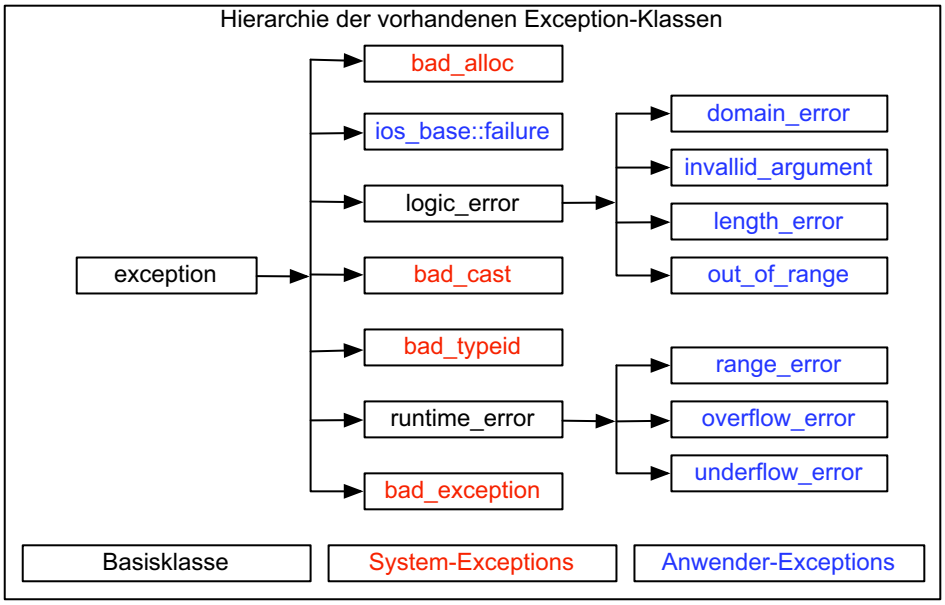
Exceptions können wie alle anderen Klassen mittels Vererbung Hierarchien aufbauen. So kann man eine Menge von Fehlerklassen gemeinsam behandeln:

```
class RationalException {};
class NennerIstNull : public RationalException {};
class LeseFehler : public RationalException {};
...
catch (RationalException) {...}; // alle Fehler
```

Damit können gemeinsame Werte (Begründungen der Fehlersituation usw.) einfach deklariert werden.

5.10.2 Standard-Exceptions (System-Exceptions)

Zur Behandlung von Ausnahmezuständen stellt die Sprache C++ bestimmte Standard Exception-Klassen zur Verfügung. Alle Exception-Klassen sind hierbei von der Basisklasse `exception` abgeleitet und liegen im Namensraum `std`.



Die Exceptions lassen sich generell in drei Bereiche unterteilen:

- Exceptions zur Unterstützung der Sprache C++ (System-Exceptions)

- Exceptions die in der Standard-Bibliothek verwendet werden (`logic_error`, `ios_base::failure`)
- Exceptions die aufgrund von Berechnungen auftreten (`runtime_error`)

5.10.3 ios_base::failure Exception

Auch Streams können beim Fehlschlagen von Operationen Exceptions auslösen. Diese Erweiterung der Streams wurde erst zu einem späteren Zeitpunkt in den C++Standard aufgenommen, als die Streams schon recht häufig eingesetzt wurden. Um mit bestehenden Programmen kompatibel bleiben zu können, lösen Streams standardmäßig zunächst keine Exceptions aus. Die Auslösung von Exception durch Streams muss explizit durch Aufruf der Stream-Memberfunktion exceptions() freigegeben werden. Als Parameter erhält sie eines oder mehrere der Statusflags:

```
std::ios::failbit// Fehler in der Verarabeitung
std::ios::eofbit // end-of-file entdeckt
std::ios::badbit // Streambuffer beschädigt
```

Bei der Verarbeitung von Dateien ist man generell mit dem unlösbaren Problem konfrontiert, dass die Datenstruktur nicht mit der Programmstruktur in Einklang zu bringen ist (Michael A.Jackson: Structure Clash). In der Vorlesung Programmentwurf wird deswegen empfohlen, den aktuellen Block in der Verarbeitung zu verlassen, wenn ein Dateiende (`eof`) erkannt worden ist und in einen neuen Block einzutreten. Diese Blöcke können dann wohl strukturiert werden.

Das folgende Beispiel zeigt exemplarisch, wie sich die einzelnen Phasen der Verarbeitung voneinander mit den `try-catch`-Blöcken trennen lassen.

Beispiel Datei verarbeiten

```
#include <iostream>
#include <sstream>

int main (int argc, char * const argv[]) {

   std::string reihe[20];
   std::istringstream eingabeDatei
      ("erstes Wort \nzweites Wort \ndrittes Wort \n");
   std::stringstream datenPuffer;

   {  // Dialog mit dem Benutzer eine Datei zu öffnen
      try {
         eingabeDatei.exceptions(std::ios::failbit);
         // hier müsste die Datei geöffnet werden
         // die Datei wurde mit einem istring simuliert
      }
      catch(const std::ios_base::failure & ex) {
         std::cout << ex.what() <<
         "; das Öffnen der Datei ist schief gegangen! "
         <<  std::endl;
         exit(1);
      }
   }
   {  // Daten einlesen
      try {
```

```
      eingabeDatei.exceptions(std::ios::failbit
         | std::ios::eofbit);
      int i = 0;
      while (true) {
         getline(eingabeDatei, reihe[i]);
         std::cout << "Reihe " << i << ": " <<reihe[i++]
            <<  std::endl;
      }
   }
   catch(const std::ios_base::failure & ex) {
      std::cout << ex.what() << "; Einlesen verlassen"
         << std::endl;
      if (eingabeDatei.rdstate() & std::ios::failbit) {
         std::cout << " etwas ist schief gegangen"
            << std::endl;
         exit(1);
      }
      std::cout << "Das Einlesen der Daten ist beendet!\n"
         << std::endl;
   }
}
{  // Daten verarbeiten
   try {
      datenPuffer.exceptions(std::ios::failbit
         | std::ios::eofbit);

      for (int i = 0; reihe[i] != ""; i++) {
         datenPuffer << reihe[i];
         std::cout << "datenPuffer " << i << ": "
            << datenPuffer.str() <<  std::endl;
      }
      std::cout
         << "Das Verarbeiten der Daten ist beendet!\n"
         << std::endl;
   }
   catch(const std::ios_base::failure & ex) {
      std::cout << ex.what()
         << " etwas ist schief gegangen" << std::endl;
      exit(1);
   }
}
{  // Daten ausgeben
   try {
      datenPuffer.exceptions(std::ios::failbit
         | std::ios::eofbit);
      int i = 0;
      while (true) {
         datenPuffer >> reihe[i];
```

```
            std::cout << "Reihe " << i << ": " <<reihe[i++]
               <<  std::endl;
         }

      }
      catch(const std::ios_base::failure & ex) {
         if (datenPuffer.rdstate() & std::ios::failbit) {
            std::cout << " etwas ist schief gegangen"
               << std::endl;
            exit(1);
         }
         std::cout << ex.what()
            << "; Der Daten-Puffer ist leer!\n"
            << std::endl;
      }
   }
     std::cout << "Hello, World!\n";
     return 0;
}
```

Ausgabe von diesem Beispiel:

```
Reihe 0: erstes Wort
Reihe 1: zweites Wort
Reihe 2: drittes Wort
basic_ios::clear(iostate) caused exception; Einlesen verlassen
Das Einlesen der Daten ist beendet!

datenPuffer 0: erstes Wort
datenPuffer 1: erstes Wort zweites Wort
datenPuffer 2: erstes Wort zweites Wort drittes Wort
Das Verarbeiten der Daten ist beendet!

Reihe 0: erstes
Reihe 1: Wort
Reihe 2: zweites
Reihe 3: Wort
Reihe 4: drittes
Reihe 5: Wort
basic_ios::clear(iostate) caused exception; Der Daten-Puffer ist
leer!

Hello, World!

Datei has exited with status 0.
```

Beispiel Wandeln

Wir wollen uns nochmal mit unserer Schablone zum Wandeln aus Kapitel 5.9.7 befassen uns die gegen Fehler absichern. Dazu müssen lediglich die Exceptions der Streams scharf gemacht werden:

```
#ifndef Schablone_h
#define Schablone_h
#include <sstream>
template <class Eingabe, class Ausgabe>
void convert(const Eingabe & eingabe, Ausgabe & ausgabe) {
   std::stringstream zwischen;
   zwischen.exceptions(std::ios::failbit);
   zwischen << std::fixed << eingabe;
            //fixed: 2.0e+01 würde sonst nicht gewandelt werden
   zwischen >> ausgabe;
}
#endif
```

- Im Hauptprogramm wird der Fehler wieder mit einem `try-catch`-Block abgefedert:

```
{
   try {
      convert (gleitpunkt, festpunkt);
      std::cout << festpunkt << " war gleitpunkt" << std::endl;
   }
   catch (const std::ios_base::failure & ex) {
      std::cout << ex.what() << " Fehler gleitpunkt = "
         << gleitpunkt << std::endl;
   }
}
```

Da wir nun wissen, dass die Streams Fehler-Bits setzen, bietet sich noch eine andere Möglichkeit an. Wir geben diese Bits als `bool` in der Funktion `convert` zurück.

```
template <class Eingabe, class Ausgabe>
bool convert(const Eingabe & eingabe, Ausgabe & ausgabe) {
   std::stringstream zwischen;
   zwischen << std::fixed << eingabe;
   return (zwischen >> ausgabe);
}
```

Das Hauptprogramm vereinfacht sich dadurch:

```
double gleitpunkt = 3.14;
int festpunkt;
...
if(convert (gleitpunkt, festpunkt))
   std::cout << festpunkt << " festpunkt war gleitpunkt"
               << std::endl;
else
   std::cout << " Fehler gleitpunkt = " << gleitpunkt << std::endl;
```

5.10.4 Benutzerdefinierte Exception-Klassen

Wenn man eigene Exception-Klassen entwerfen will, sollte man sie von der Basisklasse exception ableiten. Die Klasse ist wie folgt definiert:

Definition exception

```
class exception {
   public:
      exception() throw() { }
      virtual ~exception() throw();
      // Returns a C-style character string describing the general
      // cause of the current error.
      virtual const char* what() const throw();
};
```

Die Angabe `throw()` beim Konstruktor, Destruktor und der Funktion `what()` ist ein Versprechen, das keine Exceptions geworfen werden; `throw` hat keine Parameter. Eine derartige Angabe mit Parameter definiert, welche Exceptions geworfen werden können. Der Compiler überprüft es!

