

Programmieren in C++

(Skript basierend auf dem Skript von Prof. Schramm)

Skript nur zum persönlichen Gebrauch
Keine Weitergabe, keine Veröffentlichung, keine Vervielfältigung!

Prof. Dr. rer. nat. Carsten Meyer

Institut für Angewandte Informatik

Fachbereich Informatik & Elektrotechnik

Fachhochschule Kiel

Prof. Dr. rer. nat. Carsten Meyer

carsten.meyer@fh-kiel.de

Büro: C12-1.78, Tel. 0431 / 210 4107

Sprechzeiten: nach Vereinbarung

Biographie (1):

- 1986 – 1993: Physik-Studium in Göttingen und Freiburg / Brsg.

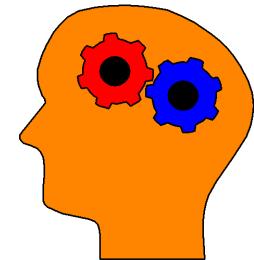


- 1993: Praktikum Managementberatung, Neuss

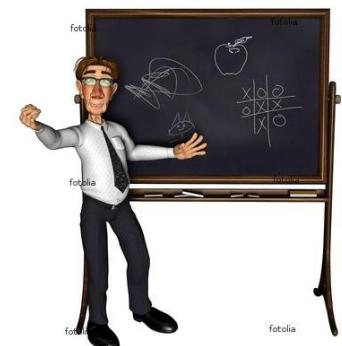


Biographie (2):

- 1993 – 1997: Promotion Universität Göttingen / Hebräische Universität Jerusalem
(Statistischer Physik neuronaler Netzwerke)



- 1998 – 2011: Wissenschaftler / Senior Scientist
Philips Forschungslabor Aachen / Hamburg
- seit März 2011: Professor an der FH Kiel



Ausgewählte Projekte, Philips Forschungslabor:



Gruppe Man-Machine-Interfaces (Aachen):

- Telefonische Auskunfts- und Dialogsysteme
- Sprachgesteuerte Navigationssysteme
- Projektleitung Professionelle Diktiersysteme

Gruppen Molecular / X-Ray Imaging Systems (Aachen):

- Simulation der dynamischen Herzbildgebung in der Nuklearmedizin
- Modellbasierte Analyse dynamischer Nuklearmedizinbilder des Herzens
- Projektleitung Entscheidungsunterstützungssysteme klinische Kardiologie
- Automatische EKG-Analyse
- Automatische Herzsegmentierung (CT, MR, 3-D Röntgen, Ultraschall)

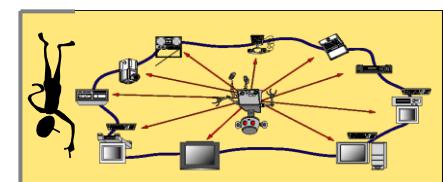
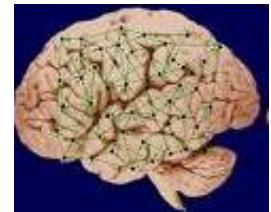
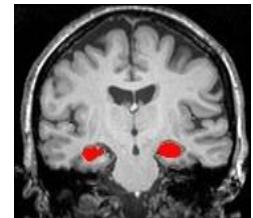
Gruppe Digital Imaging (Hamburg):

- Automatische Gehirnsegmentierung in 3-D MR-Bildern

Interessen:

Entwicklung und Anwendung intelligenter Algorithmen
in Bild-, Sprach- und Dokumentverarbeitung:

- Medizinische Bildverarbeitung und Bildverstehen
- Mustererkennung und maschinelles Lernen
- Informationsverarbeitung in neuronalen Netzwerken
- Medizinische Signalverarbeitung
- Mensch-Maschine-Schnittstellen
- Automatische Sprach- und Dokumentverarbeitung



Vorwort

Vorlesung behandelt 2 Themen:

- Objektorientierte Programmierung (OOP)
- Programmierung in C++

Übersicht (1)

Objektorientierte Programmierung (OOP):

- Einführung
- Objekt und Klasse
- Attribute, Methoden
- Geheimnisprinzip, Kapselung
- Vererbung
- Klassifikation
- Beziehungen zwischen Klassen
- Polymorphie

Schnellkurs C++:

- Einführung
- **Bekannte Sprachmittel:**
 - Variablen, Datentypen, Operatoren, Kontrollstrukturen, Arrays, Strukturen
- **Neue Sprachmittel:**
 - Referenzen
 - Funktionen: Vorgabeargumente, Überladung, Templates
 - Namensräume
 - Ein- und Ausgabe
 - Strings
 - Typumwandlung

Objektorientierte Programmierung mit C++:

- Klassen, Vererbung, Polymorphie, Mehrfachvererbung

Übersicht (2)

Objektorientierte Programmierung mit C++:

- **Klassen**
 - Klasse als Datentyp
 - Member: Instanzvariablen / -methoden, Klassenvariablen / -methoden, Konstruktor, Destruktor, this-Zeiger
 - Sichtbarkeit und Kapselung, Zugriffsspezifizierer, friends
 - Designempfehlungen: const, get/set,
 - Klassentemplates
 - Operatorüberladung für Klassenobjekte
 - Objektverwaltung: Erzeugen, Vergleichen, Kopieren, Auflösen, Komposition...
- **Vererbung**
 - Syntax und Einsatz
 - Basisklassen-Unterobjekt
 - Verdecken, Überschreiben, Überladen
 - Zugriffsmodifizierer, Zugriffsrechte

Übersicht (3)

Objektorientierte Programmierung mit C++:

- Polymorphie
 - Frühe und späte Bindung
 - Virtuelle Funktionen, virtueller Destruktor
 - Abstrakte Methoden
 - Abstrakte Klassen
- Mehrfachvererbung
- Fehlerbehandlung (exception handling)

Organisatorisches:

Veranstaltung gliedert sich in:

- 2 SWS Vorlesung:
 - Mittwochs 3. Block: jede Woche (Raum: Hörsaal 12)
- 1 SWS Labor (Herr Gabriel):
 - Dienstags / Mittwochs (je nach Gruppe), 14-tägig
 - Siehe separaten Plan
- 1 SWS Tafelübung:
 - Dienstags 2. Block: 14-tägig (Raum: C13-3.02, Termine: nächste Seite)

ACHTUNG ÄNDERUNG:

Übungsgruppen ÜL-3 und ÜL-4:

Dienstag 17:45 – 19:15 Uhr

ACHTUNG: 10:15 – 11:45 Uhr

Organisatorisches

Organisatorisches: Terminübersicht Vorlesung / Tafelübung

ACHTUNG: 10:15 – 11:45 Uhr

Termin	Vorlesung (Mi, 3. Bl.)	Termin	Tafelüb. (Di, 2. Bl.)	Bemerkung
02. April	Vorlesung	01. April	–	
09. April	Vorlesung	08. April	ÜT-1	
16. April	Vorlesung	15. April	ÜT-2	
23. April	Vorlesung	22. April	ÜT-1	
30. April	Vorlesung	29. April	ÜT-2	
07. Mai	Keine Vorlesung	06. Mai	Keine Veranstalt.	Interdiszipl. Wochen
14. Mai	Keine Vorlesung	13. Mai	Keine Veranstalt.	Interdiszipl. Wochen
21. Mai	Vorlesung	20. Mai	ÜT-1	
28. Mai	Vorlesung	27. Mai	ÜT-2	
04. Juni	Vorlesung	03. Juni	ÜT-1	
11. Juni	Vorlesung	10. Juni	ÜT-2	
18. Juni	Vorlesung	17. Juni	ÜT-1	
25. Juni	Vorlesung	24. Juni	ÜT-2	
02. Juli	Vorlesung	01. Juli	–	

Organisatorisches

Kriterien für Scheinvergabe

Für den Schein müssen sowohl Labor als auch Klausur bestanden sein!

1. Erfolgreiche Klausurteilnahme:

- mindestens 50% der Punkte
- Termin: Semesterende (Termin wird vom Prüfungsamt bekanntgegeben)

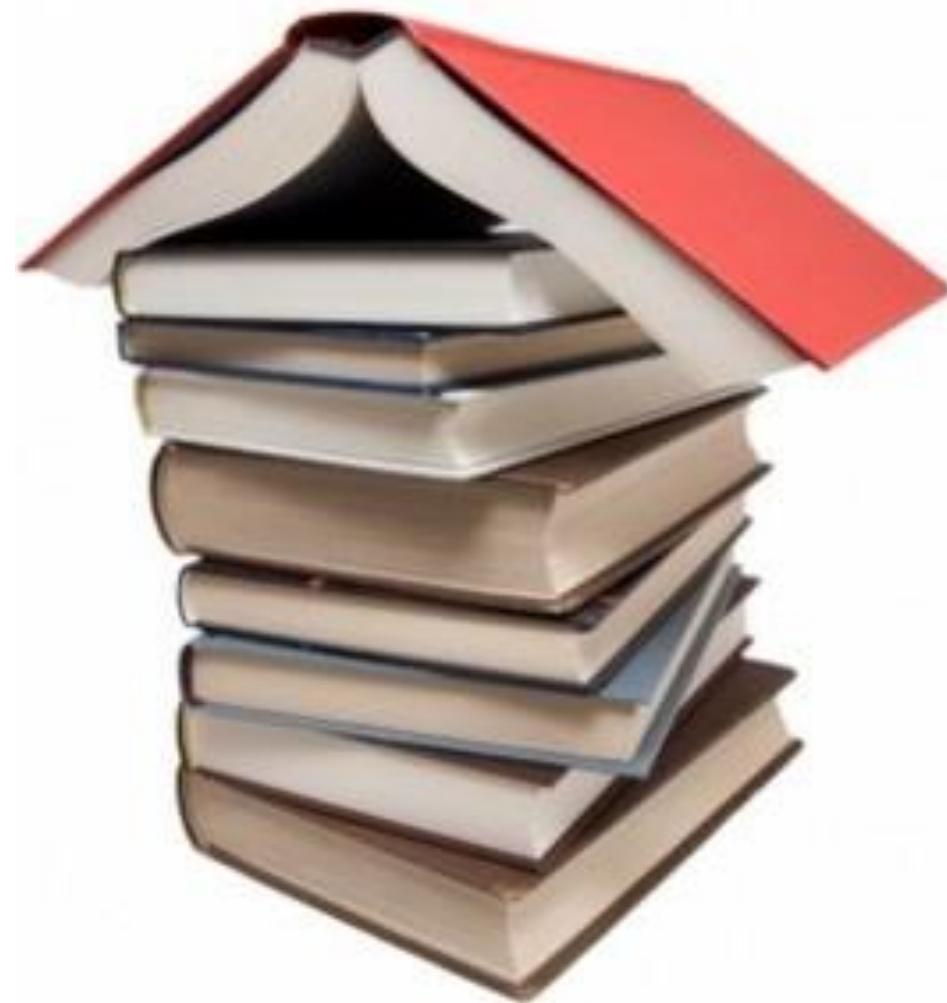
2. Erfolgreiche Laborteilnahme:

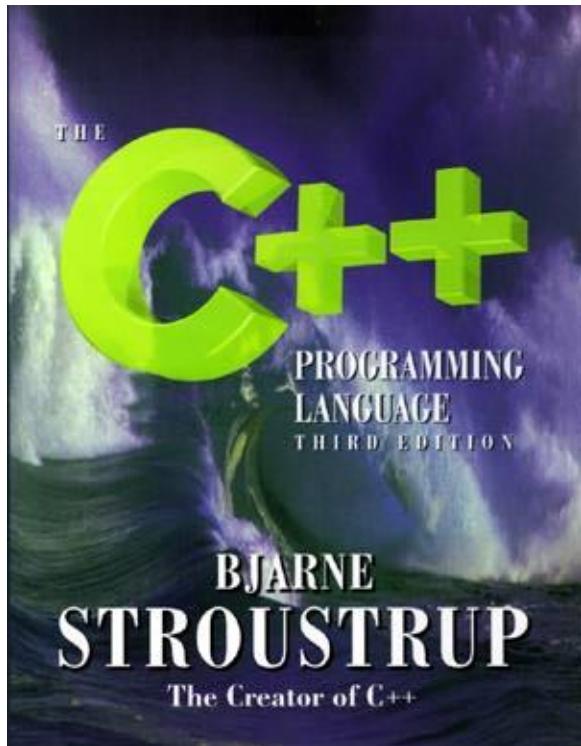
- Anwesenheitspflicht
- Abgabe *aller* Übungszettel (2-er Teams)
- 4 der 5 Aufgabenblätter müssen erfolgreich bearbeitet werden
- (Zweifelsfälle: mündliche Prüfung)
- Keine Noten für das Labor („bestanden“ oder „nicht bestanden“)
- Weitere Einzelheiten: in den Laborgruppen

Weitere Anmerkungen

- Vorlesungsfolien werden im Skripteordner zur Verfügung gestellt:
<ftp://ftp.skripte.fh-kiel.de/m/carsten.meyer/SoSe2014/C++/Vorlesung>
- Aufgabenblätter für das Labor (Hausaufgaben) können Sie herunterladen unter
<ftp://ftp.skripte.fh-kiel.de/m/carsten.meyer/SoSe2014/C++/Labor>
- Bei Fragen:
 - Sprechen Sie mich bitte nach der Vorlesung an bzw. vereinbaren Sie einen Termin
 - (Emails bitte nur, falls wirklich notwendig)
- Bitte beachten Sie das Infosystem für aktuelle Ankündigungen!

Literatur & Co.





- Das Standardwerk
- Für Einsteiger eher ungeeignet
- Deutsche Version ist mangelhaft übersetzt.
- Wer kann, sollte die Originalversion lesen

Bjarne Stroustrup, "Die C++ Programmiersprache", Addison Wesley

Interessant auch: FAQ-Seite von Bjarne Stroustrup:

http://www.research.att.com/~bs/bs_faq.html

Literatur



Objektorientiert in C++. Einstieg und professioneller Einsatz (Entwickler-Press) (Broschiert)

von [Dirk Louis](#) (Autor)

Noch keine Kundenrezensionen vorhanden: [Schreiben Sie die erste!](#)

Preis: EUR 34,90 **Kostenlose Lieferung.** [Siehe Details.](#)

Verfügbarkeit: Auf Lager, Verkauf und Versand durch **Amazon.de.**
Geschenkverpackung verfügbar. Zustellung durch **DHL.**

Nur noch 3 Stück verfügbar -- jetzt bestellen.

Lieferung bis Freitag, 7. März: Bestellen Sie in den nächsten 6 Stunden und 37 Minuten per Overnight-Express. [Siehe Details.](#)

Dirk Louis, "Objektorientiert in C++. Einstieg und professioneller Einsatz ",
Entwickler-Press)

Quelle einiger Programme, die hier besprochen werden.

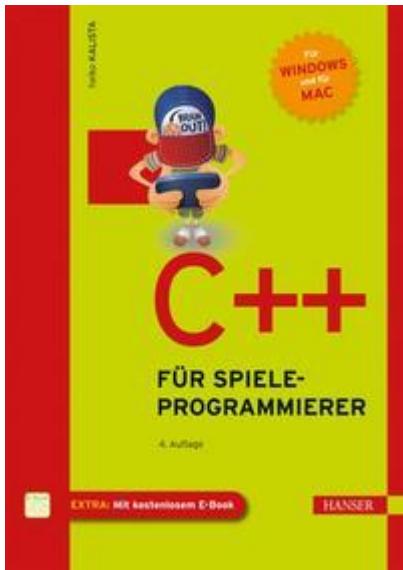
Literatur



Ulrich Breymann,
" C++ : Einführung und
professionelle Programmierung ",
Hanser Fachbuchverlag,
768 Seiten,
E-book ca. 5€ (z. Zt.)

- ausführlich

Literatur

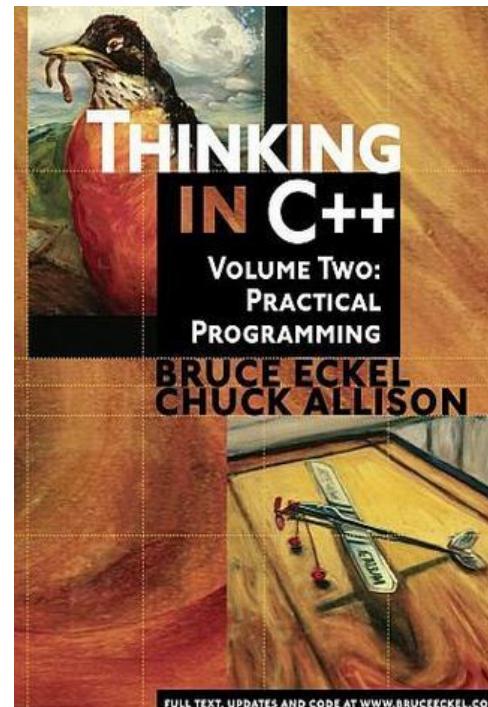
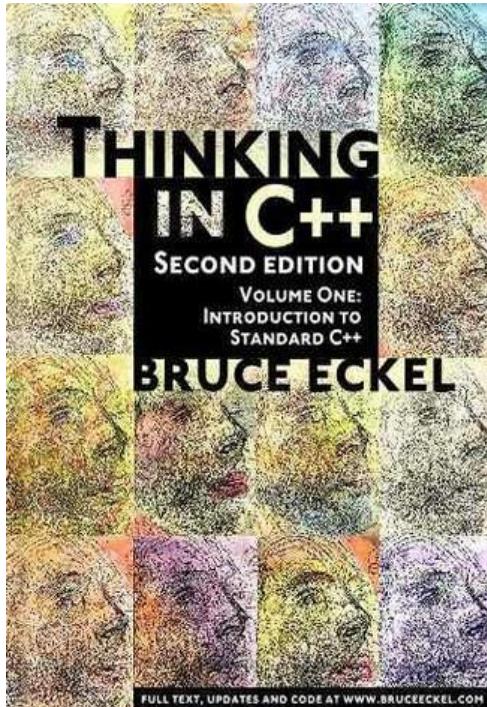


Heiko Kalista,
"C++ für Spieleprogrammierer",
Hanser Fachbuchverlag,
479 Seiten,
ca. 35€

- Einstieg in C++
- C++ für Fortgeschrittene
- Beispiele aus der Spieleentwicklung
- Lockere Sprache

Von Studenten empfohlen.

Literatur



Bruce Eckel: Thinking in C++, Volume 1: Introduction to Standard C++, Prentice Hall, 2000, ISBN 0-13-979809-9, auch als kostenloses online-Buch verfügbar

Bruce Eckel:Thinking in C++, Vol. 2: Practical Programming, Prentice Hall, 2003, ISBN 0-13-035313-2, auch als kostenloses online-Buch verfügbar, Beispiele als Quelltext

Tutorials

www.cplusplus.com

The screenshot shows a Microsoft Internet Explorer window displaying the C++ Language Tutorial from cplusplus.com. The page has a dark blue header with the site's logo and navigation links. A sidebar on the left contains a detailed table of contents for various C++ topics. The main content area features a toolbar simulation at the top, followed by a large heading 'C++ Language Tutorial'. Below the heading, a brief introduction explains the purpose of the tutorials. The page is published by Juan Soulie and last updated on August 6, 2006. The main content is organized into several sections with expandable table of contents:

- Introduction:**
 - Instructions for use
- Basics of C++:**
 - Structure of a program
 - Variables, Data Types.
 - Constants
 - Operators
 - Basic Input/Output
- Control Structures:**
 - Control Structures
 - Functions (I)
 - Functions (II)
- Compound Data Types:**
 - Arrays
 - Character Sequences
 - Pointers
 - Dynamic Memory
 - Data Structures
 - Other Data Types
- Object Oriented Programming:**
 - Classes (I)
 - Classes (II)
 - Friendship and inheritance
 - Polymorphism
- Advanced Concepts:**
 - Templates
 - Namespaces
 - Exceptions
 - Type Casting
 - Preprocessor directives
- C++ Standard Library:**
 - Input/Output with files
- Object Oriented Programming:**
 - Classes (I)
 - Classes (II)
 - Friendship and inheritance
 - Polymorphism
- Advanced Concepts:**
 - Templates

Tutorials

[http://de.wikibooks.org/wiki/C++-Programmierung](http://de.wikibooks.org/wiki/C%2B%2B-Programmierung)

The screenshot shows a Microsoft Internet Explorer window displaying the C++ Programming page from Wikibooks. The title bar reads "C++-Programmierung - Wikibooks - Microsoft Internet Explorer bereitgestellt von HRZ". The address bar shows the URL "http://de.wikibooks.org/wiki/C++-Programmierung". The page content includes a sidebar with navigation links like "Hauptseite", "Aktuelles", and "Buchkatalog". The main content area features a section titled "C++-Programmierung" with a sub-section "Altes Inhaltsverzeichnis" containing a list of topics such as "Der Einstieg", "Fortgeschrittene Themen", and "Werkzeuge". A sidebar on the right contains links for "Bearbeiten" and "Einstellungen". The bottom of the screen shows the Windows taskbar with various open application icons.

Tutorials

Liste von Webseiten, Tutorials, Online-Buechern usw.

- <http://www.desy.de/gna/html/cc/Tutorial/tutorial.html>
- <http://www.cprogramming.com/tutorial.html>
- <http://cplus.kompf.de/tutor.html>
- <http://www.intap.net/~drw/cpp/>
- <http://www.cs.wustl.edu/~schmidt/C++/>
- <http://www.4p8.com/eric.brasseur/cppcen.html>
- <http://www.learn.cpp.com/>
- <http://newdata.box.sk/bx/c/htm/fm.htm>
- <http://www.cpp-tutor.de/cpp/hinweise.html>
- <http://www.cplusplus.com/doc/tutorial/>
- <http://www.gia.rwth-aachen.de/Lehre/Cpp/script/online/index.html>
- <http://www.informit.de/books/c++21/data/inhalt.htm>

Einleitung: Warum OOP?

1. Einleitung

Sir Isaac Newton secretly admitted to some friends: He understood how gravity *behaved*, but not how it *worked!*

Tomlin, „The search for Signs of Intelligent Life in the Universe“

Einleitung

Ein Ziel der Vorlesung: Erlernen der **objektorientierten Programmierung (OOP)**

OOP ist Methode zur effizienten Software-Entwicklung für große Systeme wie z.B.

- industrielle Software
- Forschungssoftware

Eigenschaften großer Software-Systeme:

- Viele Entwickler
- Lange Lebensdauer (ursprüngliche Entwickler sind nicht mehr da)
- Große Komplexität
 - Verständnis aller Details ist für „normalen“ Entwickler praktisch unmöglich
 - DaVincies unter den SW-Entwicklern könnten es vielleicht – aber zu wenige
 - Wir brauchen eine Methode zum Umgang mit Komplexität

OOP ist eine geeignete Methode zur Beherrschung von Software-Komplexität

Software-Komplexität

Software wird immer komplexer (gestiegener Umfang, gestiegene Anforderungen)

Software-Komplexität ist inherent und hat vor allem folgende Ursachen

- Komplexität des gegebenen Problems
- Schwierigkeiten beim Management des Entwicklungsprozesses
- Software-Flexibilität

Software-Komplexität

Komplexität des gegebenen Problems

- **Grosse Zahl konkurrierender (evtl. widersprüchlicher) Anforderungen**
 - Beispiele: autonomer Roboter, Flugzeug, Handy
 - Funktionale Anforderungen: Technische Spezifikation
 - Nicht-funktionale Anforderungen: Benutzbarkeit, Leistung, Kosten, Zuverlässigkeit,...
- **Unklares Anforderungsprofil (Benutzerwünsche)**
 - Benutzer und Entwickler von Softwaresystemen sprechen nicht die gleiche Sprache
 - Beschreibung der Benutzerwünsche schwierig
 - umfangreiche Requirement Spezif. mit Raum für unterschiedliche Auslegungen
- **Anforderungen wechseln während des Entwicklungsprozesses (Moving Target)**

Software-Komplexität

Schwierigkeiten beim Management des Entwicklungsprozesses

- Grösse der Softwarepakete erfordert Team von Entwicklern
- Komplexe Kommunikation und Koordination
 - geographisch verteilte Mitglieder
- Gewährleistung der Einheitlichkeit und Integrität des Designs ist Hauptproblem bei der Softwareentwicklung im Team

Software-Komplexität

Software-Flexibilität

- Kaum „Bauvorschriften“ oder Standards für fundamentale Code-Bestandteile
 - Baubranche: Qualitätsstandards für Rohmaterial, einheitliche Bauvorschriften
- Ultimative Flexibilität von Software erlaubt beliebige Abstraktionen / Designs

Die gestiegene Komplexität von Software führte Mitte der 1960 Jahre zur sog. Software-Krise, die allgemein als Auslöser für die Entwicklung der objektorientierten Sprachen angesehen wird.

Die Software Krise

- Mitte der 60'er Jahre des letzten Jahrhunderts kam es zu einer Software-Krise
- Ursache:
 - Gestiegene Anforderungen an Programme führten zu komplexerer Software.
 - Fehlerhäufigkeit stieg
- Als Lösung wurden neue Programmierkonzepte vorgeschlagen:
„Objektorientierte Programmierung“

Nach den *prozeduralen Programmiersprachen* (z.B. C, Basic, Pascal) wurden nun *objektorientierte Programmiersprachen* entwickelt (z.B. C++, C#, Java)

Prozedurale Programmiersprachen
(z.B. C, Basic, Pascal)

Objektorientierte Programmiersprachen
(z.B. C++, C#, Java)



Prozedurale Programmierung

- Führe einen Schritt aus, dann führe den nächsten Schritt aus...
- Jede Aktion wird mit Hilfe einer Programmiersprache als eine **Anweisung** abgebildet
- Anweisungen werden von oben nach unten abgearbeitet
- Anweisungen, die eine gemeinsame Aufgabe bearbeiten, werden mit Hilfe von **Prozeduren** zusammengefaßt
- Die **Anweisungen sowie die zu bearbeitenden Daten sind getrennt**

Strukturierte Programmierung

- ... ist eine Untergruppe der prozeduralen Programmierung
- Nutzt das Prinzip „teile und herrsche“
- Ein Programm wird baumartig in Teilprogramme zerlegt
- Jedes Programm hat einen definierten Anfang und ein definiertes Ende

„Teile und herrsche“ / „divide and conquer“:

Uralte Technik zur Beherrschung von Komplexität

Übertragen auf komplexe Software Systeme:

- Zerlegung des Systems in immer kleinere Teile, die unabhängig optimiert werden
- Entwickler muss nicht alles auf einmal verstehen, sondern nur Teile

Einführungsbeispiel

Programm für die Verwaltung der FH Bibliothek

Ziel: Verwaltung des Bücherbestandes, d.h.

- Welche Bücher gibt es (Titel, Autor, ISBN)
- Wo steht ein Buch?
- Ist es gerade verliehen – ggf. an wen?

Vorgehen in **prozeduraler** Programmierung:

1. Geeignete Datenstruktur entwerfen
2. Benötigte Funktionen schreiben, die diese Daten verändern

Einleitung: Warum OOP?