- Folgende Dateien müssen für das Beispiel included werden:

```
#include <iostream>
#include <sstream>
#include <string>
#include <exception>
#include <cxxabi.h> //zum demangle
```

- Die letzte Datei erlaubt es, die intern verkürzt dargestellten (mangled) Namen wieder lesbar zu machen.

Etwas unbequem ist die Funktion `what()`, weil sie die Information mit alten C-Strings darstellt. Wir haben stattdessen den Operator << überladen.

```
class ConversionException : std::exception {
public:
   ConversionException(const std::string &what) : std::exception(),
         m_what(what) {};
   friend std::ostream& operator<<(std::ostream& ost,
         const ConversionException & e) {
      return ost << e.m_what;
   };
   virtual ~ConversionException() throw() {};
private:
   std::string m_what;
};
```

Unsere Schablone mit der Funktion zum Wandeln ist mit einem umfangreichen Fehlertext versehen.

```
template <class Eingabe, class Ausgabe>
void conversion(const Eingabe & eingabe, Ausgabe & ausgabe) {
   std::stringstream zwischen;
   zwischen << std::fixed << eingabe;
   if(!(zwischen >> ausgabe)) {
      //conversion error
      size_t length; // fuer demangle
      int status;    // fuer demangle
      // hier wird ein umfangreicher Fehlertext zusammengestellt
      std::ostringstream errs;
      errs << "conversion of: __"
```

```
            << eingabe
            << "__ to: "
            << abi::__cxa_demangle(typeid(ausgabe).name(), 0,
                  & length, & status)// interner typ lesbar
            << " failed";
         throw ConversionException (errs.str());
      }
   }
```

Als erstes wird die Eingabe ausgeben. Dann folgt der Typ in den gewandelt werden sollte. Die Funktion `typeid()` gibt aber nur verkürzte Namen wieder, wie sie intern verwendet werden. `abi::__cxa_demangle` macht sie wieder lesbar. Nun kann endlich dieser Fehlertext mit der Fehlerklasse `ConversionException` geworfen werden.

```
   {
      try{
         double e = 1234.5e+20;

         conversion(e, t);
         std::cout << t << " war gleitpunkt " << std::endl;
         conversion(t, e);
         std::cout << e << " wieder gleitpunkt " << std::endl;

      }
      catch (const ConversionException & e){
         std::cerr << "ERROR: " << e << std::endl;
      }
   }
```

Im Hauptprogramm wird die Wandel-Funktion wie üblich mit einem `try-catch`-Block verwendet. Im `catch`-Block wird mit dem überladenen `<<` Operator das Objekt `e` der Fehlerklasse `ConversionException` ausgegeben.

- Wenn man sich an diese Struktur gewöhnt hat, kann man sehr übersichtliche Programme gestalten.
- Wie das letzte Beispiel gezeigt hat, kann man gerade beim Testen in die `catch`-Blöcke viel Information ausgeben, die das Fehlersuchen wesentlich erleichtert.

5.11 Standard Template Library (STL)

Die Standard Template Library (STL) ist eine C++Bibliothek mit Behälter-Klassen (container classes), Algorithmen (algorithms) und Iteratoren (iterators). Sie bietet viele elementare Algorithmen und Datenstrukturen. Die STL ist eine generische Bibliothek, weil ihre Komponenten parametriert sind. Fast jede Komponente ist eine Schablone (template).

- Eine umfangreiche Einführung und Definition der STL ist zu finden unter:

`http://www.sgi.com/tech/stl`

Behälter und Algorithmen (container, algorithm)

Behälter-Klassen können beispielsweise für beliebige Typen von Objekten instantiiert werden. - Siehe unsere Schablone `convert`, die beliebige Typen wandeln kann. `vector <int>` kann wie ein

gewöhnliches C-Array benutzt werde, ohne dass man sich um die dynamische Speicherbelegung kümmern muss.

```
vector <int> v(3);// Deklaration Vektor mit 3 Elementen
v[0] = 7;
v[1] = v[0] + 3;
v[2] = v[0] + v[1];// v[0] == 7, v[1] == 10, v[2] == 17
```

Algorithmen manipulieren Daten in Behältern. `reverse` dreht zum Beispiel die Reihenfolge der Elemente im Vektor um.

```
reverse(v.begin(), v.end());// v[0] == 17, v[1] == 10, v[2] == 7
```

`reverse` ist eine globale oder freie Funktion und damit keine Methode einer Klasse (member function). `reverse` kann also nicht nur Elemente von `vector`, sondern auch von `list` und sogar auch Elemente eines C-Arrays bearbeiten.

```
double a[6] = {1.2, 1.3, 1.4, 1.5, 1.6, 1.7};
reverse(a, a+6);
for(int i=0; i<6; ++i)
   std::cout << "a[" << i << "] = " << a[i];
```

- Der Bereich (range) `(a, a+6)` zeigt auf das erste Element und **hinter** das letzte Element!

Iteratoren (iterators)

Um im Beispiel die Reihenfolge des C-Arrays umzudrehen sind die Argumente von `reverse` vom Typ `double*`. Von welchem Typ sind aber die Argumente, wenn `reverse` `vector` oder `list` bearbeitet? Das ist genau das, was `reverse` deklariert, von welchem Typ die Argumente sein sollen, und was `v.begin()` und `v.end()` liefert.

Die Argumente von `reverse` sind Iteratoren! Iteratoren sind eine Verallgemeinerung von Pointer und Pointer selbst sind in diesem Sinne Iteratoren. Deswegen war es möglich, die Reihenfolge vom C-Array umzudrehen. `vector<int>::iterator` ist der Typ den `v.begin()` und `v.end()` zurückgeben.

Der folgende `find` Algorithmus aus STL benutzt Iteratoren:

```
template <class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T&
value) {
   while(first != last && *first != value) ++first;
   return first;
}
```

`find` hat drei Argumente: zwei Iteratoren für den Bereich und einen Wert, nach dem in dem Bereich gesucht wird. Jeder Iterator wird in dem Bereich `(first, last)` geprüft; gestoppt wird, wenn entweder ein Iterator, der auf den Wert `value` zeigt, gefunden oder das Ende erreicht ist.

`first` und `last` sind als `InputIterator` deklariert; das ist eine Schablone für alle möglichen Typen. Der Compiler substituiert beim Übersetzen den formalen Typ-Parameter `InputIterator` und `T` mit den aktuellen Typen. Wenn die beiden ersten Argumente vom aktuellem Typ `int*` sind und der dritte vom aktuellen Typ `int` ist, generiert der Compiler folgenden Code:

```
int* find(int* first, int* last, const int& value) {
   while(first != last && *first != value)++first;
   return first;
```

```
}
```

STL-Konzepte (concepts)

Der Compiler hat allerdings noch ein Problem: `int*` als Typ für `first` und `last` macht ja noch Sinn, aber was soll `*first` bedeuten? Der Dereferenzierungs-Operator `*` ist doch Unsinn für `int`. Die generelle Antwort ist, dass `find` implizit Regeln definiert, die für einzelne Typen (hier `int`) beim Instantiieren erfüllt werden müssen. `InputIterator` ist also kein `type` sondern ein `concept`, das die angesprochenen Regeln definiert. Folglich ist `int*` in der Ausdrucksweise von STL ein `model` von `InputIterator`. Der Vorteil davon ist, dass das Konzept für sich steht und eben nicht nur für `find`, sondern generell für `vector`, `list`, C-Arrays und viele andere Typen gilt.

- Als Regel für den Hausgebrauch: Wenn `iter` als `iterator` deklariert ist, ergibt `*iter` das Element an dessen Position. In Kapitel 5.9.6, Seite 91 hatten wir das schon verwendet.

Was für `find` im besonderen gilt, trifft im Allgemeinen für Behälter (container) zu: Will man find anwenden, so müssen alle Argumente Modelle von `InputIterator` sein.