Einführungsbeispiel in prozeduraler Programmierung

I. Datenstruktur entwerfen

```
struct buch {  
    char title[50];  
    char autor[50];  
    char isbn[14];  
    char ausleihertyp[40];  
    char ausleihdatum[11];  
    long regalplatz;  
}
```

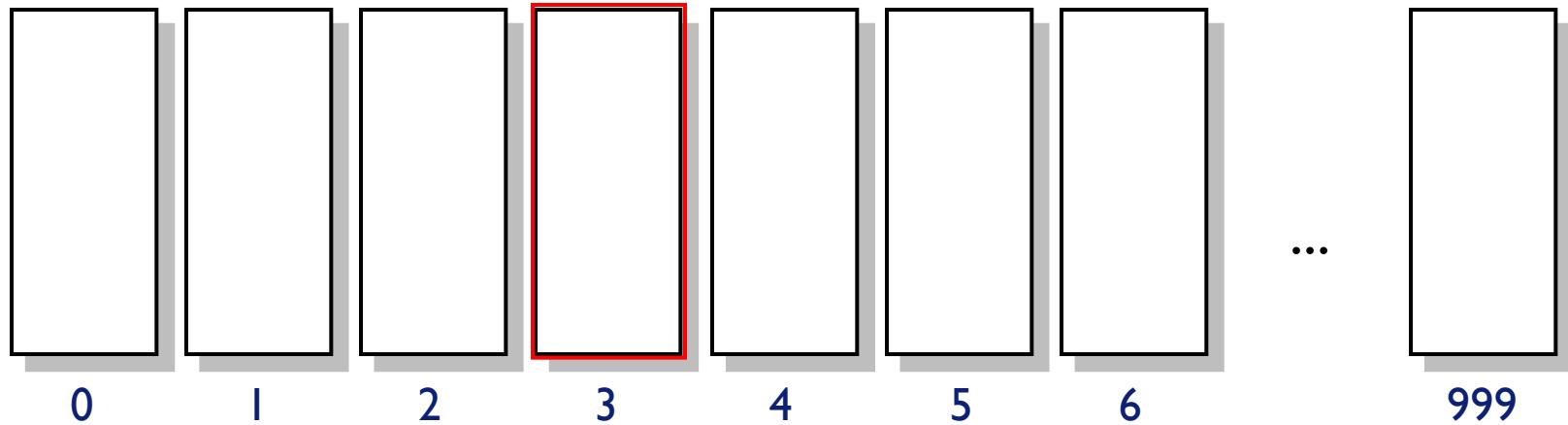
Speicherplatz für z.B. 1000 Bücher bereitstellen:

```
struct buch bucher[1000];
```

Einleitung: Warum OOP?

Einführungsbeispiel in prozeduraler Programmierung

→ (Leerer) Speicherplatz für Informationen über 1000 Bücher steht bereit

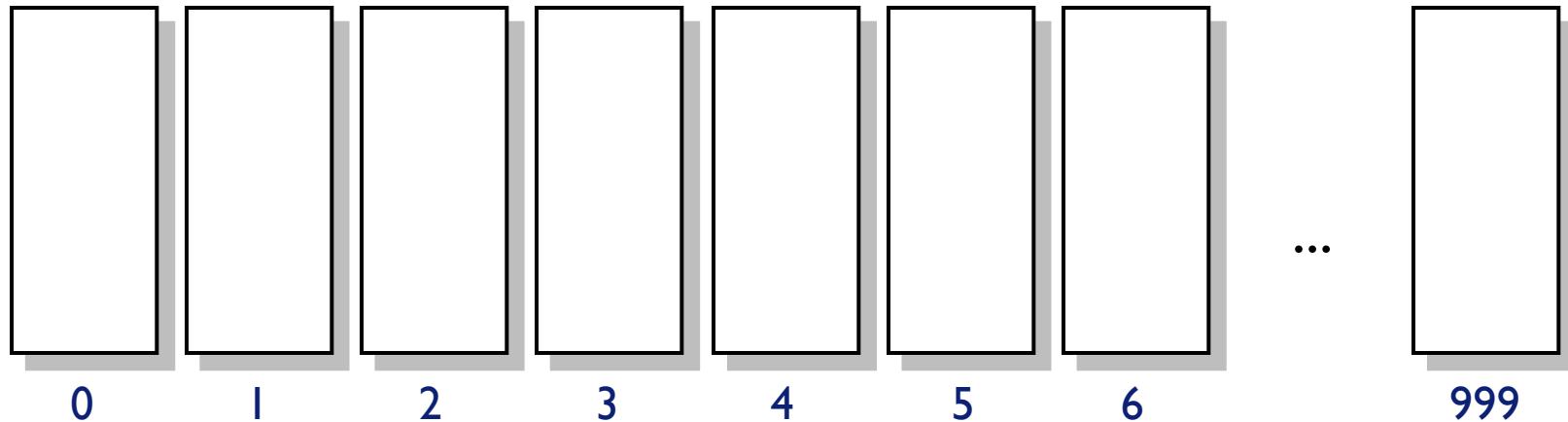


Zugriff auf ein bestimmtes Buch mit Hilfe des Index: `buecher[3]`

Einleitung: Warum OOP?

Einführungsbeispiel in prozeduraler Programmierung

→ (Leerer) Speicherplatz für Informationen über 1000 Bücher steht bereit



Jetzt muss dieser Speicherplatz mit Leben (d.h. Information) gefüllt werden

- Aufnahme neuer Bücher
- Löschen von Büchern
- Ausleihen von Büchern
- Zurücknahme von Büchern

→ Funktionen

Einführungsbeispiel in prozeduraler Programmierung

2. Funktionen entwickeln

Benötigte Funktionen:

- Neues Buch aufnehmen : buchEingeben (int index)
- Buch löschen : buchLoeschen (int index)
- Buch ausleihen : buchAusleihen (int index, char* ausleiher)
- Buch zurücknehmen : buchZurueck (int index)

Funktionen operieren auf der Datenstruktur, um den Inhalt zu manipulieren

Wichtig:

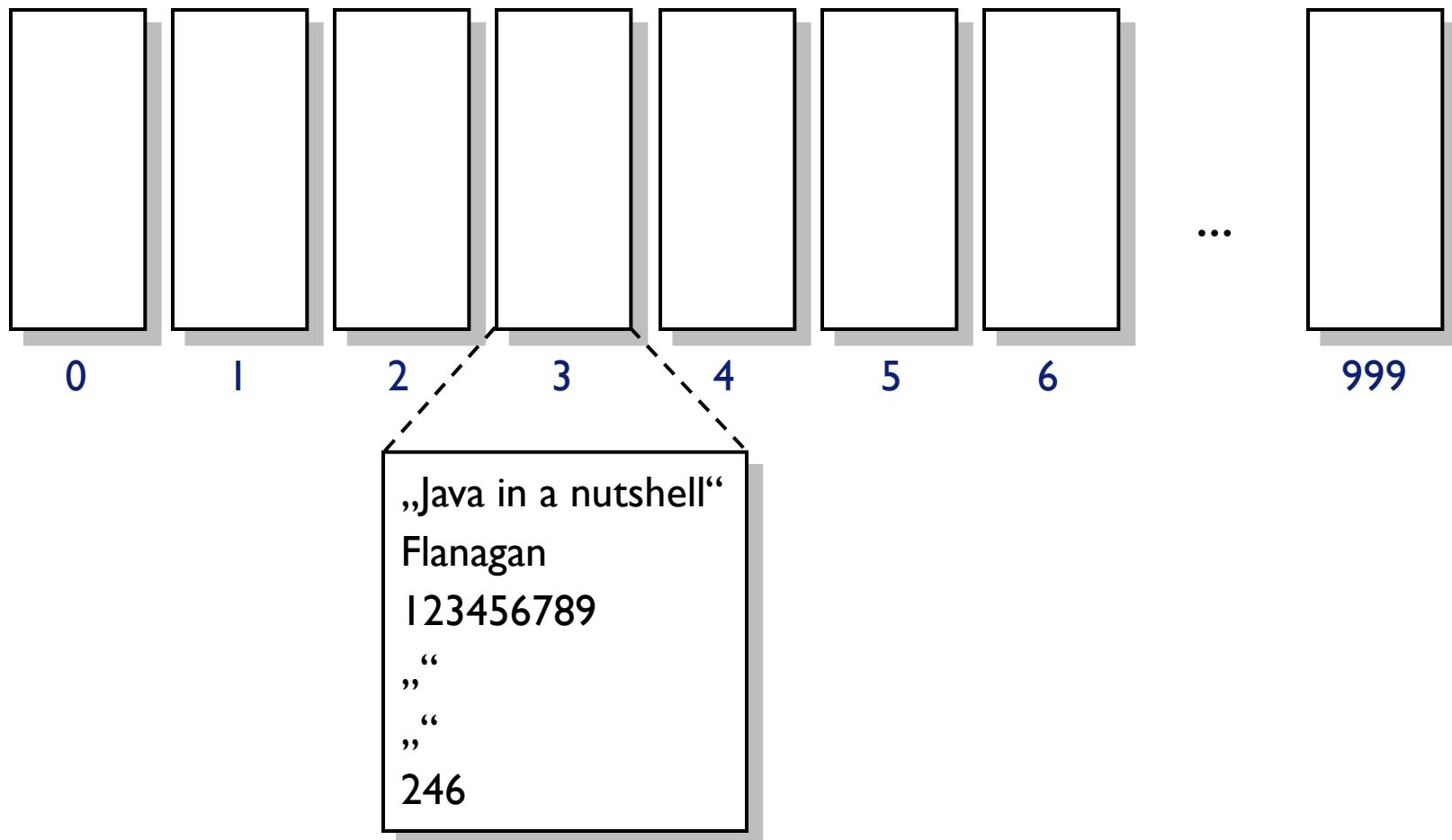
- Es gibt **keine feste Bindung** zwischen den Daten und den Funktionen
- Beziehung zwischen einer Funktion und einem Buch wird über Index hergestellt

Einleitung: Warum OOP?

Einführungsbeispiel in prozeduraler Programmierung

2. Funktionen entwickeln

Beispiel: buchEingeben (3)



Einführungsbeispiel in prozeduraler Programmierung: Probleme (I)

I.) Erweiterung der Datenstruktur

Was passiert im Falle einer benötigten Erweiterung der Datenstruktur?

```
struct buch {  
    char title[50];  
    char autor[50];  
    char isbn[14];  
    char ausleiher[40];  
    char ausleihdatum[11];  
    long regalplatz;  
    char erscheinungsjahr[20];  
}
```

Einführungsbeispiel in prozeduraler Programmierung: Probleme (I)

In realer Software würden viele Funktionen im Bibliotheks-Programm mit der Struktur buch arbeiten (möglich sogar: mehrere Funktionen machen das gleiche).

→ Anpassung **jeder** dieser Funktionen notwendig

Daten und Funktionen liegen getrennt voneinander, d.h. Funktionen können beliebig im Quelltext verstreut liegen

→ Allein die Suche nach zu ändernden Stellen kann schon sehr aufwendig sein

Bei steigender Komplexität der Software wird es schwierig zu überblicken,

- wer gerade welche Datenstrukturen verändert und
- welche Programmteile mit welchen Datenstrukturen arbeiten

→ Chaos aus im Programm verstreuten Daten und Funktionen ist wahrscheinlich

Einleitung: Warum OOP?

Einführungsbeispiel in prozeduraler Programmierung: Probleme (2)

2.) Einführung weiterer, spezialisierter Datenstrukturen

```
struct Fachbuch {  
    char title[50];  
    char autor[50];  
    char isbn[14];  
    char ausleiher[40];  
    char ausleihdatum[11];  
    long regalplatz;  
    char erscheinungsjahr[20];  
    int abteilung;  
}
```

```
struct Roman {  
    char title[50];  
    char autor[50];  
    char isbn[14];  
    char ausleiher[40];  
    char ausleihdatum[11];  
    long regalplatz;  
    char erscheinungsjahr[20];  
    char genre[40];  
    char sprache[40];  
}
```

Einleitung: Warum OOP?

Einführungsbeispiel in prozeduraler Programmierung: Probleme (2)

Lösungsmöglichkeit I: Jeweils eine neue, eigene Struktur für Fachbuch und Roman

- Vorhandene Funktionalität muß neu geschrieben werden, Code-Duplizierung, natürliche Beziehung zwischen Buch und Fachbuch / Roman wird nicht genutzt

Lösungsmöglichkeit 2: Eigene Struktur für Fachbuch und Roman unter Verwendung der Struktur „Buch“

```
struct Fachbuch {  
    struct Buch buchInfo;  
    int abteilung;  
}
```

```
struct Roman {  
    struct Buch buchInfo;  
    char genre[40];  
    char sprache[40];  
}
```

- nach wie vor keine einheitliche Behandlung der drei verwandten Datenstrukturen

Objektorientierte Programmierung (OOP)

- ... bietet eine abstrahierte Abbildung realer, interagierender Objekte in einer Programmiersprache
- Daten und zugehörige Funktionen werden zu einer **Einheit** zusammengefaßt:
 - **Klasse** (übergeordneter Begriff): benutzerdefinierter **Typ** aus Daten & Funktionen
 - **Objekt** (konkrete Realisierung): Instanz / Exemplar einer Klasse
- Eigenschaften des objektorientierten Ansatzes
 - **Kapselung**
 - **Vererbung**
 - **Polymorphie (Vielgestaltigkeit)**

Objektorientierte Programmierung: Objekte

Warum wird das Objekt als fundamentaler Baustein verwendet?

- Systeme in der realen Welt bestehen aus miteinander interagierenden Objekten

Beispiel: System Bank

Objekte: Kunden, Angestellte, Konten, Direktor, Geldautomat, ...

Interaktion:

- Kunde eröffnet Konto
- Kunde zahlt Geld auf sein Konto ein
- Angestellter zahlt Geld vom Konto an Kunden aus

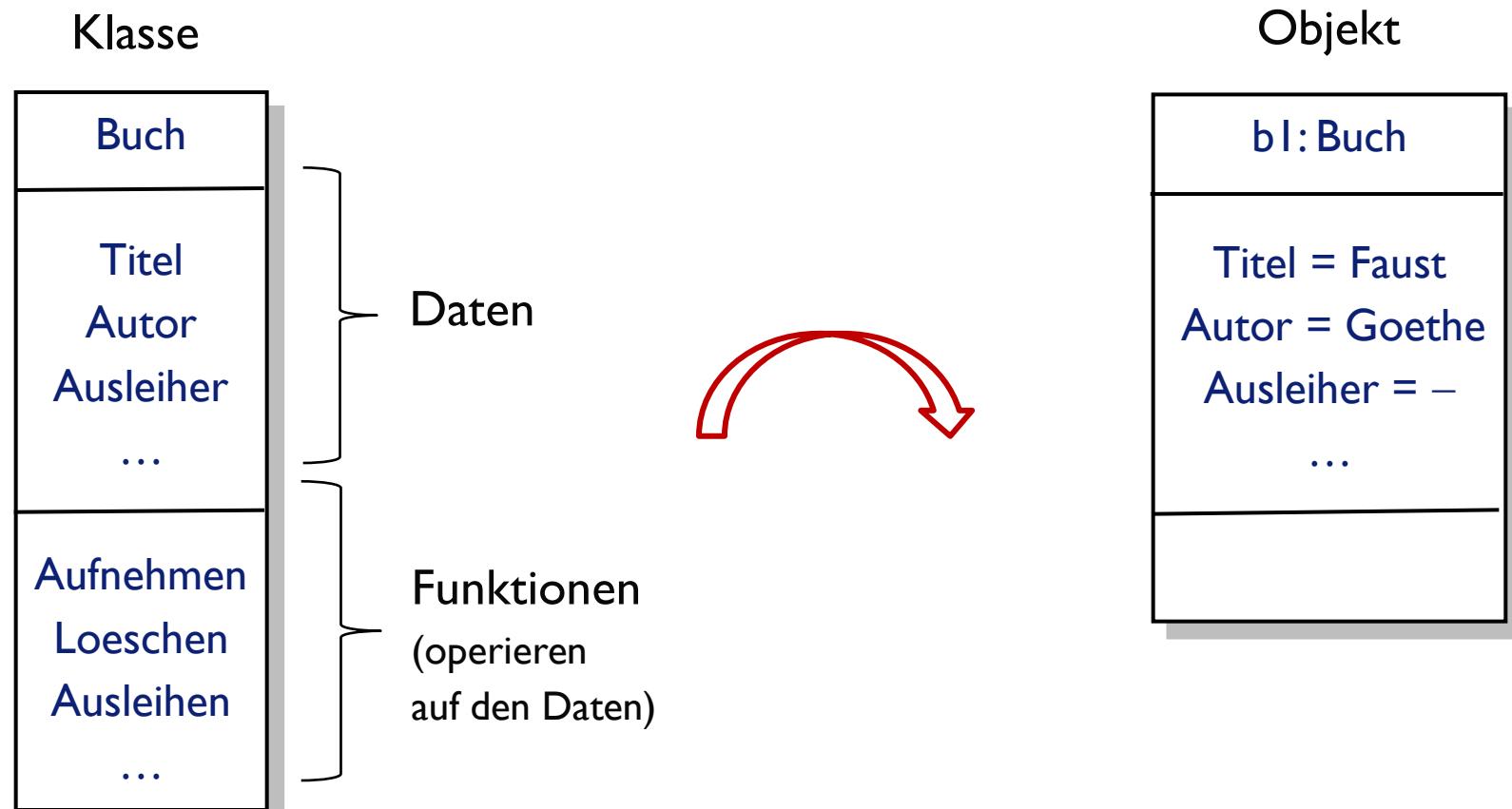
- Zum Objekt gehören
 - **Merkmale, Eigenschaften → Daten / Variablen:** z.B. Zinssatz, Gehalt ...
 - **Verhaltensweisen → Methoden / Funktionen:** z.B. einzahlen, überweisen ...
- Software soll solche realen Systeme abbilden

Einleitung: Warum OOP?

Einführungsbeispiel in objektorientierter Programmierung

- Beispiel: Verwaltung der FH Bibliothek

→ Einführung einer Klasse „Buch“, in dem Daten (Eigenschaften, Attribute) und zugehörige Funktionen (Methoden) zusammengefaßt sind



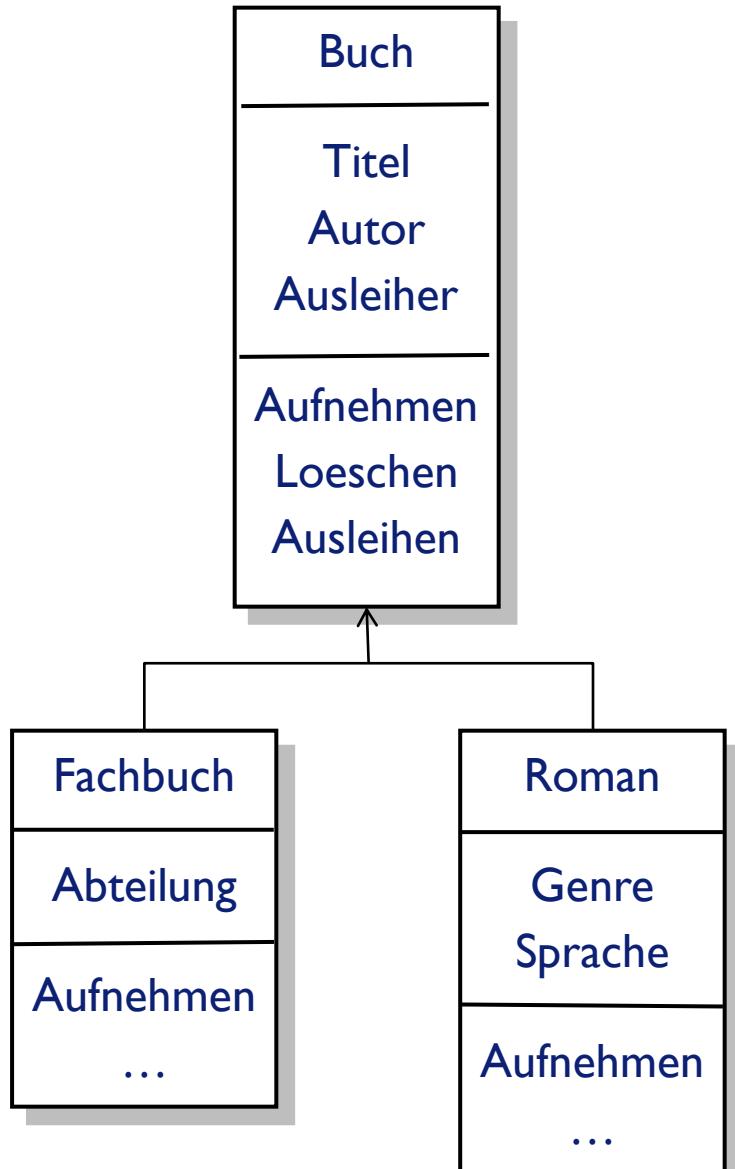
Einführungsbeispiel in objektorientierter Programmierung

Vorteile für die Entwickler in diesem Beispiel

- **Funktionen operieren direkt auf den Daten:**
 - b1.Ausleihen: ruft Funktion „Ausleihen“ mit den Daten von Buch b1 auf
- **Leichte Änderbarkeit:**
 - Bei Änderung der Datenstruktur (zusätzlicher Eintrag „Erscheinungsjahr“) sind die betroffenen Funktionen direkt ersichtlich und können ggf. leicht erweitert werden
 - Solange sichergestellt ist, dass sich das Objekt „nach aussen“ gleich verhält, müssen keine weiteren Stellen im Programmcode verändert werden
- **Leichte Einführung weiterer, spezialisierter Datenstrukturen:**
 - Über das Konzept der **Vererbung**
 - Einführung von „abgeleiteten Klassen“ Fachbuch und Roman
 - „erben“ Datenelemente und Funktionen der Klasse Buch (direkt verfügbar)
 - Nur weitergehende Daten und Funktionen müssen implementiert werden

Einleitung: Warum OOP?

Objektorientierte Programmierung: Vererbung und Polymorphie



Vererbung:

- Direkte Verfügbarkeit von Daten und Methoden der Klasse `Buch` in den abgeleiteten „`Fachbuch`“ und „`Roman`“

Polymorphie:

- Unterschiedliche Implementierung einer Funktion in den jeweiligen Klassen
- Bsp: Funktion „`Aufnehmen`“ in Klasse „`Fachbuch`“ setzt auch die Abteilung und in „`Roman`“ auch Genre / Sprache
- Beim Funktionsaufruf wird die passende Funktion (anhand des Objekttyps) ausgewählt

Objektorientierte Programmierung: Kapselung

- Trennung von interner Implementierung und äußerer Schnittstelle
- Analogie: Automotor
 - Fahrer muß nicht wissen, wie der Motor im Einzelnen funktioniert
 - Motor: „Black Box“; Eigenschaften / Funktionen des Motors sind gekapselt
 - Mit Hilfe von Funktionen (Schnittstellen): Veränderung der Eigenschaften
 - Bsp: durch Bremsen wird die Eigenschaft „Geschwindigkeit“ verändert
- Objektorientierte Programmierung:
 - Genau definierte Schnittstellen nach außen
 - Verbergen der Implementierungsdetails
- Bsp: Klasse Buch:
 - Eigenschaften „Standort“, „Preis“ nach außen nicht sichtbar (Implementierung)
 - Schnittstelle zur Benutzung: Funktionen wie „Anzeigen“, „Ausleihen“ etc.

Vorteile der objektorientierten Programmierung

- Die objektorientierte Betrachtungsweise ist natürlich und intuitiv erfassbar
Identifikation und Beschreibung beteiligter Objekte für gegebenes System ist i.d.R. einfach
- Kapselung, klare Schnittstellen: Anwender von Objekten müssen nur dessen Schnittstelle kennen und keine internen Details der Implementierung
Beispiel Motor
- Bessere Modularität, bessere Strukturierbarkeit besonders bei großen Systemen
Klassen: abgeschlossene Module, ermöglichen leichte Modularisierung
- Leichte Wartbarkeit
Kopplung von Funktion und Daten
- Leichte Erweiterung / Weiterentwicklung, hohes Mass an Wiederverwendbarkeit
der Objekte in anderen Programmteilen
Negativbeispiel Millennium-Bug:
Tausendfache Korrektur ähnlicher Routinen zur Datumsberechnung und -formatierung

Grundbegriffe der OOP

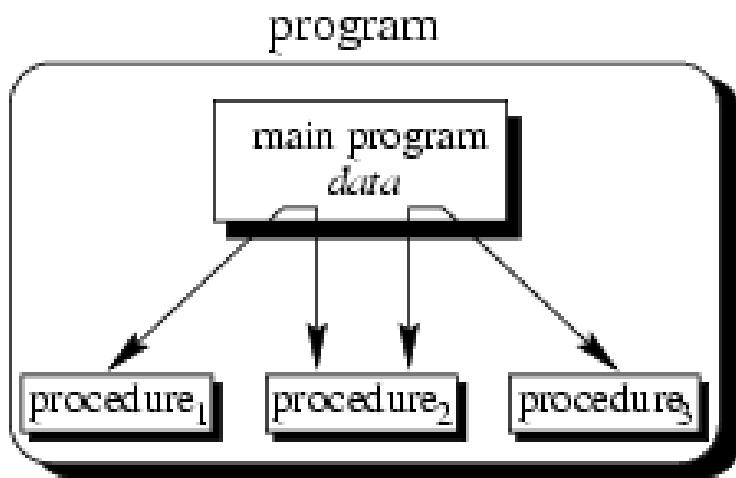
Grundbegriffe der OOP

- Objekte und Klassen
- Geheimnisprinzip und Kapselung
- Vererbung
- Polymorphie

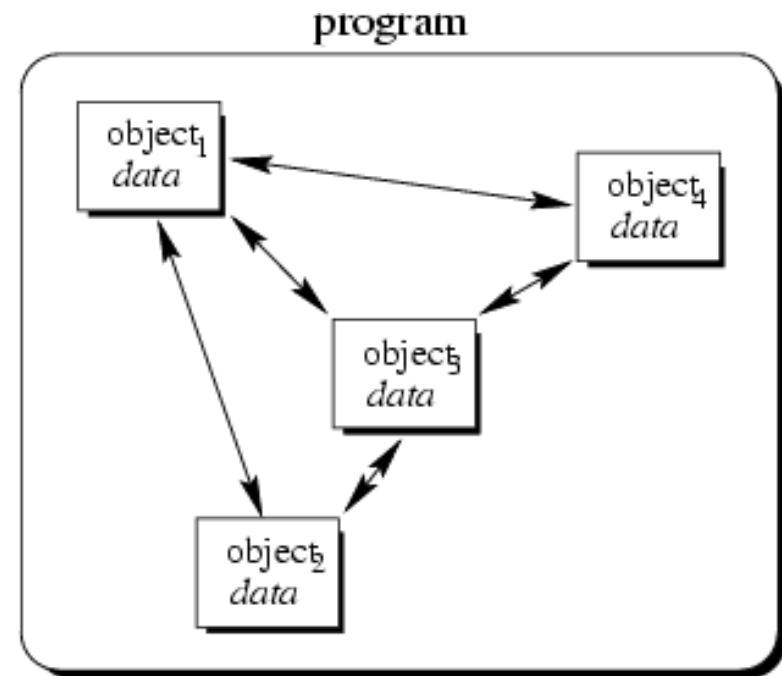
Grundbegriffe der OOP

Prozedurale und objektorientierte Programmierung

Prozedurale Programmierung



Objektorientierte Programmierung



- Hauptprogramm koordiniert Funktionsaufrufe
- und übergibt Daten als Parameter

- Hauptprogramm übergibt Kontrolle an erstes Objekt
- Programmobjekte interagieren durch Austausch von Nachrichten

From: <http://www.tiem.utk.edu/~gross/c++man/tutorial.html>

OOP-Definition

OOP ist eine Implementationsmethode, in der Programme als eine kooperative Sammlung von Objekten organisiert sind, welche Exemplare von Klassen repräsentieren, die wiederum Mitglieder einer Klassenhierarchie mit Vererbungsbeziehungen sind.