Behälter (container)

Ein Behälter ist ein Objekt, das andere Objekte (seine Elemente) speichert. Er besitzt Methoden, um auf diese Elemente zugreifen zu können. Jeder Typ, der ein Modell des Behälters ist, hat einen zugehörigen Iterator, der es erlaubt, auf alle Elemente zuzugreifen. Der Behälter ist Eigentümer der Elemente: Seine Elemente können nicht länger leben als er selbst. Es existieren folgende Funktionen:

```
a.begin()
a.end()
a.size()
a.max_size()
a.empty()
a.swap(b)
```

- Das sind nur wenige. Die folgenden Container-Klassen haben wesentlich mehr zu bieten:

```
<vector>    eindimensionaler Vektor
<list>      doppelt verkettete Liste
<queue>     Warteschlangen, Einfügen von Elementen nur am Anfang,
            Entfernen von Elementen nur am Ende
<deque>     Warteschlange, die auch Zugriff auf
            beliebige Positionen erlaubt
<stack>     Keller, Einfügen und Löschen nur am oberen Ende
<map>       assoziatives Array; mit "Keys" und "Values"
<set>       Menge
<bitset>    Menge von Booleans, für Bitfolgen feststehender Größe
```

sichere Schleife

- Statt einer normalen Schleife kann man die folgende Funktion verwenden. Sie hat den Vorteil, dass keine Speicherverletzung auftreten kann.

```
for_each(v.begin(), v.end(), verarbeite);
```

  Diese Funktion übergibt die einzelnen Elemente der Funktion `verarbeite`.

```
void verarbeite(int element) {
   std::cout << element << std::endl;
}
```

5.12 Sonstiges

5.12.1 Prototypen für Funktionen und C-Parameterdeklarationen

Im Gegensatz zu Standard-C sind Funktionsprototypen immer erforderlich. Dies ist einer der Bereiche, in denen C++ keine vollständige Obermenge zu C bildet.

Beispiel 1:

Der folgende Code wird unter C++ nicht kompiliert:

```
// Programm ohne Funktionsprototyp, kompiliert nicht.
#include <iostream>
void main() {
   display("Hallo");
}
void display(char *s) {
   std::cout << s;
}
```

Hier ist das Einfügen eines Funktionsprototypen

```
void display(char *s);
```

vor dem Aufruf von `main()` erforderlich.

Die früher in C verwendete Form der Parameterdeklaration ist in C++ ebenfalls unzulässig.

Beispiel 2:

```
// unzulässig in C++:
void display(s)
char *s {
   std::cout << s;
}
```

5.12.2 Plazierung von Variablendeklarationen

Im Gegensatz zu C müssen Variablendeklarationen in C++ nicht am Beginn eines Blocks stehen, solange Variablen vor ihrer Benutzung deklariert werden.

Durch die Deklaration von Variablen nahe der Stelle an der sie benutzt werden soll der Code lesbarer werden. Dies sollte jedoch nur für wirklich lokal eingesetzte Variablen eingesetzt werden, da ansonsten die Variablendeklarationen schwer zu finden sind.

Beispiel 1:

```
// Deklaration einer Variablen nahe ihres Einsatzes:
#include <iostream>

void main() {
  std::cout << "Geben Sie bitte einen Wert ein: ";
  int wert;
```

```
  std::cin >> wert;
  std::cout << "Der Wert ist " << wert << '.\n';
}
```

So ist beispielsweise folgender Ausdruck zulässig (scheitert aber trotzdem z.B. in älteren gcc-Versionen am Compiler):

```
for (int zaehler = 0; zaehler < MAXCTR ; zaehler++)
```

Unzulässig sind dagegen Ausdrücke wie:

```
if(int i == 0)     // unzulässig!
```

oder:

```
while(int j == 0)  // unzulässig!
```

Achtung:

Bei der Platzierung von Variablendeklarationen ist auf den Gültigkeitsbereich der Variablen zu achten.

Beispiel 2:

```
// Platzierung von Variablendeklarationen:
#include <iostream>

void main() {
   for (int lineno = 0; lineno < 3; lineno++) {
      int temp = 22;
   std::cout <<"Dies ist Zeile "<< lineno << " temp ist " << temp
      << "\n";
   }
   if (lineno == 4) // lineno ist noch gültig
      std::cout << "Oops!";
   // temp ist hier nicht verfügbar!
}
```

Im Beispiel ist `temp` nur innerhalb der **`for`**-Schleife verfügbar, `lineno` dagegen ab der Deklaration bis zum Ende von **`main()`**.

5.12.3 Namensräume (Bezugsoperator „::“)

Global deklarierte Größen landen automatisch in einem globalen Namensraum, der sich von anderen dadurch unterscheidet, dass er keinen Namen hat. Darum steht vor dem Bezugsoperator „::“ (Scope-Operator) kein Name.

Beispiel:

```
// Scope operator
#include <iostream>
namespace {
int wert = 123;     // globale Variable
}

int main(){
 int wert = 456;    // lokale Variable
 std::cout << ::wert;    // global Variable
 std::cout << '\n';
```

```
    std::cout << wert;      // lokale Variable
}
```

In diesem Beispiel lautet die Ausgabe:

```
123
456
```

Generell sollten aber explizite Namensräume dem globalen vorgezogen werden, d.h. freischwebende globale Variable vermieden werden.

```
namespace Allgemein {
int wert = 123;     // globale Variable
}
int main(){
   ...
   std::cout << Allgemein::wert;    // global Variable
   ...
}
```

Bei allen Dingen (z.B. `cout`) aus der C++Bibliothek haben wir ja schon immer den `namespace std` verwendet. Interessant ist der Namensraum bei abgeleiteten Klassen. Jede Klasse hat ihren eigenen!

Beispiel: Namensraum bei Vererbung

```
#include <iostream>
class Base {
   public:
      void Func(int Param){
         std::cout << "Func in Base: " << Param << std::endl;
      }
};

class Derived : public Base {
   public:
      void Func(double Param){
         std::cout << "Func in Derived:" << Param << std::endl;
      }
};

int main() {
   Derived d;
   d.Func(10);
   d.Func(12.4);
   return 0;
}
```

In diesem Beispiel lautet die Ausgabe:

```
Func in Derived: 10
Func in Derived: 12.4
```

Man könnte sich nun wünschen, dass die Festpunktzahl von `Base` und die Gleitpunktzahl von `Derived` verarbeitet würde. Der Compiler kennt aber nur die Funktion `Func` in `Derived`, weil sie die von `Base` überschreibt.

`d` ist eine Instanz der Klasse `Derived`. Wird über eine solche Instanz eine Methode aufgerufen, schaut der Compiler als erstes in den Namensraum der Klasse. Da er hier eine Methode Func findet, endet die Namensauflösung. Weitere Namensräume werden nicht mehr berücksichtig. Tatsächlich überdeckt `Func` aus `Derived` also `Func` aus `Base`.

Damit der Compiler beide Funktionen sehen kann, müssen sie sich in denselben Namensraum befinden. Dieses Ziel erreicht man mittels einer `using`-Deklaration:

```
#include <iostream>
class Base {
   public:
      void Func(int Param){
         std::cout << "Func in Base: " << Param << std::endl;
      }
};

class Derived : public Base {
   public:
      using Base::Func;
      void Func(double Param){
         std::cout << "Func in Derived:" << Param << std::endl;
      }
};
```

In diesem Beispiel lautet die Ausgabe nun:

```
Func in Base: 10
Func in Derived: 12.4
```

Die `using`-Deklaration befördert `Func` aus `Base` nun in den Namensraum von `Derived`. Der obere Aufruf führt dann zum gewünschten Ergebnis!

Wir hatten die `using` schon benutzt, um das `std` beim `cout` weglassen zu können. Das ist aber eine `using`-Direktive!

- `using`-Deklaration erstellt ein lokales Synonym:

```
using Base::Func;    //lokales Synonym für Base::Func
```

- `using`-Direktive transportiert alle Namen aus `std`, als wären sie global in dem Namensraum dort, wo die Direktive gegeben wird. Das kann zu Doppeldeutigkeiten führen. Der Compiler erkennt das nicht sofort, erst wenn ein derartiger Name benutzt wird.

```
using namespace std; // alle Namen aus std sind global
```

5.12.4 Das Schlüsselwort const

Unterschiede zu C

- In C++ (im Gegensatz zu C) werden Variablen, die mit dem Schlüsselwort **`const`** deklariert sind, genauso wie echte konstante Ausdrücke (so wie die Zahl 17) behandelt.
- So sind Ausdrücke wie der folgende möglich:

Beispiel 1:

```
const int SIZE = 5;
char string[SIZE];      // wäre illegal in C
```

- In C++ können **const**-Ausdrücke in Header-Files vorkommen (was in C beim mehrmaligen Verwenden des Header-Files einen Linker-Fehler erzeugt!).
- Gegenüber **#define**-Anweisungen haben **const**-Ausdrücke den Vorteil, dass sie in Debuggern als symbolischer Ausdruck zur Verfügung stehen.

Zeiger und const-Ausdrücke

Die Deklaration

```
const char *zeiger_auf_konst_char = 'a';
```

erzeugt einen Zeiger auf ein konstantes **char**-Objekt, das nach der Initialisierung nicht verändert werden kann.

Ein konstanter Zeiger wird mit

```
char *const konst_zeiger_auf_char = mybuf;
```

deklariert. Hier kann der Zeiger nicht verändert werden, der Inhalt, auf den er zeigt, jedoch sehr wohl.

Eine Kombination der beiden Formen ist natürlich ebenso möglich:

```
const char *const konst_zeiger_auf_konst_char = mybuf;
```

Beispiele

Folgende Deklarationen sind gültig (`konst_ch` ist ein Objekt vom Typ **const char**, `ch` ist ein Objekt vom Typ **char**).

```
const char *pch1 = &konst_ch;
const char *const pch4 = &konst_ch;
const char *pch5 = &ch;   // pch5 hat nur lesenden Zugriff auf ch
char *pch6 = &ch;
char *const pch7 = &ch;
const char *const pch8 = &ch;
```

Nicht erlaubt sind dagegen folgende Deklarationen:

```
char *pch2 = &konst_ch;          // Fehler, nicht erlaubt
char *const pch3 = &konst_ch;    // Fehler
```

Anwendung von Zeigern auf konstante Objekte

Ein häufiger Einsatz von Zeigern auf konstante Objekte sind Funktionsdeklarationen.

```
char *strcpy(char *Ziel, const char *Quelle);
```

Hier wird sichergestellt, dass `Quelle` nicht von der Funktion `strcpy` manipuliert werden kann. Da es eine Standardumwandlung von `typename *` nach `const typename *` gibt, kann der Funktion eine Variable vom Typ `char *` übergeben werden.

Referenzen auf Konstanten

Mit **const** kann eine Referenz (Vergleiche Abschnitt 5.3) auf eine Variable gebildet werden, die nur Lesezugriffe erlaubt:

```
int i;
const int &auch_i = i;
```

Methoden konstanter Objekte

Wird ein Objekt als **const** deklariert, z.B.

```
const Datum Geburtstag(3, 10, 1990);
```

so lässt der Compiler keinen Aufruf der Methoden des Objekts zu, da er nicht überwachen kann, ob eine Methode nur lesend oder auch schreibend wirkt.

Um Methoden, die nur lesend wirken, für solche Objekte freizugeben, können sie in der Klassendefinition mit dem Schlüsselwort **`const`** nach der Parameterliste gekennzeichnet werden.

Beispiel:

```
class Datum {
public:
   Datum(int tg, int mnt, int jhr);   // Konstruktor
   int getTag() const;
   int getMonat() const;
   int getJahr() const;
   int display() const;
   void setTag(int tag);
   ...
private:
   int tag, monat, jahr;
}

// Wichtig: in der Definition der Methode ist das Schlüsselwort
// const zu wiederholen!
int Datum::getTag() const {
   return tag;
}
...

// Anwendung:
int x;
const Datum Geburtstag(3, 10, 1990);

x = Geburtstag.getTag();  // ok
Geburtstag.setTag(3);   // Fehler!
```

Konstante Attribute

Sind Attribute einer Klasse als konstant deklariert, so können sie nur vom Konstruktor gesetzt werden, indem der Konstruktor auf den Konstruktor des Attributs zurückgreift (auch für Standardtypen).

Beispiel:

```
class Wert {
public:
   Wert(int w); // Konstruktor
private:
   const int wert;
}

// falsche Lösung:
Wert::Wert(int w) {
```

```
   wert = w;    // Fehler!!!
}

// richtige Lösung:
Wert::Wert(int w) : wert(w) {}
```

5.12.5 Volatile

- Mit dem Schlüsselwort **`volatile`** deklarierte Variablen dürfen sich jederzeit ohne Einwirkung des Programms ändern, z.B. durch Hardware.
- Optimierungen des Compilers werden für solche Variablen unterbunden und bei jedem Zugriff des Programms auf eine solche Variable wird sie neu vom Speicher geladen (also nicht z.B. in Registern gehalten).
- Die für **`const`**-Pointer angegebenen Regeln (Abschnitt 5.12.4) gelten genauso für **`volatile`**-Pointer.

5.12.6 Statische Attribute und Methoden

Statische Attribute (Klassendaten)

- Sollen alle Instanzen einer Klasse ein gemeinsames Attribut haben, so kann dieses Attribut als statisches Attribut deklariert werden.
- Dieses Attribut existiert dann nur einmal und wird von allen Objekten dieser Klasse geteilt.
- Definition und Initialisierung eines statischen Datenelements **außerhalb** der Klassendefinition:

```
type class::identifier = value;
```

- Typische Anwendung ist das Zählen von Objekten.

Beispiel:

```
class Kiste {
   public:
      Kiste(void): num(0), demon(1){...}
      ...
   private:
      int num, demon;
      static int zaehlerKisten; // Deklaration
};

int Kiste::zaehlerKisten = 0; // Definition und Initialisierung
```

Statische Methoden (Klassenmethoden)

- Zugriff auf diesen Zähler mit passender Methode:

```
class Kiste {
   public:
      static int anzahlKisten(void){
         return zaehlerKisten;
      }
```

```
   private:
      int num, demon;
      static int zaehlerKisten;
};
```

- Aktualisierung bei jedem Konstruktor-Aufruf:

```
class Kiste {
   public:
      Kiste(void): num(0), demon(1){
         zaehlerKisten++;
      }
      ...
   private:
      int num, demon;
      static int zaehlerKisten; //
};
```

- Klassenmethoden können aufgerufen werden, ohne dass ein Objekt existiert:

Aufrufsyntax

```
class::identifier(parameter, ...);
```

Beispiel:

```
std::cout << Kiste::anzahlKisten(); // 0
Kiste kiste;
std::cout << Kiste::anzahlKisten(); // 1
```

5.12.7 Aufzähltypen (enum)

Wie in C können Aufzähltypen mit dem Schlüsselwort **enum** deklariert werden. Im Gegensatz zu C ist für anschließende Variablendeklarationen das Schlüsselwort **enum** nicht erforderlich (wie übrigens auch für struct).

Beispiel 1:

```
enum farbe {rot, blau, gruen, gelb};
farbe myColor;   // hier Schlüsselwort enum nicht erforderlich
```

Jeder Wert der Aufzählung entspricht einem Integer-Wert, standardmäßig beginnend mit Null. Es sind jedoch andere Deklarationen möglich:

Beispiel 2:

```
enum farbe {rot = 5, blau, gruen, gelb}; // Integer-Werte 5, 6, 7, 8
enum tag {montag = 1, dienstag, mittwoch = 7, donnerstag};
                                         // Integer-Werte 1, 2, 7, 8
enum richtung {nord = 1, sued, ost = 1, west};
                                         // Integer-Werte 1, 2, 1, 2
```

Von Aufzähltypen zu Integer-Werten existiert eine Standardumwandlung, jedoch nicht umgekehrt.

Beispiel 3:

```
int i;
farbe myColor = blau, yourColor;
i = myColor;            // ok, i = 1;
```

```
yourColor = i;          // nicht möglich!
yourColor = (farbe) i;  // erlaubt, aber fehleranfällig!
```

5.12.8 Linken von C-Programmen mit C++

Um C-Code mit C++-Programmen zu linken, ist im allgemeinen eine **`extern „C"`** Anweisung erforderlich.

```
extern "C"
{
  // hier kann C-Code stehen
}
```

Für Standard C include-Libraries ist dies normalerweise nicht erforderlich.

5.13 Modularisierung/Aufbau eines Projektes, Makefiles

5.13.1 Header- und Implementierungsdateien

- Der Programmcode wird in C++ normalerweise in Header- und Implementierungsdateien gegliedert.
- Empfohlene Endungen der Dateinamen: `*.h` und `*.cc`, alternativ auch `*.cpp` und `*.cxx` (die manchmal verwendete Endung `*.C` in UNIX wird wegen der Probleme beim Übergang in DOS-kompatible Dateisysteme nicht empfohlen).
- Als **`inline`** deklarierte Funktionen **müssen** in der Header-Datei stehen, da der Compiler Zugriff auf die Definition braucht.
- Normalerweise sollte für jede Klasse eine Header- und eine Implementierungsdatei verwendet werden (außer die Klassen sind sehr klein oder eng verwandt).

5.13.2 Makefiles

Begriffe:

Linken

Zusammenfassen mehrerer kompilierter Objektdateien und Bibliotheken zu einer ausführbaren Datei oder Bibliothek.

Makefile:

- Datei, die Anweisungen zum Kompilieren, Linken und Installieren von Programmen und Bibliotheken enthält.
- Das Makefile wird von einem Kommandogenerator (wie z.B. make oder gmake) ausgeführt.
- Eingesetzt meist unter UNIX-Systemen.

Struktur von Makefiles:

- Einträge in Makefiles haben folgende Struktur:

```
<Target>: <Dependency-Liste>
   <Regel>
```

- Wichtig: Vor der Regel steht ein Tabulator-Zeichen, **keine** Blanks!

Beispiel:

- Der Quellcode für `meinProgramm` setzt sich zusammen aus den Dateien `klasse1.cc`, `klasse1.h`, `klasse2.cc` und `klasse2.h` und greift darüberhinaus auf Funktionen aus der Bibliothek `libMeineLib.a` zu.
- Die Header-Dateien sind über `#include`-Kommandos in den `*.cc`-Dateien enthalten und müssen daher im Makefile nur in den Dependency-Listen erscheinen.