Grady Booch, Object-Oriented Analysis And Design

1. OOP verwendet Objekte (keine Algorithmen) als fundamentale Bausteine
2. Jedes Objekt ist ein Exemplar (oder eine Realisierung) einer Klasse
3. Klassen sind durch Vererbungsbeziehungen miteinander verbunden

Wenn eines dieser Elemente fehlt, ist ein Programm nicht objektorientiert!

OOP-Definition

Kurzbeschreibung der OOP nach Alan Kay (*Erfinder der Sprache Smalltalk*)

- I. Alles ist ein Objekt** (anders gesagt: Die Welt besteht aus Objekten)
- 2. Objekte kommunizieren über Nachrichtenaustausch**
- 3. Objekte haben ihren eigenen Speicher**
- 4. Jedes Objekt ist ein Exemplar einer Klasse**
- 5. Die Klasse modelliert das gemeinsame Verhalten ihrer Objekte**
- 6. Ein Programm wird ausgeführt, indem dem ersten Objekt die Kontrolle übergeben wird und der Rest als dessen Nachricht behandelt wird.**

Objektorientierte Programmierung

- ... abstrahiert **Gegenstände** der realen Welt, um sie in einem Programm abzubilden.
 - ... versucht **Daten**, die ein Objekt beschreiben, **und Funktionen**, die die Daten verändern, **in einer Struktur** zusammenzufassen.
 - Objekt: Fundamentaler Baustein der realen Welt
 - Aufgaben in der realen Welt haben i.a. mit Objekten zu tun
 - Für objektorientierte Aufgabenstellungen sollten auch objektorientierte (Software)Lösungen gefunden werden!
- **Objektorientierte Programmierung**

Objekte

- ... beschreiben einen Gegenstand, Person etc. aus der realen Welt
 - Physikalische Einheit: Lastwagen, Baum, Haus, Salat, Apfel, Kunde, ...
 - Konzeptuelle Einheit: Chemischer Prozess, Geschäftsprozess, ...
 - Softwareeinheit: Liste, Datenbank, User-Interface, ...
- Haben einen Zustand: Objekteigenschaften
 - Realisierung: Attribute, die die Eigenschaften charakterisieren
- Haben ein Verhalten: Reaktions- oder Interaktionsmöglichkeiten des Objekts
 - Realisierung: Methoden (die einer Objektklasse zugeordneten Algorithmen)
- Haben eine Identität: „individuelle Ausprägung“ des Objekts
 - Realisierung: eindeutiger Speicherort



Objekte

Objekt
Zustand
Verhalten
Identität

- Attribute, die die Eigenschaften charakterisieren
- Methoden (die einer Objektklasse zugeordneten Algorithmen)
- eindeutiger Speicherort

Ein Objekt ist ein Konzept, eine Abstraktion oder eine Entität mit klaren Grenzen und eindeutiger Bedeutung in einem gegebenen Kontext.

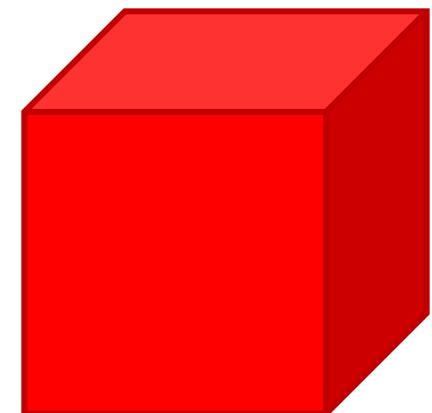
Ein Objekt hat einen Zustand, ein Verhalten und eine Identität.

Ein Objekt wird durch seinen Zustand und die Manipulationsmöglichkeiten auf diesen Zustand (Verhalten) charakterisiert.

Objekte: Beispiel

Ein Würfel wird beschrieben:

- Der Würfel ist rot.
 - Der Würfel hat eine Kantenlänge von 5cm.
 - Der Würfel kann gedreht werden.
 - Der Würfel kann neu eingefärbt werden.
 - Es ist der Würfel von Hans.
- } Zustand / Attribute
- } Verhalten / Methoden
- Identität



Grundbegriffe der OOP

Objekte: Beispiel

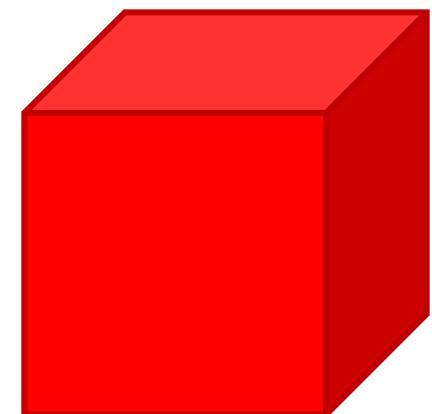


Zustand / Attribute



Verhalten / Methoden

Identität



Quelle: RRZN

Zustand eines Objektes

- Zustand eines Objekts repräsentiert gegenwärtige Eigenschaften
- Zustand eines Objekts wird durch **Attribute** charakterisiert
 - Objekt Rennwagen
Attribute: Geschwindigkeit, Motordrehzahl, Fahrtrichtung, Füllstand Benzin, ...
 - Objekt Politiker
Attribute: Parteizugehörigkeit, Ämter, Gehalt, ...
- Der Zustand eines Objekts kann (und wird) sich über die Zeit verändern
 - Objekt Mitarbeiter
Status: Sachbearbeiter → Status: Gruppenleiter
 - Objekt Oskar Lafontaine
Parteizugehörigkeit: SPD → Parteizugehörigkeit: Die Linke
 - Objekt Erde
Durchschnittstemperatur ...
- Realisierung: Wert der Attributvariablen und Beziehung zu anderen Objekten
Beispiel: Ist-mein-Freund Beziehung
Objekt mit vielen Freund-Beziehungen ist in anderem Zustand als Objekt ohne

Verhalten eines Objektes

- Verhalten bestimmt, wie ein Objekt agiert und reagiert
Sozusagen auf „Umwelteinflüsse“ – also auf andere Objekte in seiner Umwelt
- Verhalten eines Objekts wird durch **Methoden** charakterisiert
 - Methoden: die einer Klasse von Objekten zugeordneten Algorithmen
- Verhalten bestimmt, wie ein Objekt auf Aufrufe anderer Objekte reagiert
 - Auto: Gaspedal treten ändert die Geschwindigkeit (oder auch nicht...)
 - Politische Partei: Wie reagiert sie auf den Erhalt einer Spende?
 - OOP-Sprache (C++): Wirkung der Methoden auf die Werteverteilung der Attribute
- Sichtbares Objektverhalten wird festgelegt durch Menge und Art der Methoden
 - Entspricht der Menge der Operationen, die das Objekt ausführen kann
 - Beispiel: Wasserhahn mit 2 Drehknöpfen für kaltes und warmes Wasser
 - OOP-Sprache (C++): Menge der Methoden, die ein Objekt zur Verfügung stellt

Verhalten eines Objektes: Methoden und Botschaften

- Methode: Funktion / Algorithmus, die / der zu einer Objektklasse gehört
- Botschaft: besteht aus dem Namen der Methode und evt. Argumenten
- Objekte verständigen sich über „Botschaften“
- Objekt A sendet eine Botschaft an Objekt B
 - Bei Objekt B wird zugehörige Methode ausgelöst
 - Methode kann Antwort als Rückgabewert an Objekt A senden
- Eine Methode sollte eine einfache, zusammenhängende Funktion ausführen
 - addStudent statt addStudentAndSortList

Die wichtigsten Methoden in der OOP sind:

- Konstruktoren: Erzeugen ein Objekt
- Destruktoren: Zerstören ein Objekt
- Mutatoren: Verändern den Wert eines Attributes
- Accessoren: Gewähren Zugriff auf Attribut(e)

Identität eines Objektes

Jedes Objekt hat eine eindeutige Identität, auch wenn es einen identischen Zustand wie ein anderes Objekt hat.

Man unterscheidet zwischen Gleichheit von Objekten und Objektidentität.

Zwei Objekte sind gleich (nicht identisch!), wenn sie gleiche Attributwerte haben.

Beispiel:

Objekt **Lehrer**, mit Attributen „Nachname“ und „Fächer“

- Wieviele Lehrer „Meier“ mit den Fächern „Mathe“ und „Physik“ gibt es?
- Jeder hat eigene Identität trotz Gleichheit der Attribute.

Realisierung in C++:

eindeutiger Speicherort

Klassen

... sind Baupläne / Schablonen für die Beschreibung und Erzeugung von Objekten.

Eine Klasse definiert für eine Sammlung von „gleichartigen“ Objekten deren

- Eigenschaften
→ Attribute
- Verhalten
→ Botschaften, auf die die Objekte der Klasse reagieren können
- Beziehungen zu anderen Objekten
→ Assoziationen und Vererbungsbeziehungen

Die Klassenbeschreibung legt fest, welche Attribute (Name und Wertebereich) und welches Verhalten die zugehörigen Objekte besitzen.

Klassen besitzen einen Mechanismus zur Erzeugung neuer Objekte (*object factory*).

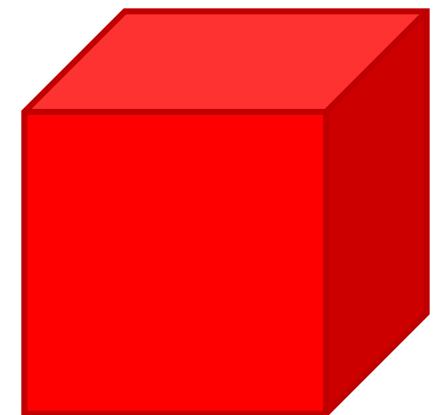
Klassen: Beispiel

Welche Eigenschaften hat jeder Würfel?

- Farbe
- Kantenlänge

Welche Methoden hat jeder Würfel?

- Drehen
- Einfärben
- Zeichnen



Quelle: RRZN

Klassen: Abstraktion

Eine Klasse ist eine **Abstraktion** indem sie

- wichtige (für ihre spezielle Aufgabe) Eigenschaften betont
- andere Charakteristiken unterdrückt

Beispiel:

Lebewesen sind extrem kompliziert → müssen zur Verarbeitung vereinfacht werden

Pferd	→	Arbeitspferd (Haferaufnahme, Stallfläche, Zugkraft)
Mensch	→	Mitarbeiter (Gehalt, Arbeitsmittel, Arbeitsleistung)

- Man kann und muss ein Pferd nicht in seiner vollen Komplexität beschreiben.
- Stattdessen kann man das Pferd auf seine wesentlichen Merkmale (die für das Problem relevant sind) abstrahieren.

Abstraktion: wichtiges Werkzeuge im Umgang mit Komplexität

Eine **Abstraktion** ist

- eine Reduktion auf das Wesentliche
- immer bezogen auf die Sicht des Betrachters

Beispiel:

Katze aus Sicht einer Oma, die Gesellschaft haben möchte

- Schnurrt beim Streicheln
- Frisst Katzenfutter, trinkt Milch und Wasser
- Sonstiges: Sauberes Katzenklo und hundefreies Zuhause

Katze ist viel komplizierter aber das ist unserer Oma egal.

Eine Tierärztin abstrahiert ganz anders

- Herz-Kreislauf System
- Magen-Darm Trakt
- Knochen ...

Beziehung zwischen Klassen und Objekten

Klasse:

- Allgemeine Beschreibung eines oder mehrerer Objekte mit übereinstimmenden Attributen (Eigenschaften) und Methoden (Funktionalität)
- Konstruktionsplan / Schablone für Objekte gleicher Art

Objekt:

- Abbildung eines konkreten Gebildes der Realität; wird aus Klasse erzeugt
- **Instanz einer Klasse**, d.h. ein konkretes Exemplar / Realisierung einer Klasse

Beachte:

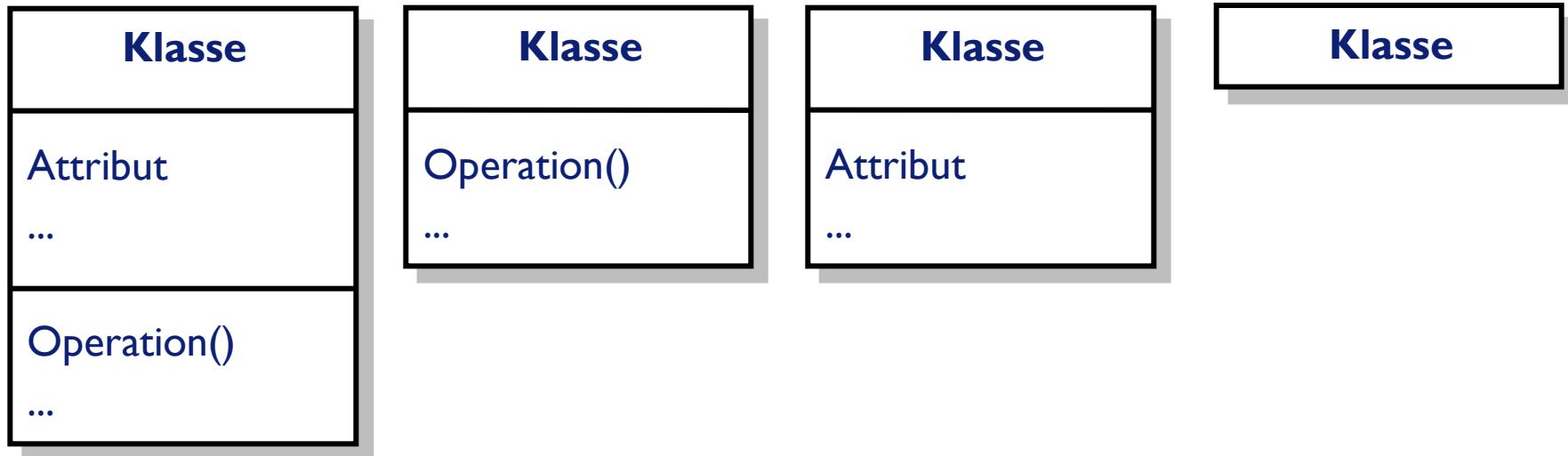
- Eine Klasse ist praktisch immer statisch. Existenz, Bedeutung und Beziehung zu anderen Klassen ist vor der Ausführung eines Programmes festgelegt.
- Objekte werden während der Programm-Ausführung typischerweise in großer Zahl erzeugt und vernichtet