Makefile:

```
meinProgramm: klasse1.o klasse2.o
   g++ -o meinProgramm klasse1.o klasse2.o libMeineLib.a
klasse1.o: klasse1.cc klasse1.h
   g++ -c klasse1.cc
klasse2.o: klasse2.cc klasse2.h
   g++ -c klasse2.cc
```

Aufruf:

```
gmake meinProgramm
```

6 Entwicklung objektorientierter Software

Hinweis:

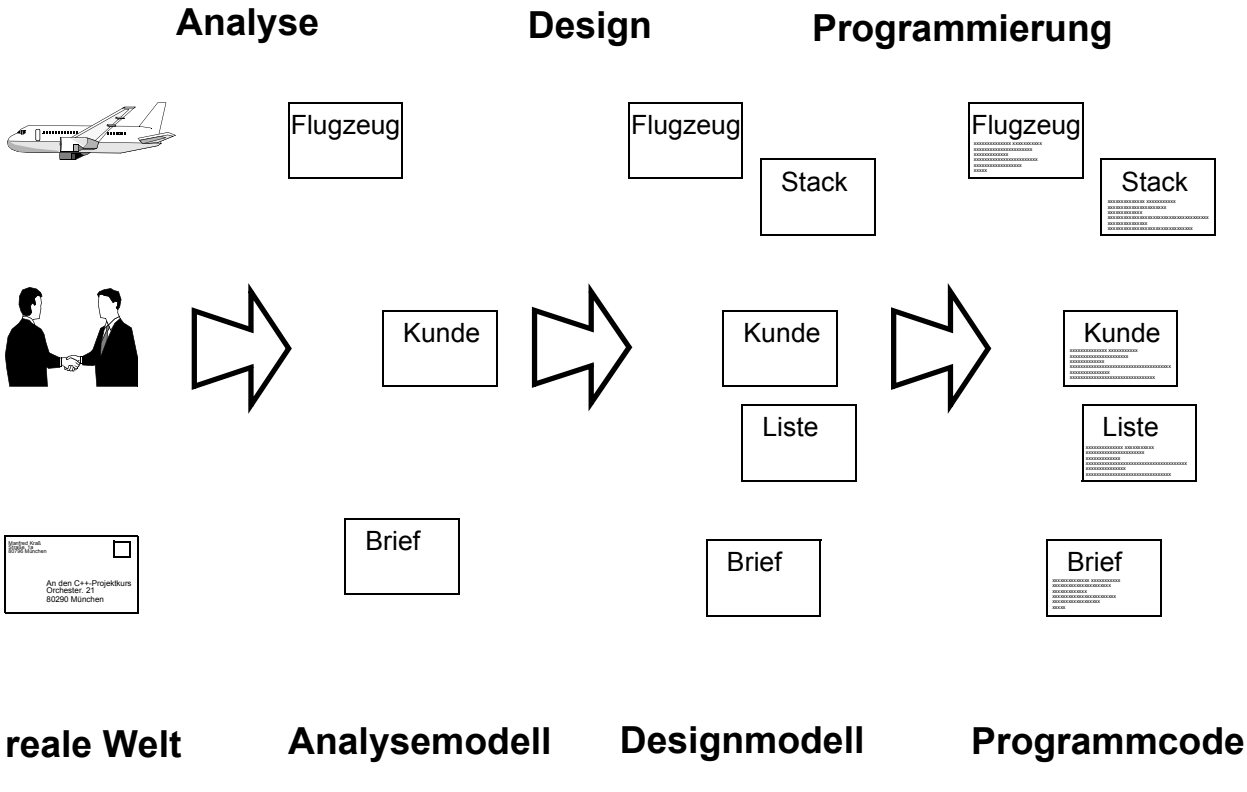
Dieses Kapitel im Rahmen der Veranstaltung „Objektorientiertes Programmieren in C++“ kann nur einen knappen Überblick über den objektorientierten Entwurfsprozess geben.

Ausführlichere Darstellung in anderen Veranstaltungen („Software-Engineering“) und in der Literatur (insbesondere [Booch 95], [Rumbaugh 93], [Coad 91]).

Einen guten Überblick über die Thematik bietet [Schäfer 94].

6.1 Der objektorientierte Entwurfsprozess

6.1.1 Überblick (aus [Schäfer 94], Seite 23)



Zielsetzung und Vorgehen

- Struktur des Problembereiches soll möglichst genau auf die Implementierung abgebildet werden.
- Sanfter Übergang zwischen den Phasen des objektorientierten Entwurfsprozesses.
- Durchgängigkeit der Modelle von Analyse über Design bis zur Implementierung.
- Stabilität des Entwurfsprozesses erleichtert Wiederverwendbarkeit von Analyse-, Design- und Programmierergebnissen.
- Ein gutes Design erfordert den Aufbau entsprechender Erfahrung.

Methoden

Notation und Vorgehen gemäß objektorientierten Entwurfsmethoden:

- Object Modeling Technique (Rumbaugh u.a.)
- OOA und OOD (nach Coad und Yourdon)
- Object-Oriented Analysis and Design (Grady Booch)
- weitere: Jacobsen, Shlaer und Mellor, Wirfs-Brocj, ...
- neu: UML Unified Modeling Language (Booch, Rumbaugh, Jacobsen) (nur Notation!), siehe [OMG UML] Seite 130

Modelltypen

- Beschreibung eines Systems nicht durch ein einziges Modell (eine einzige Sicht, ein einziger Blickwinkel) möglich.
- Alle Methoden verwenden daher mehrere Modelle (verschiedene Diagrammformen).
- Vergleich verschiedener Sichten erlaubt das Aufdecken von Unvollständigkeiten und Widersprüchen.
- Unterscheidung: statische und dynamische Modelle
  - Klassenhierarchie (mit Assoziationen und Aggregationsbeziehungen) ist eine statische Sicht (z.B. Klassendiagramm).
  - Objekte (Instanzen der Klasse) sind dynamisch (sie können während des Programmablauf entstehen, sich verändern oder verschwinden).
  - Ablauf der Kommunikation zwischen Objekten ist ebenfalls dynamisch (darstellbar z.B. durch Nummerierung von Pfeilen oder getrennte Zeitablaufdiagramme).
- Unterscheidung: logische und physikalische Modelle
  Logisches Modell betrachtet nicht physikalische Voraussetzungen (wie hardwarenahe Implementierungsdetails, Aufteilung des Programmablaufs auf mehrere Prozessoren oder Aufteilung des Source-Codes auf einzelne Dateien), sondern abstrakte Konzepte (Klassen, Objekte und ihre Strukturen)
- Logisch/physikalisch und statisch/dynamisch sind orthogonale Unterscheidungen.

6.1.2 Objektorientierte Analyse (OOA)

Was ist OOA?

- Untersuchen und Beschreiben des Systembereiches
- Unabhängig von der Programmiersprache
- Präzisierung des Problems (gegenüber mehrdeutigen, unvollständigen oder inkonsistenten Beschreibungen in natürlicher Sprache)
- Wichtig: Klärung der Frage „was“ das System tun muss, nicht „wie“ es zu implementieren ist

Nutzen und Ziele

- Verringerung der Komplexität:
  Herausfiltern der für die Aufgabe wichtigen Aspekte
- Visualisierung:
  - Objektmodelldiagramme machen Ideen klarer und überprüfbar, ob sie praktikabel sind
  - Erkennen von vergessenen Schnittstellen, unnötigen Komponenten oder konzeptionellen Schwächen
- Kommunikation mit Problemexperten, Anwendern und Kunden über einfache aber eindeutige Notation
- Wiederverwendbarkeit von Analyseergebnissen
- Testen von Einheiten bereits vor Ihrer Fertigstellung
  (Beispiel: Durchspielen von Szenarien in Simulationen)

Schritte der OOA

- Ausgangspunkt: Problembeschreibung in Form eines Pflichtenheftes o.ä. (kann unvollständig oder in der Form unbrauchbar sein)
- Identifizieren von Klassen und Objekten
- Identifizieren der Beziehungen zwischen den Klassen und Objekten
- Beschreibung der Klassen und Objekte durch Definition von Attributen und Methoden
- Bildung von Subsystemen und Definition logischer Abläufe (Funktionalität wird grundsätzlich mit Objekten in Verbindung gebracht)
- Zielsetzung: Nachbildung der Realität

Identifizierung von Klassen und Objekten

- Identifikation der Klassen wird in den einzelnen Methoden auf unterschiedliche Art vorgenommen (Heuristiken, grammatikalische Untersuchung des Pflichtenheftes, Ableitung aus Datenflussdiagrammen, ...)

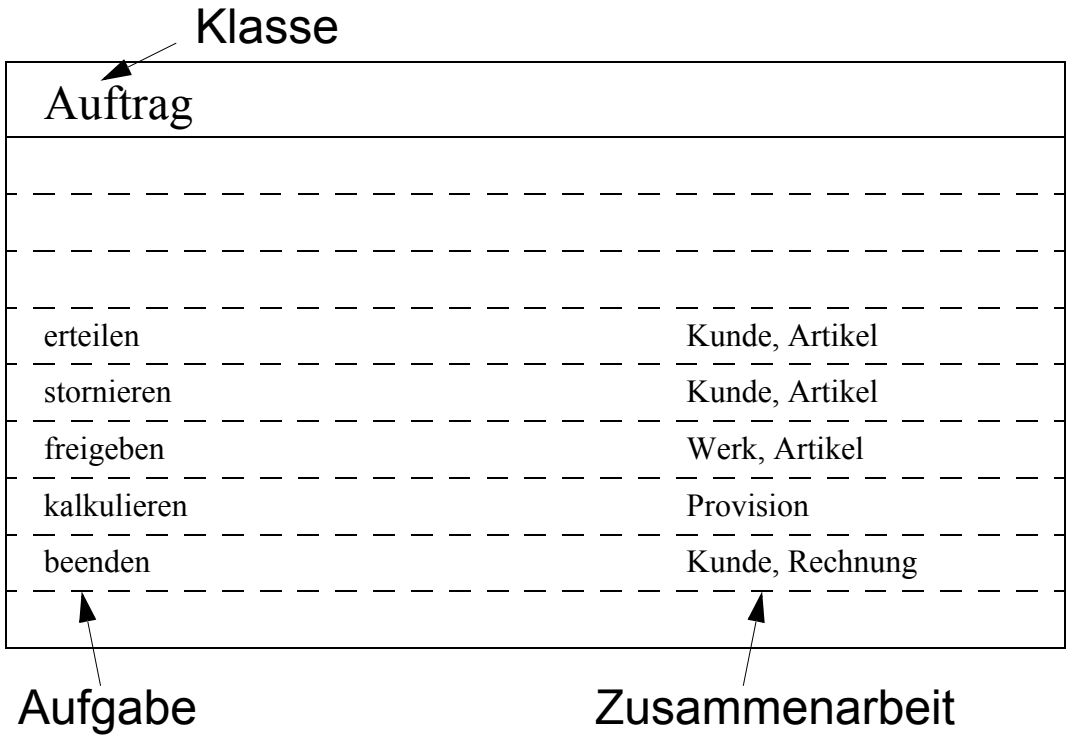
grammatikalische Untersuchung des Pflichtenheftes

- pseudo-formale Methode:
  Substantive bilden mögliche Klassen
  Verben charakterisieren mögliche Methoden
- Verfahren stellt nicht sicher, dass man alle Klassen, Methoden und Attribute findet
- Entwickler wird nicht gezwungen, den Problembereich zu verstehen
- bestenfalls als Einstieg geeignet

CRC-Methode

- einfache, aber wirkungsvolle Methode (nach Beck und Cunningham)
- eine Karteikarte für jeden Klasse (Class): Eintrag von Aufgaben (Responsibilities) und Beziehungen zu anderen Klassen (Collaborations)

- Beispiel für CRC-Karte:



- guter Einstieg (auch in weiterreichende Methoden)

6.1.3 Objektorientiertes Design (OOD)

- Abbilden des Realitätsmodells auf den Lösungsbereich
- Vorstufe der Umsetzung in eine Programmiersprache
- Hinzufügen von Klassen, Attributen und Methoden, die für die Realisierung gebraucht werden (z.B. verkettete Listen)
- Designüberlegungen wie Wahl des Betriebssystems und der Programmiersprache (sofern diese Wahlmöglichkeit besteht)
- Einbeziehung bestehender Entwürfe und Klassenbibliotheken (z.B. Grafik-Library) in das Design
- Übergang von OOA nach OOD fließend, Zielsetzung ist jedoch sehr unterschiedlich (OOA: Modellierung der Realität, OOD: „Erfinden“ von Abstraktionen und Mechanismen, um die gewünschte Funktionalität des Programms zu erbringen)

6.1.4 Objektorientierte Programmierung (OOP)

- Umsetzung des erstellten Designmodells in eine Implementierungssprache, z.B. C++.
- Basis ist das Ergebnis der OOD, nicht mehr das Pflichtenheft (jedoch iterative Rückkehr zu OOA und OOD möglich).
- siehe vorangegangene Kapitel!

6.1.5 Software-Lebenszyklus

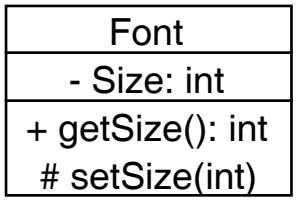
- Gesamter Software-Lebenszyklus integraler Bestandteil der objektorientierten Methoden.
- Integration von Tests in OOA, OOD und OOP.
- Während der Wartungsphase (z.B. Erweiterung des Programms, Portierungen, etc.) kann in die Analyse- und Designphase zurückgegangen werden (da diese ja nie vollkommen abgeschlossen werden).
- Erleichterte Wiederverwendbarkeit von Ergebnissen der Analyse-, Design- und/oder Programmierphase.

6.2 Objektorientierte Entwurfsverfahren

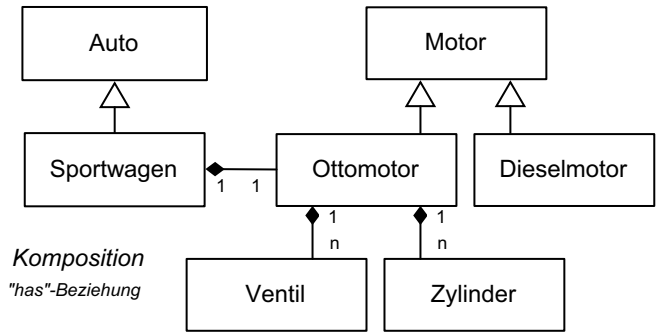
- Vorbemerkung: Booch-Notation wird abgelöst durch eine gemeinsame Notation (nicht Methode!) von Booch, Rumbaugh und Jacobsen („Unified Modeling Language“) (Version 2.0 kurz vor der Verabschiedung; siehe [OMG UML]).
- Methode anwendbar auf alle objektorientierten und objektbasierten Programmiersprachen (Smalltalk, C++, ObjectPascal, Ada, CLOS, ...).
- Booch-Methode erlaubt sehr detaillierte Systembeschreibung (nicht immer erforderlich oder erwünscht).
- sechs verschiedene Diagrammformen zur Modellierung verschiedener Aspekte des Systems (nicht immer alle notwendig)
- Wichtige Elemente (Klassen, Methoden, Objekte und Module) können zusätzlich durch sog. *Spezifikationen* detaillierter formal beschrieben werden (Formulare, die nach Bedarf während des Entwurfsprozesses ausgefüllt werden).

6.2.1 Klassendiagramm

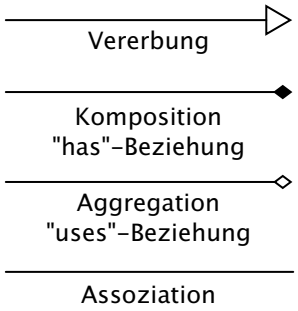
- statische/logische Sicht
- Ein oder mehrere Klassendiagramme beschreiben die Klassen des Systems und ihre Beziehungen untereinander.
- Unterschiedlicher Typen von Klassen: normale Klasse, parametrierte Klasse, instantiierte Klasse, Metaklasse (nur in Smalltalk), abstrakte Klasse.
- *wichtige* Methoden und Attribute sollen im Klassendiagramm dargestellt werden



- Beispiel für Klassenmodell (aus Abschnitt 3.4)



Beziehungen zwischen Klassen



Subsysteme

Größere Systeme können in Subsysteme untergliedert werden.

6.2.2 Zustandsdiagramm

- Zustandsübergangsdiagramme können als Ergänzung der Klassenbeschreibung verwendet werden.
- Einfache oder hierarchische (nebenläufige) Zustandsdiagramme zur Beschreibung der möglichen Zustandsübergänge innerhalb einer Klasse.

6.2.3 Spezifikation von Klassen

Beispielhaft für Spezifikationen wird hier die Spezifikation von Klassen betrachtet.

Spezifikationen sind ähnlich Formularen und werden nach Bedarf nach und nach ausgefüllt.

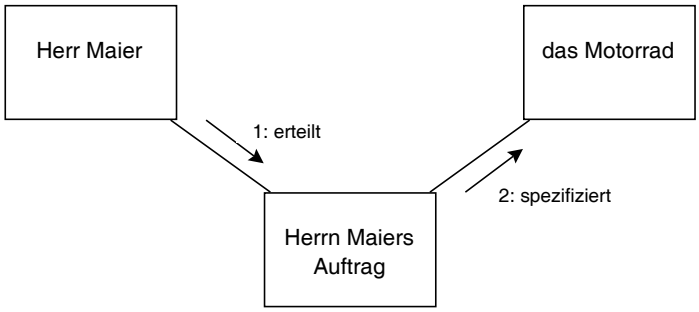
Felder der Klassenspezifikation

- Name
- Responsibilities:
  beschreibt Aufgaben (in Textform)
- Attributes, Operations, Constraints:
  Liste der Attribute und Methoden
- State machine:
  Verweis auf ein oder mehrere Zustandsdiagramme

- Export control:
  public: Klasse ist außerhalb ihres Subsystems bekannt
  | implementation: Klasse wird nur intern zur Implementierung benutzt
- Cardinality:
  Anzahl der erlaubten Instantiierungen der Klasse (0 für abstrakte Klassen)
- Parameters:
  Liste der Parameter (für parametrierte Klassen)
- Persistance:
  transient | persistant (Instanzen der Klasse sind transient oder persistent)
- Concurrency:
  sequential | guarded | synchronous | active (Aussage für nebenläufige Systeme)
- Space complexity:
  Aussage über ungefähren Speicherplatzbedarf einer Instanz

6.2.4 Objektdiagramm

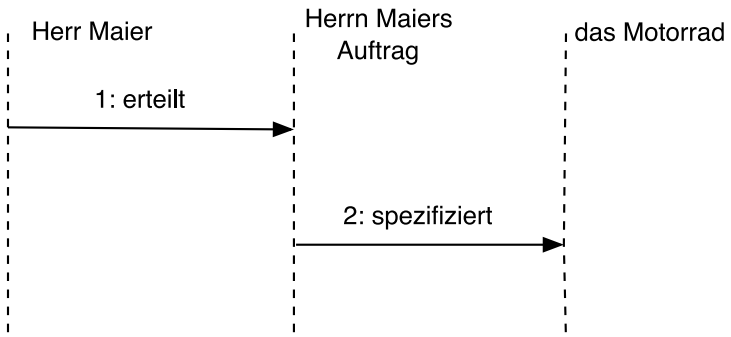
- dynamische/logische Sicht
- Momentaufnahme des Systems (da Objekte flüchtig sein können)
- Beziehungen aus Klassendiagrammen spiegeln sich in Objektdiagrammen wider.