Erzeugung eines Objektes aus einer Klasse:

- Analogie: Fertigung eines Autos aus Konstruktionsplan eines Fahrzeugtyps

Darstellung einer Klasse

UML (*Unified Modeling Language*)



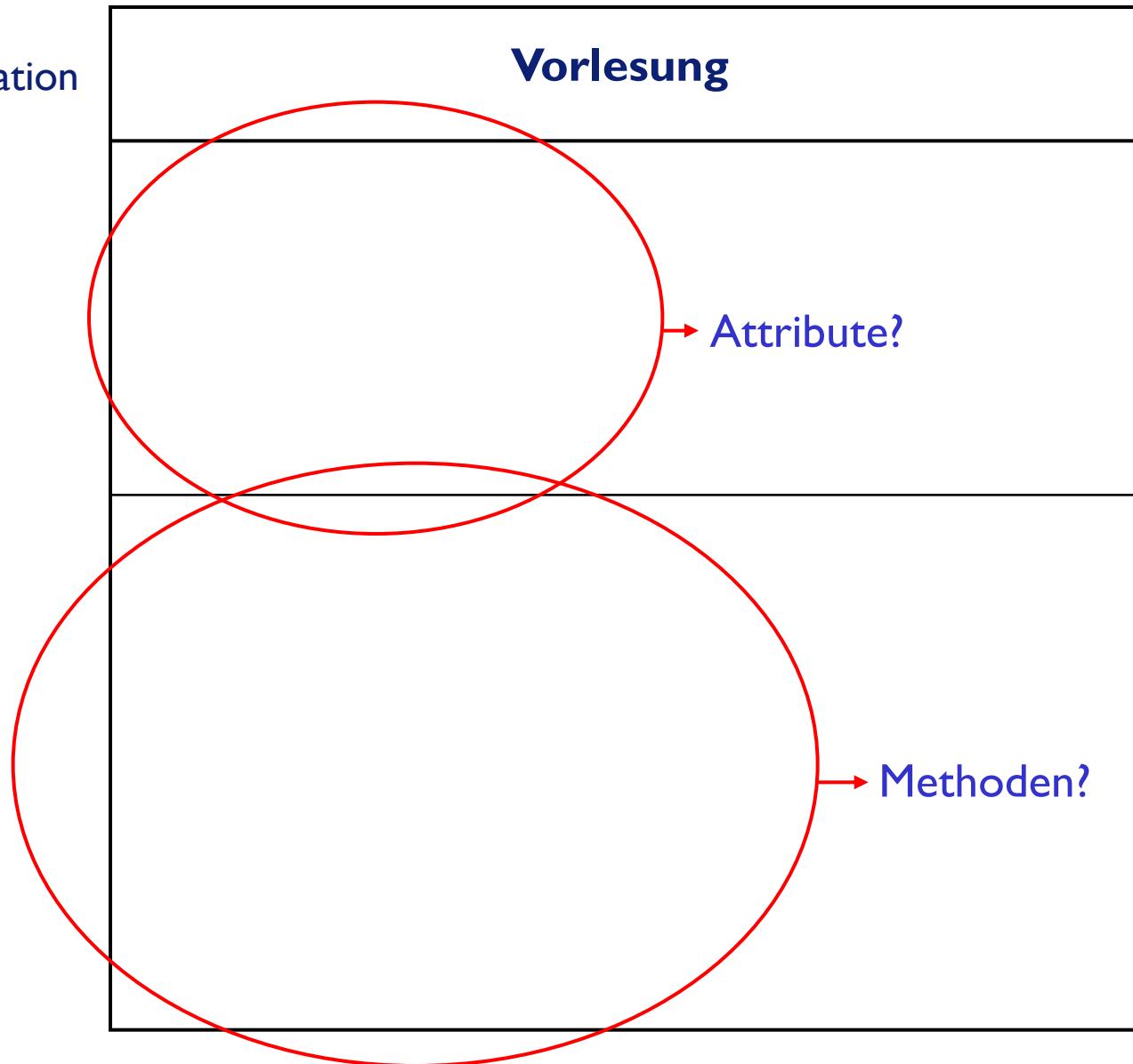
Bemerkungen

- Kurzformen werden verwendet, wenn fehlende Details unwichtig oder woanders definiert sind.
- Der Klassenname wird immer **fettgedruckt** und zentriert dargestellt und beginnt mit einem Grossbuchstaben.

Grundbegriffe der OOP

Beispielklasse

UML Notation



Verwaltung einer
Vorlesung

Beispielklasse

Vorlesung

Name (der Vorlesung)

Hörsaal

Termin

Teilnehmerliste

Füge einen Studenten hinzu

Entferne einen Studenten

Ausgabe des Vorlesungsinhaltes

Ist die Vorlesung voll?

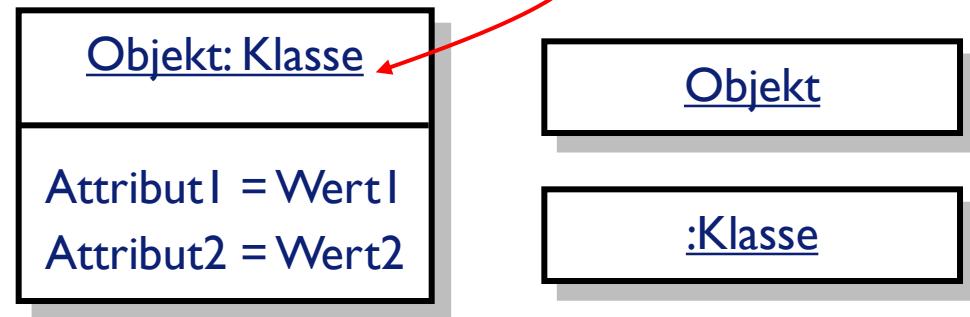
Grundbegriffe der OOP

Darstellung eines Objekts

Graphische Darstellung der OOP-Zusammenhänge:

Darstellung eines Objekts
in der UML:

UML (*Unified Modeling Language*)



:Klasse

Anonymes Objekt der Klasse, d.h. „irgendein“ Objekt -
kein bestimmtes. Daher wird kein Objektname angegeben.

Objekt: Klasse

Objekt mit dem Namen „Objekt“ der Klasse „Klasse“

Objekt

Name der Klasse ist aus dem Kontext ersichtlich

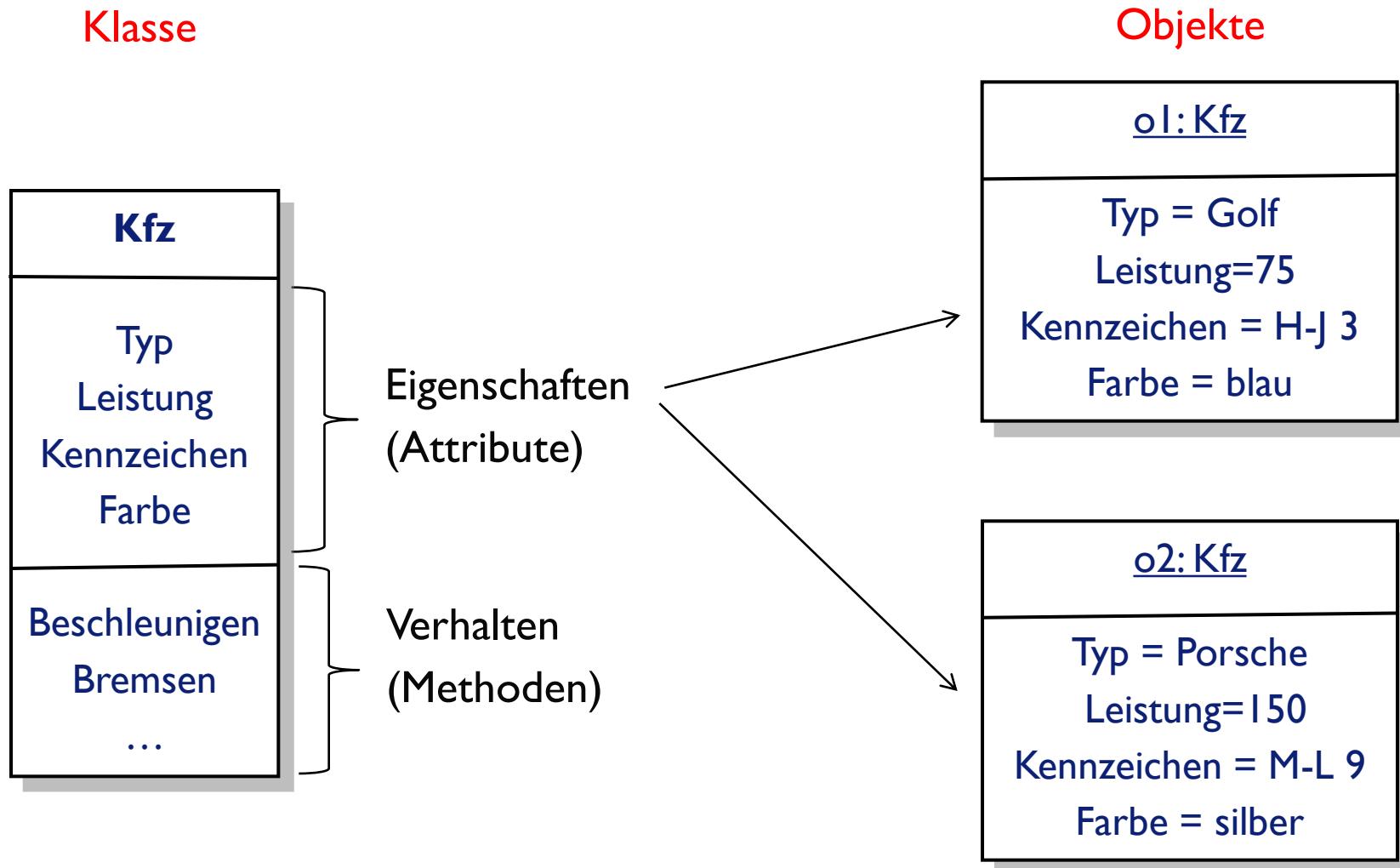
Darstellung eines Objekts

Beispiel für ein konkretes Objekt:



Grundbegriffe der OOP

Objekte und Klassen: Beispiel



Die Methoden der Klasse stehen jedem Objekt zur Verfügung!

Prinzipien der OOP

- **Geheimnisprinzip und Datenkapselung**
 - Zugriff auf Attribute bzw. Methoden erfolgt kontrolliert über Schnittstellen; keine unerwartete Veränderung des internen Objektzustandes
- **Vererbung als Spezialfall einer Klassenbeziehung**
 - Klassen können ihre Eigenschaften an spezifischere Klassen vererben.
→ Klassenhierarchie mit verschieden tiefer Abstraktion
- **Polymorphie**
 - Eine Funktion kann mehrmals implementiert sein. Beim Funktionsaufruf wird die passende Funktion ausgewählt und ausgeführt.

Geheimnisprinzip

- Geheimnisprinzip: Interna des Systems sollten verborgen (nicht sichtbar) sein
 - Interna: Innere Struktur des Datentyps, Details der Implementierung etc.
 - Grund: Fehler und Fehlverhalten vermeiden!
- Objekt sollte möglichst viele seine Attribute vor anderen Objekten verbergen
Nur das Objekt selbst sollte seine Attribute kennen und verändern dürfen
- Objekt sollte möglichst viele seiner Methoden vor anderen Objekten verbergen
Nur Methoden, die für die Benutzung des Objekts (durch andere Objekte) unbedingt erforderlich sind, sind sichtbar.

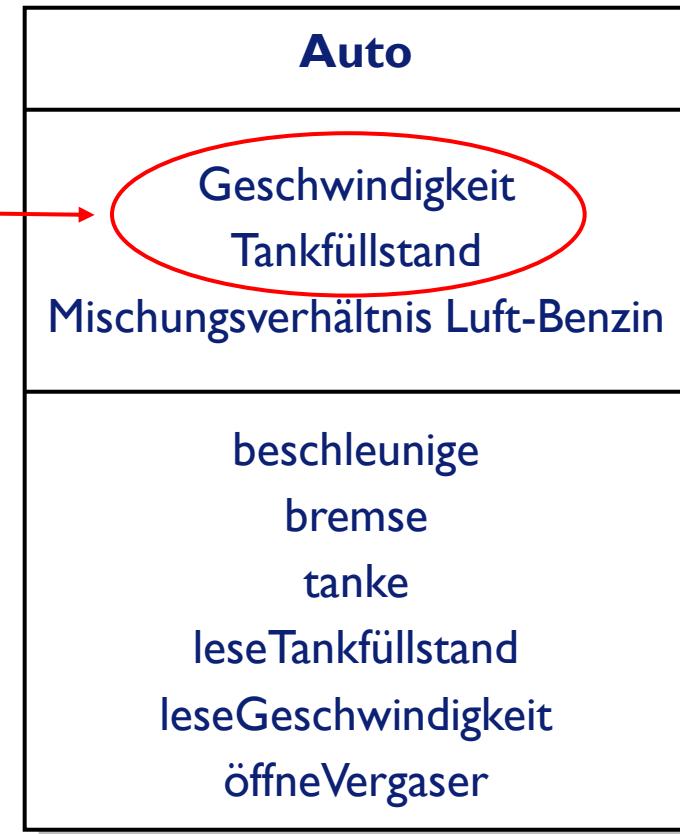
Wie kann ein Objekt trotzdem die Attribute eines anderen Objekts verändern?

- Jedes Objekt sollte Methoden zur Veränderung seiner Attribute bereit stellen
- Diese Methoden werden von anderen Objekten für den Zugriff verwendet
 - Weiterhin muss nur das Objekt seine eigenen Datenstrukturen kennen
 - Objekt behält Kontrolle darüber, welche Werte seine Attribute annehmen

Geheimnisprinzip

Beispiel:

Attribute, die für die Benutzung des Objekts wichtig sind.



- Geschwindigkeit und Tankfüllstand müssen für den Benutzer veränderbar sein (Zugriff nur über geeignete Interface-Methoden, die überprüfen, ob Werte sinnvoll sind.)

Grundbegriffe der OOP

Geheimnisprinzip

Beispiel:

Attribut, das für die Benutzung des Objekts völlig unwichtig ist.



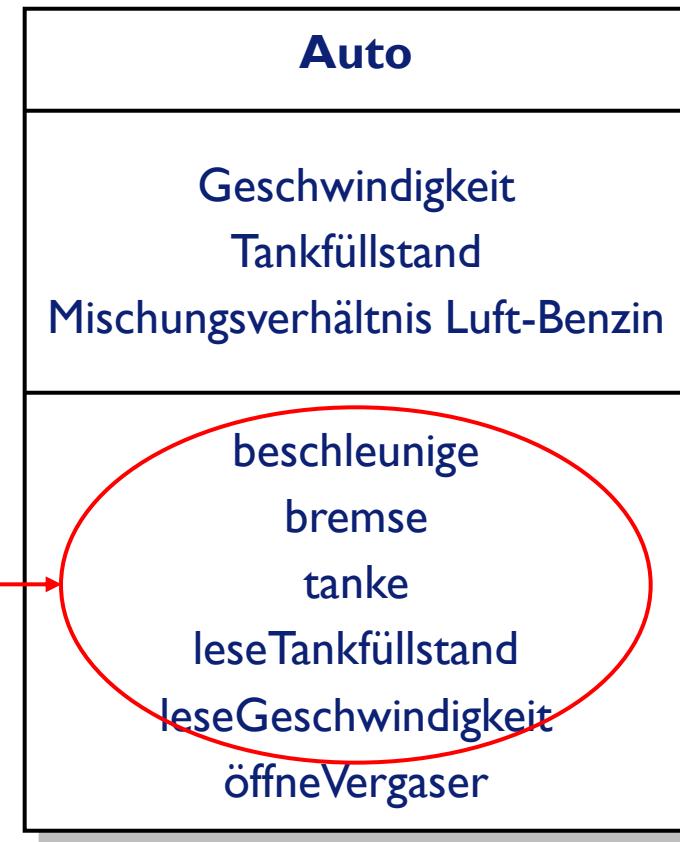
- **Geschwindigkeit** und **Tankfüllstand** müssen für den Benutzer veränderbar sein (Zugriff nur über geeignete Interface-Methoden, die überprüfen, ob Werte sinnvoll sind.)
- **Mischungsverhältnis Luft-Benzin** muss für Benutzer nicht direkt zugänglich sein
Sollte für den Benutzer verborgen sein (Detail der Implementierung, nicht der Benutzung)

Grundbegriffe der OOP

Geheimnisprinzip

Beispiel:

Methoden, die für die Benutzung
des Objekts wichtig sind.



- Benutzer muss bremsen, beschleunigen, tanken und Tank überprüfen können

Grundbegriffe der OOP

Geheimnisprinzip

Beispiel:

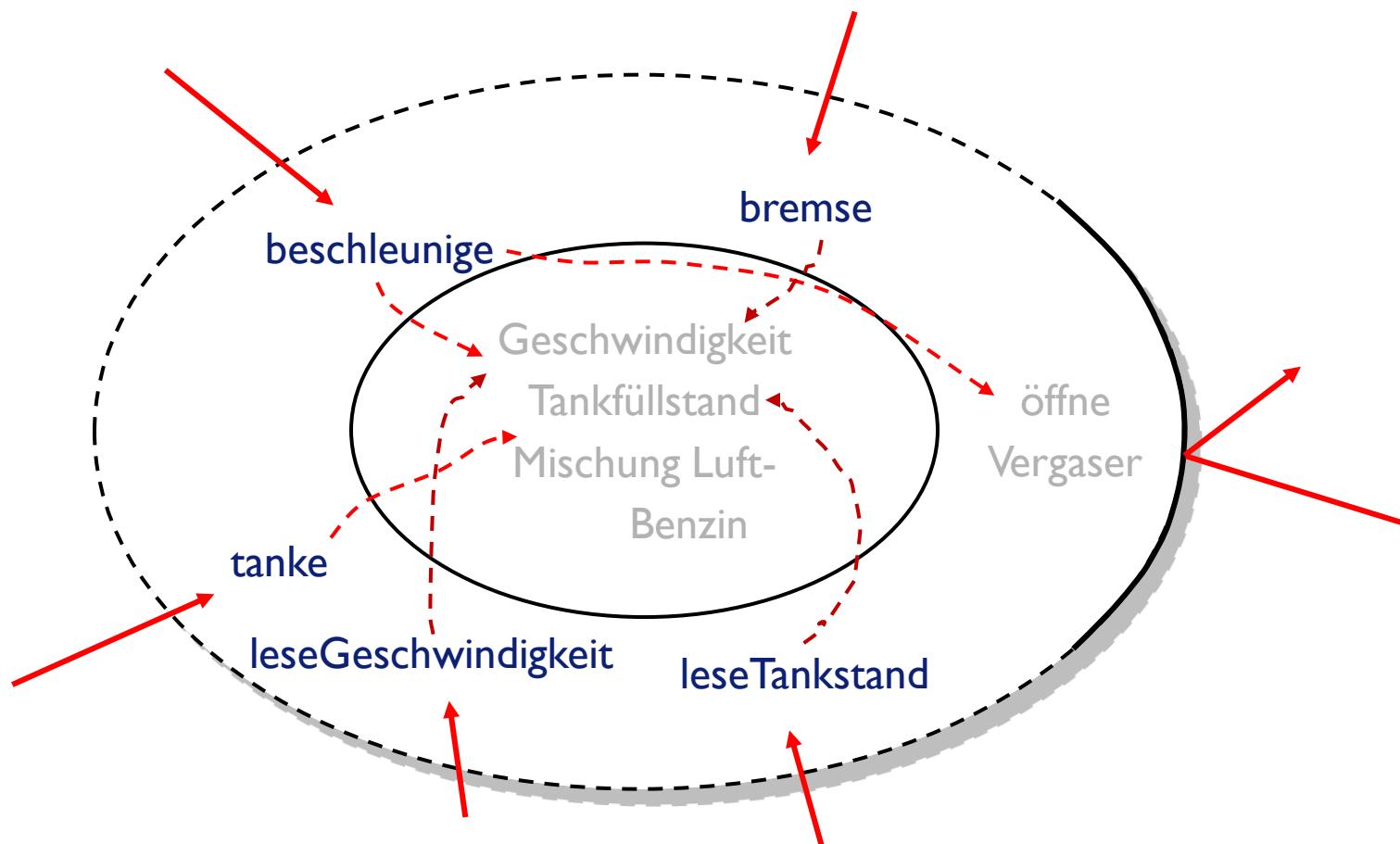
Methode, die für die Benutzung
des Objekts unwichtig ist.
(Aber: Wichtig für das Funktionieren!)



- Benutzer muss bremsen, beschleunigen, tanken und Tank überprüfen können
- Methode öffneVergaser muss für den Benutzer nicht direkt aufrufbar sein
Wird von Methode beschleunige aufgerufen
Sollte für den Benutzer verborgen sein

Grundbegriffe der OOP

Geheimnisprinzip



Sichtbarkeit und Zugriffsarten

Umsetzung des Geheimnisprinzips durch **Regulierung des Zugriffs** auf Attribute und Methoden durch folgende Zugriffsarten (in Klammern die UML Kurznotation):

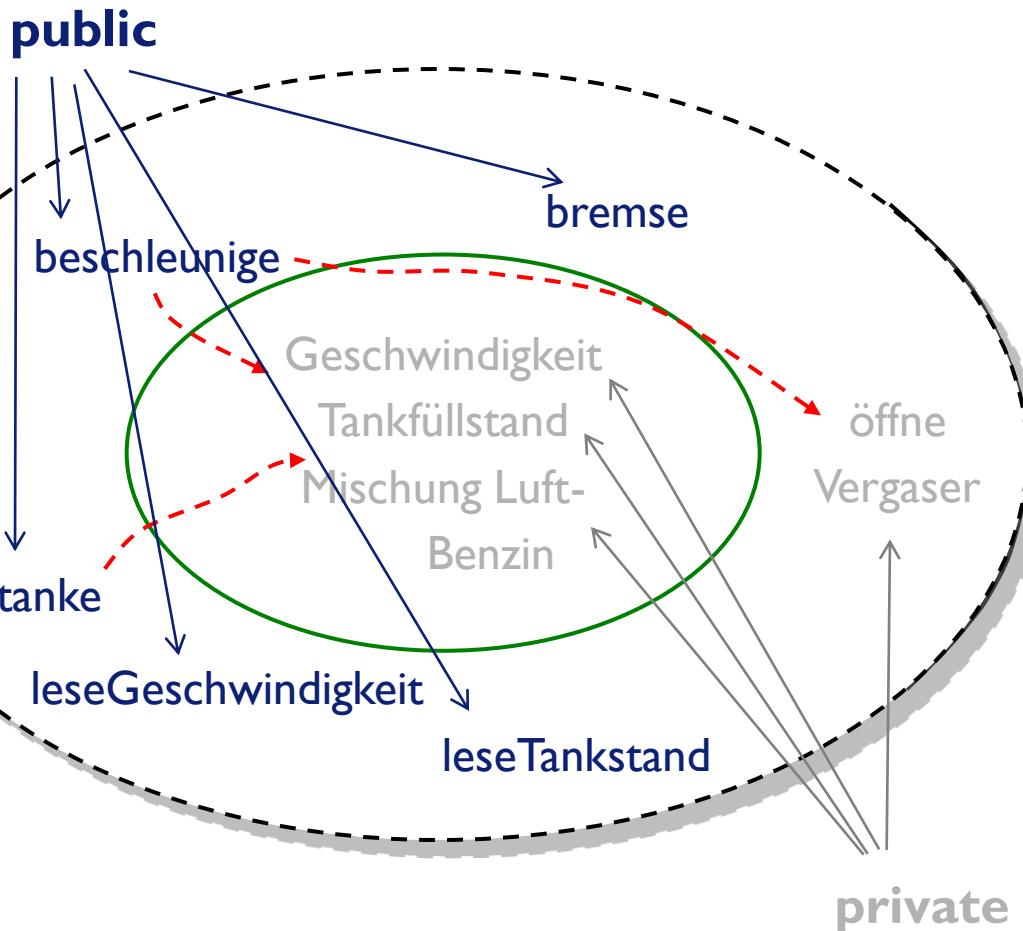
- **public (+)**: keine Zugriffsbeschränkungen, von außen zugreifbar (über Objekt)
- **private (-)**: kein Zugriff von außen, nur innerhalb der Klasse sichtbar
- **protected (#)**: kein Zugriff von außen, nur innerhalb der Klasse und in Spezialisierungen der Klasse sichtbar (siehe später)

Beispiel:

- Methode „Öffne Vergaser“ soll geheim sein → „private“
- Methode „Beschleunige“, „Bremse“ müssen öffentlich verfügbar sein → „public“

Grundbegriffe der OOP

Sichtbarkeit und Zugriffsarten: Beispiel



Auto

- Geschwindigkeit
 - Tankfüllstand
 - Mischungsverh. Luft-Benzin
-
- + beschleunige
 - + bremse
 - + tanke
 - + leseTankfüllstand
 - + leseGeschwindigkeit
 - öffneVergaser

Kapselung

- Kapselung: Daten (Attribute) sind mit Operationen (Methoden) assoziiert
 - Diese und nur diese Methoden können auf die Daten zugreifen
 - kontrollierter Zugriff auf Methoden / Attribute von Klassen

Jedes Objekt sollte aus einer Schnittstelle und einer Implementierung bestehen

Schnittstelle: Beinhaltet die Aussensicht

→ alle Erwartungen, die ein Benutzer an das Objekt haben könnte

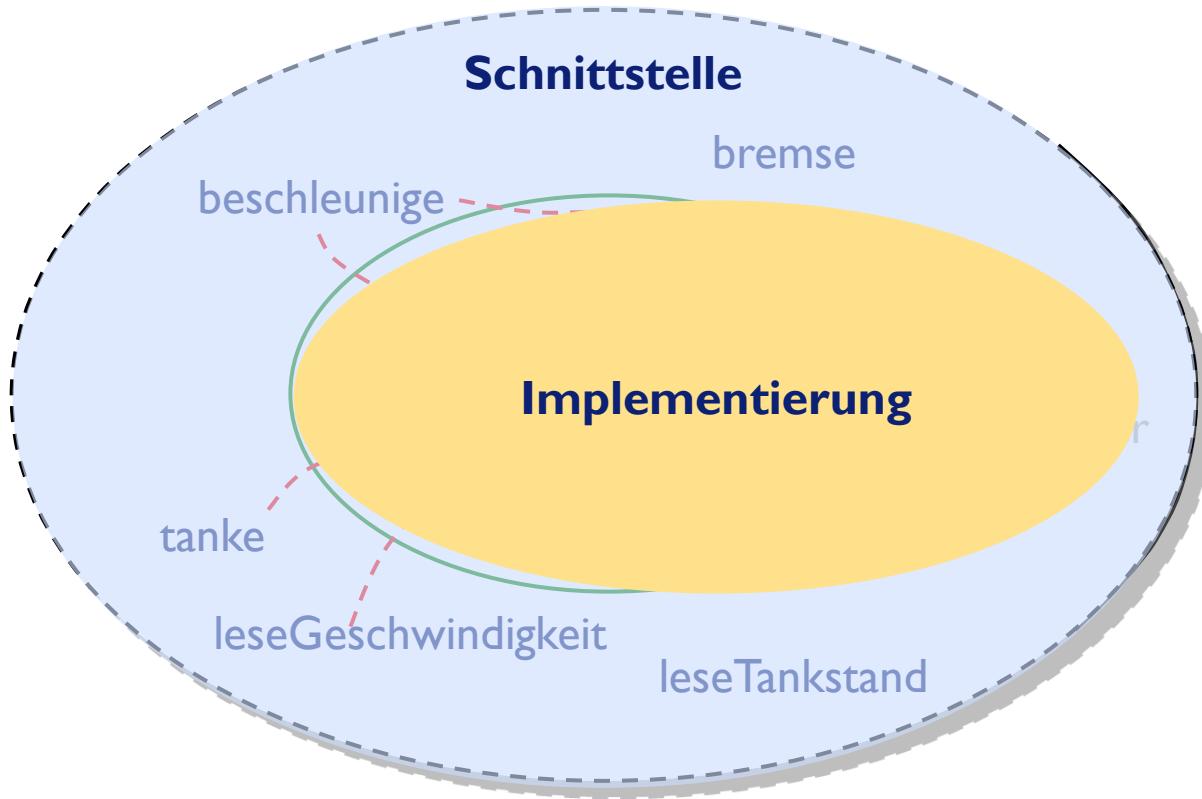
Implementierung: Kapselt alle Details, die der Benutzer nicht wissen muss und will

→ Schutz vor unerwünschten Veränderungen

Kein direkter Zugriff auf interne Datenstruktur – nur über Schnittstellen!