6.2.5 Interaktionsdiagramm

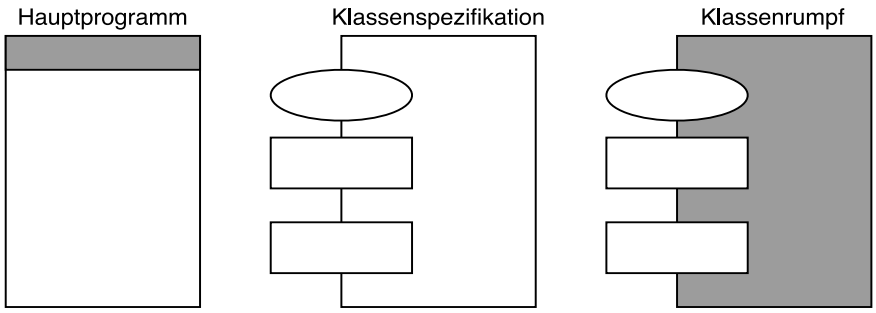
- dynamische/logische Sicht
- Ergänzung der Objektbeschreibung (bessere Visualisierung, keine andere Information)
- Zeitablaufdiagramme mit Objekten (vertikale Linien) und Nachrichten (horizontale Pfeile)
- keine Übergabeparameter, Return-Werte

- Modellierung von Nebenläufigkeit durch Markieren von gleichzeitig aktiven Objekten möglich



6.2.6 Moduldiagramm

- statische/physikalische Sicht
- Zuordnung von Klassen zu physikalischen Programmteilen
- Kennzeichnung von Compiler-Abhängigkeiten durch Pfeile
- Verwendete Notation:



6.2.7 Prozessdiagramm

- dynamische/physikalische Sicht
- Darstellung der Aufteilung von Teilprogrammen oder Prozessen auf unterschiedliche Prozessoren oder Rechner
- Einbeziehung externer Geräte, Hardware
- Modellierung von Nebenläufigkeit
- wird hier nicht weiter betrachtet

6.2.8 Der Entwurfsprozess

Überblick

- Unterscheidung eines Macro-Prozesses (OOA, OOD, OOP) und eines Micro-Prozesses
- „Round-trip Gestalt Design“ (inkrementelles und iteratives Vorgehen):
  - Identifizierung von Klassen und Objekten
  - Identifizieren der Semantik (der Aufgaben) dieser Klassen und Objekte

  - Identifizieren der Beziehungen zwischen den Klassen und Objekten
  - Spezifikation der Implementierung von Klassen und Objekten
- Vorgehen in den einzelnen Phasen (Analyse, Design) ähnlich, Zielsetzung jedoch sehr unterschiedlich.
- Bei jedem Durchgang des Micro-Prozesses Sprung auf niedrigere Abstraktionsebene.

Identifizierung von Klassen und Objekten

- Analyse: Suchen von Klassen und Objekten im Problembereich
- Design: Identifizierung von Klassen, die zur Abbildung auf Software nötig sind (z.B. verkettete Listen)
- Implementierung: Identifizieren von weiteren Klassen, die die Systemarchitektur vereinfachen
- Vorgehen:
  - Heuristiken,
  - Szenarien zur Beschreibung von Ereignissen,
  - „Use-Case-Analysis“ (nach Jacobsen) zur Betrachtung von Systemabläufen
  - System-Entwickler muss sich mit der Terminologie des Systembereiches auseinandersetzen.
- Unterstützung durch CRC-Karten, Data Dictionaries (in die alle identifizierten Klassen und ihre Eigenschaften eingetragen werden).

Identifizieren der Semantik (der Aufgaben) der Klassen und Objekte

- Betrachten des Verhaltens der Klassen und Objekte (nicht der Attribute).
- Unterstützung durch CRC-Karten, später durch Objekt- und Interaktionsdiagramme, evtl. auch Zustandsdiagramme.
- sowohl isolierte Betrachtung der Klasse als auch Suche nach Mustern und Gemeinsamkeiten zwischen Klassen
- schrittweise Verfeinerung der CRC-Karten, des Data Dictionaries oder der Spezifikationen
- Festlegung des public-Teils des Klassenheaders

Identifizieren der Beziehungen zwischen den Klassen und Objekten

- besonders kreativer Teil des Entwurfs
- Erkennen kooperierender Objekte und Systemgrenzen
- Beschreibung von Beziehungen wie Vererbung, Assoziation und (später) Aggregation oder Instantiierung von generischen Klassen
- Verfeinerung der Klassen- und Objektdiagramme
- Neugestaltung der Klassenorganisation (z.B. durch Einführung von Vererbungsbeziehungen)

Spezifikation der Implementierung von Klassen und Objekten

- Festlegung des inneren Aufbaus von Klassen (Attribute und Methoden)
- Identifikation weiterer Klassen (Übergang zum nächsten Micro-Prozess)
- Ziel: detaillierte Spezifikationen, Klassen- und Zustandsdiagramme (ggf. auch Interaktionsdiagramme)
- Modellierung der physikalischen Architektur (in späteren Phasen)

6.3 Tool-Unterstützung

Entwicklung großer Softwaresysteme erfordert Werkzeuge (Tools)

CASE Computer-Aided Software Engineering

Arten von Werkzeugen:

- Grafik-Editoren
  - Grafikprogramm (idealerweise mit umfangreichen Zusatzfunktionalitäten)
  - Methodenunterstützung durch vorgegebene Diagrammformen
  - Navigieren durch die unterschiedlichen Sichten (Herausheben von Wichtigem gegenüber Unwichtigem)
  - Mehrbenutzerbetrieb für Teamarbeit
  - Syntaktische und semantische Überprüfung des Modells
- Code-Generatoren
  - automatische Erstellung von Klassendefinitionen
  - Erstellung von Makefiles (z.B. aus Moduldiagrammen)
  - Reverse-Engineering (z.B. zur Unterstützung in der Wartungsphase)
  - Gefahr: aus allgemeingültigem Entwurf kann „grafische Programmierung“ werden
- Klassenbrowser
  Erlaubt einfaches Navigieren im Entwurf oder auch im Source-Code
- Inkrementelle Compiler und Linker
- Debugger
  Debuggen objektorientierter Programme schwieriger als bei funktionaler Gliederung
- CASE-Tools (Vereinigung von Einzel-Tools zu einem integrierten Paket)
  - Data-Dictionary im Hintergrund (als Grundlage für Konsistenzprüfungen und Code-Generation)
  - aktuelle Übersicht: http://www.jeckle.de/umltools.htm
  - speziell UML (kostenlose Tools) http://lmtm.de
- Konfigurationsmanager

Bewertung

- große Dynamik auf dem Markt objektorientierter CASE-Tools
- Viele Werkzeuge bieten nur eingeschränkte Funktionalität (Malprogramm als CASE-Tool).
- Erfolg eines Tools abhängig von der Verbreitung der unterstützten Methode (und umgekehrt).
- Selbst bekannte und verbreitete Programme sind derzeit noch stark in der Fortentwicklung.
- CASE-Tools können den Entwurfsprozess nur unterstützen. „Design and programming are human activities; forget that and all is lost.“ (Bjarne Stroustrup 1991, zitiert nach [Schäfer 1994])

A Hinweise zur Erstellung von C++-Programmen am LDV

Um im Rahmen von Programmierarbeiten qualitativ guten Code zu erstellen, der später von anderen verstanden und/oder wiederverwendet werden kann, empfehlen wir folgende Stil-Richtlinien bei der Erstellung von C++-Code zu beachten:

A.1 Einteilung des Source-Codes in Dateien

- Für jede Klasse soll eine Header(„*.h“)- und eine Implementierungs(„*.ca.“ oder „*.cpp“)-Datei erstellt werden.
- Jede Datei sollte einen **Dateikopf** (also einige Kommentarzeilen zu Beginn) enthalten, der kurz den Inhalt der Datei, den Autor, das Datum und eventuell die Version und vorgenommene Änderungen an der Datei beschreibt.

Beispiel:

```
//------------------------------------------------------------------
// StringMa.h
//
// Autor:             Max Mustermann
// Datum:             1. Januar 1997
// Beschreibung:      Enthaelt die Deklaration der Klasse
//                    "StringManip" zur Manipulation von Strings.
//                    Bestandteil der Klassenbibliothek "MEGA++".
// Aenderungen:
//------------------------------------------------------------------
```

A.2 Header-Dateien

- Die Header-Datei enthält:
  - Klassendeklarationen
  - evtl. Funktionsprototypen (z.B. Operator-Deklarationen)
  - Implementierung von Inline-Funktionen
- Vor der Klassendeklaration wird die Aufgabe der Klasse in einigen Kommentarzeilen erläutert. Beginn und Ende dieses Kommentarblockes werden durch eine Zeile mit Füllzeichen optisch hervorgehoben.

Beispiel:

```
//---------------------------------------------------------------
// StringManip
//
// Diese Klasse dient diesem und jenem Zweck ....
// ...
//---------------------------------------------------------------
class StringManip {

}
```

- Innerhalb der Klassendeklaration sind die Attribute und vor allem die Methoden (einschließlich der Parameter und Rückgabewerte) in wenigen Kommentarzeilen zu erläutern. Die Methodendeklarationen enthalten die Namen der Parameter.

Beispiel:

```
public:
   // kopiereString kopiert den Inhalt von "string1" in einen
   // neu allozierten String (Rückgabewert der Funktion).
   // Ist "laenge" ungleich 0, so wird nur die entsprechende
   // Anzahl an Zeichen kopiert. Ist string1 kuerzer als diese
   // Anzahl, so ist "laenge" ohne Bedeutung.
   char* kopiereString(char* string1, int laenge = 0);
```

- Header-Dateien sind durch Präpozessoranweisungen vor Problemen durch mehrfach verschachtelte `#include`-Anweisungen zu sichern.

Beispiel:

```
#ifndef STRINGMA_H
#define STRINGMA_H

// ... hier steht der gesamte Inhalt der Header-Datei

#endif
```

A.3 Implementierungsdateien

- In Implementierungsdateien werden die Methoden und Funktionen möglichst in derselben Reihenfolge wie die Deklarationen in der Header-Datei implementiert.
- Vor jeder Methode/Funktion sind zwei bis drei Leerzeilen und ein Kommentarblock zu setzen. Zu beschreiben sind die Aufgabe, die Parameter und die Rückgabewerte der Methode/Funktion.
- Kommentare innerhalb der Implementierung einer Funktion sind erforderlich, wenn ein Teil ungewöhnlich, erklärungsbedürftig oder besonders raffiniert programmiert ist.
- Methoden bzw. Funktionen sollten kurz gefaßt sein, längere Abschnitte können oft durch neue Funktionen/private Methoden strukturiert werden.

A.4 Allgemeine Hinweise

- Kommentare erläutern Ihren Code und *nicht* was der Compiler macht. Beschreiben Sie also die Aufgabe von Methoden, sowie die Rückgabewerte und Parameter. Kommentare wie „Dies *ist der Konstruktor*“ sind überflüssig.
- Wo kein Parameter- oder Rückgabewert vorhanden ist, sollte grundsätzlich void und nicht *nichts* stehen.
- Empfohlene Einstellung für Tabulatoren und Einrückungen: 2 bis 4 Zeichen
- Die Zeilenlänge des Source-Codes sollte 80 Zeichen nicht übersteigen, um Probleme in verschiedenen Editoren und beim Ausdrucken zu vermeiden.
- In Programmen, die portierbar sein sollen (PC und UNIX) sind Umlaute in Kommentaren zu vermeiden.
- Später vorgenommene Änderungen an Programmcode (insbesondere an fremdem Code) sollten in Kommentaren an der entsprechenden Stelle im Code und in einer kurzen Anmerkung im Dateikopf erwähnt/beschrieben werden.

B Übersicht über C++ Operatoren

Sortiert und gruppiert nach Priorität	
Bereichsauflösung	Klassenname :: Element
Bereichsauflösung	Namensbereichs-Name :: Element
Global	:: Name
Global	:: qualifizierter Name
Elementselektion	Objekt.Element
Elementselektion	Zeiger -> Element
Indizierung	Zeiger [Ausdruck]
Funktionsaufruf	Ausdruck (Ausdrucksliste)
Werterzeugung	Typ (Ausdrucksliste)
Postinkrement	Lvalue ++
Postdekrement	Lvalue --
Typidentifikation	typeid (Typ)
Laufzeit-Typinformation	typeid (Ausdruck)
zur Laufzeit geprüfte Konvertierung	dynamic_cast<Typ>(Ausdruck)
zur Übersetzungszeit geprüfte Konvertierung	static_cast<Typ>(Ausdruck)
ungeprüfte Konvertierung	reinterpret<Typ>(Ausdruck)
const-Konvertierung	const_cast<Typ>(Ausdruck)
Objektgröße	sizeof Objekt
Typgröße	sizeof (Typ)
Präinkrement	++ Lvalue
Prädekrement	-- Lvalue
Komplement	~ Ausdruck
Nicht	! Ausdruck
unäres Minus	- Ausdruck
unäres Plus	+ Ausdruck
Adresse	& Lvalue
Dereferenzierung	* Ausdruck
Erzeugung (Belegung)	new Typ
Erzeugung (Belegung und Initialisierung)	new Typ (Ausdrucksliste)
Erzeugung (Platzierung)	new (Ausdrucksliste) Typ
Erzeugung (Platzierung und Initialisierung)	new (Ausdrucksliste) Typ (Ausdrucksliste)
Zerstörung (Freigabe	delete Zeiger
Feldzerstörung	delete [] Zeiger
Cast (Typkonvertierung)	(Typ) Ausdruck
Elementselektion	Objekt.*Zeiger-auf-Element
Elementselektion	Objekt->*Zeiger-auf-Element

Sortiert und gruppiert nach Priorität	
Multiplikation	Ausdruck * Ausdruck
Division	Ausdruck / Ausdruck
Modulo (Rest)	Ausdruck % Ausdruck
Addition	Ausdruck + Ausdruck
Subtraktion	Ausdruck - Ausdruck
Linksshift	Ausdruck << Ausdruck
Rechtsshift	Ausdruck >> Ausdruck
kleiner als	Ausdruck <Ausdruck
kleiner gleich	Ausdruck <=Ausdruck
größer als	Ausdruck >Ausdruck
größer gleich	Ausdruck >=Ausdruck
gleich	Ausdruck ==Ausdruck
ungleich	Ausdruck !=Ausdruck
bitweises UND	Ausdruck & Ausdruck
bitweises Exklusiv-Oder	Ausdruck ^ Ausdruck
bitweises ODER	Ausdruck | Ausdruck
logisches UND	Ausdruck && Ausdruck
logisches ODER	Ausdruck || Ausdruck
bedingte Zuweisung	Ausdruck ? Ausdruck : Ausdruck
einfache Zuweisung	Lvalue = Ausdruck
Multiplikation und Zuweisung	Lvalue *= Ausdruck
Division und Zuweisung	Lvalue /= Ausdruck
Modulo und Zuweisung	Lvalue %= Ausdruck
Addition und Zuweisung	Lvalue += Ausdruck
Subtraktion und Zuweisung	Lvalue -= Ausdruck
Linksshift und Zuweisung	Lvalue <<= Ausdruck
Rechtsshift und Zuweisung	Lvalue >>= Ausdruck
UND und Zuweisung	Lvalue &= Ausdruck
ODER und Zuweisung	Lvalue |= Ausdruck
Exklusiv-Oder und Zuweisung	Lvalue ~= Ausdruck
Ausnahme werfen	throw Ausdruck
Komma (Sequenzoperator)	Ausdruck , Ausdruck

Literatur zur Vorlesung

[Strasser 03] Strasser, Thomas; C++ Programmieren mit Stil, Eine systematische Einführung; dpunkt Verlag; 2003 (3898642216)

[Stroustrup 97] Bjarne Stroustrup; Die C++ Programmierspracheng Language, 3. Auflage; Addison-Wesley, 1997 (3827312965)

[Qualline 03] Qualline, Steve; Praktische C++ Programmierung; O'Reilly; 2003 (3897213583)

[Qualline 03] Qualline, Steve; How Not to Program in C++; No Starch Press; 2003 (1886411956)

[Josuttis 99] Josuttis, Nicolai; The C++ Standard Library - A Tutorial and Reference; Addison-Wesley; 1999 (online: http://www.josuttis.com/libbook/idx.html)

[ANSI C++ 03] Programming Language C++, ISO/IEC 14882-2003

[Booch 95] Grady Booch; Objektorientierte Analyse und Design; Addison-Wesley; 1995 (3893196730)

[Jacobson 99] Jacobson, Ivar; Das UML-Benutzerhandbuch; Addison-Wesley; 1999 (3827314860)

[Oestereich 04] Oestereich, Bernd; Objektorientiertes Softwareentwicklung; Oldenbourg; 2004 (3486272667); http://www.oose.de

[Coad 91] Peter Coad, Edward Yourdon; Object-Oriented Design; Prentice Hall, Englewood Cliffs, NJ; 1991 (0136300707)

[OMG UML] Unified Modeling Language Resource Page; Online-Informationen von der Object Management Group; http://www.omg.org/uml (gute UML Tutorials)

[STL] Standard Template Library Programmer's Guide; http://www.sgi.com/tech/stl

[Rumbaugh 93] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen; Objektorientiertes Modellieren und Entwerfen; Hanser Fachbuch; 1993 (3446175202)

[Schäfer 94] Steffen Schäfer; Objektorientierte Entwurfsmethoden; Addison-Wesley; 1994 (3893196927)

[C++ FAQ] Liste der häufig gestellten Fragen (mit Antworten) der Usenet News Group „comp.lang.c++"

[Object FAQ] Liste der häufig gestellten Fragen (mit Antworten) der Usenet News Group „comp.object".

Hinweis: Eine Einführung in C++ für C-Programmierer finden Sie unter:

`http://www.lrz-muenchen.de/~ebner/C++/Slides/cxxkurs_000.html`