Grundbegriffe der OOP

Kapselung



Auto
– Geschwindigkeit
– Tankfüllstand
– Mischungsverh. Luft-Benzin
+ beschleunige
+ bremse
+ tanke
+ leseTankfüllstand
+ leseGeschwindigkeit
– öffneVergaser

Beispiel:

- Kein direkter Zugriff auf Attribut „Geschwindigkeit“ (Implementierung)
- Zugriff nur über Methoden „beschl.“ / „bremsen“ / „leseGeschw.“(Schnittstelle)

Kapselung: Vor- und Nachteile

Vorteile:

- Einfachere Benutzung der Klasse
 - Benutzer muß nur Schnittstelle verstehen
- Sicherere, zuverlässigere Programme
 - Zugriff auf Objektdaten wird kontrolliert
- Größere Flexibilität der Programme
 - unter Beibehaltung der Schnittstelle kann die Implementierung einer Klasse geändert werden, ohne das restliche Programm ändern zu müssen

Nachteile:

- Zusätzlicher Programmieraufwand für die Erstellung der Zugriffsfunktionen
- Ggf. Geschwindigkeitseinbußen durch den Aufruf von Zugriffsfunktionen (direkter Zugriff auf die Datenelemente wäre schneller)

Klassifikation

Wie finden wir die richtigen Klassendefinitionen?

Leider gibt es weder

- ein Kochrezept zum Finden der Klassen noch
- überhaupt so etwas wie eine perfekte Klassenstruktur

Auf einer Konferenz wurden Entwickler gefragt, welche Regeln sie anwenden, um Klassen und Objekte zu identifizieren.

Eine Antwort lautete: „Das ist eine fundamentale Frage, auf die es keine einfache Antwort gibt. Ich probiere einfach Sachen aus.“

Klassifikation

Prozess in der objektorientierten Analyse

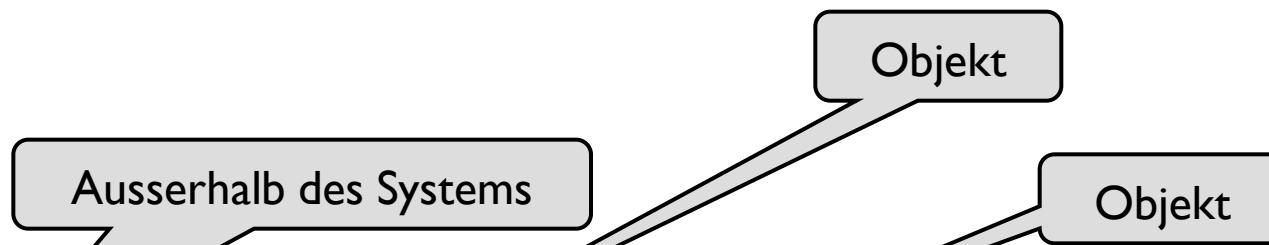
- I. Aufschreiben von Szenarien, wie ein Benutzer das Programm verwendet („use case diagram“)
2. Beschreibung der Systemteile:
 - I. Finden der Entitäten, z.B. durch Substantiv-Analyse → Objekte

Beispiel: Beschreibung einer Bibliotheks-Anwendung in Umgangssprache

In der Bibliothek gibt es Bücher und Zeitschriften. Von Büchern kann es mehrere Exemplare geben. Einige Bücher sind nur für Kurzzeitentleihe gedacht. Alle anderen Bücher können von jedem Bibliotheksmitglied für drei Wochen entliehen werden. Mitglieder der Bibliothek können normalerweise bis zu 6 Einheiten entleihen, während Mitarbeiter der Bibliothek bis zu 12 Einheiten zur gleichen Zeit entleihen können. Nur Mitarbeiter der Bibliothek dürfen Zeitschriften entleihen.

Grundbegriffe der OOP

Klassifikation



Beispiel: Beschreibung einer Bibliotheks-Anwendung in Umgangssprache

In der Bibliothek gibt es Bücher und Zeitschriften. Von Büchern kann es mehrere Exemplare geben. Einige Bücher sind nur für Kurzzeitentleihe gedacht. Alle anderen Bücher können von jedem Bibliotheksmitglied für drei Wochen entliehen werden. Mitglieder der Bibliothek können normalerweise bis zu 6 Einheiten entleihen, während Mitarbeiter der Bibliothek bis zu 12 Einheiten zur gleichen Zeit entleihen können. Nur Mitarbeiter der Bibliothek dürfen Zeitschriften entleihen.

Klassifikation

Prozess in der objektorientierten Analyse

- I. Aufschreiben von Szenarien, wie ein Benutzer das Programm verwendet („use case diagram“)
2. Beschreibung der Systemteile:
 - I. Finden der Entitäten, z.B. durch Substantiv-Analyse → Objekte
 2. Gruppierung der Objekte in Klassen
 3. Zuordnung der Funktionalitäten zu den Objekt-Klassen
3. Schrittweise Verbesserung der Beschreibung durch Hinzunahme weiterer Szenarien

Klassifikation

Ähnliches Verfahren: Dokumentenanalyse

Analyse der in einer Anwendung verwendeten Dokumente

- Bestellformulare
- Rechnungen
- ...

Daraus lassen sich in der Regel die wichtigen Objekte ableiten, wie z.B.

- Kunde
- Auftrag
- Artikel
- ...

Klassifikation

Eigene Erfahrung in der Entwicklung von Forschungssoftware:

Diskussion der Szenarien mit Kollegen → Erster Entwurf

Weitere Diskussion des Entwurfs → Verbesserter Entwurf

Implementation und ggf. weitere Verfeinerungen während der Programmierung

Klassifikation

Einige Hinweise zum Entwurf von Klassen:

- Es sollte so viel wie möglich an Information in der Klasse **verborgen** werden (Geheimnisprinzip, Kapselung)
- Die **Schnittstelle** für den Anwender sollte so **klein und einfach wie möglich**, aber so **umfangreich wie nötig** sein, um das gegebene Problem zu lösen
- Die Klasse sollte so programmiert werden, daß spätere **Erweiterungen** einfach realisiert werden können (siehe später; Vererbung)

Beziehungen zwischen Klassen

Wichtiger Aspekt des objektorientierten Programmierens:

- Verwende bereits vorhandene Klassen, um neue Klassen zu definieren
 - Baue Softwaresystem aus mehreren Klassen zusammen

Beispiel: Flugzeug

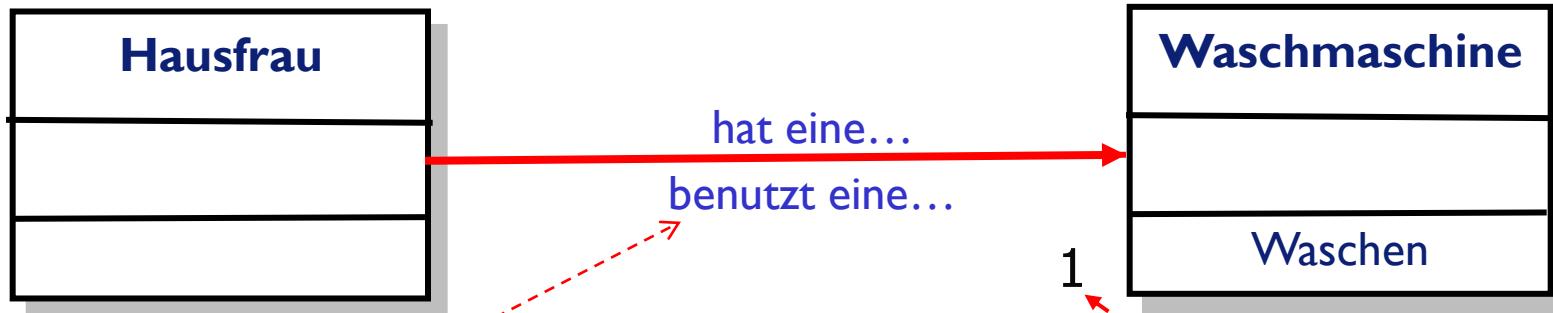
- Einzelteile haben inhärente Neigung, zur Erde zu fallen
 - Setze Einzelteile zu Flugzeug zusammen:
Erst die Zusammenarbeit aller Teile ermöglicht das Fliegen
Das Ganze ist mehr als die Summe seiner Teile.

Beziehungen zwischen den Klassen:

- Assoziation („kennt ein“ / „benutzt ein“ – Beziehung)
 - Aggregation, Komposition („hat ein“ / „besteht aus“ – Beziehung)
 - Vererbung („ist ein“ / „ist eine Art von“ – Beziehung)

Assoziation

- Einfachste Form einer Beziehung zwischen (den Objekten von) Klassen; „lose Kopplung“
- Ein Objekt kennt das andere und kann mit ihm kommunizieren
- Beide Objekte existieren völlig unabhängig voneinander
- Beispiel: **binäre Assoziation** (Verbindung zwischen **zwei Objekten**)
 - Uni- oder bidirektionale Assoziation (einseitige / gegenseitige Bekanntschaft)



- Darstellung: Linie / Pfeil; ggf. zusätzliche Angaben:
 - Name, evt. Leserichtung ►, evt. Navigationsrichtung (Pfeil →)
 - Multiplizität: Anzahl Objekte, die über Assoziation bekannt sind

Grundbegriffe der OOP

Assoziation: Beispiele

Unidirektionale Assoziation:



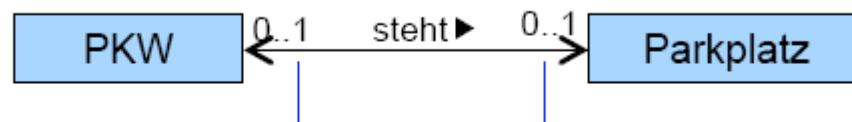
Kunde kennt Auftrag;
Auftrag weiß nicht, wer ihn erteilt hat

Bidirektionale Assoziation:



Kunde kennt Auftrag;
Auftrag kennt Auftraggeber

Multiplizität:



Jeder Parkplatz wird von keinem
oder einem PKW besetzt

Jeder PKW steht auf keinem
oder genau einem Parkplatz

Realisierung in C++: z.B. durch Zeiger (siehe später)

- Unidirektional: Zeiger in einer Klasse (in Klasse Kunde: Zeiger auf Klasse Auftrag)
- Bidirektional: Zeiger in beiden Klassen

Quelle: Bittel

Aggregation und Komposition

- Aggregation: Zusammensetzung eines Objekts (Aggregatobjekt) aus Einzelteilen
 - Spezielle Form der Assoziation: „part-of“-Beziehung, d.h. das Aggregatobjekt besteht aus den Teilen (und kennt sie nicht nur)
 - Teile sind **nicht vom Ganzen existenzabhängig** (Lebensdauer ggf. unabhängig)
 - Darstellung: Linie mit weißer Raute
 - Implementierung: Wie Assoziation



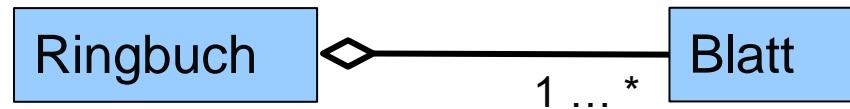
- Komposition (strenge Form der Aggregation): Zusammensetzung eines Objekts (Kompositionsobjekt) aus verschiedenen Einzelteilen (Komponenten)
 - Teile sind **vom Ganzen existenzabhängig**
 - Darstellung: Linie mit schwarzer Raute
 - Implementierung: z.B. Einbettung des Teilobjekts als Attribut (siehe später)



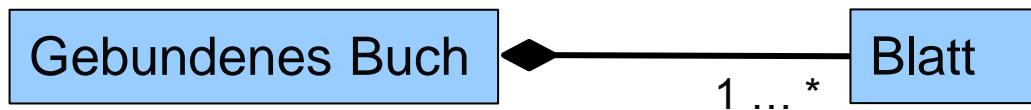
Quelle: Bittel

Aggregation und Komposition: Beispiel

- Aggregation: Zusammensetzung eines Objekts (Aggregatobjekt) aus Einzelteilen
 - Teile sind *nicht* vom Ganzen existenzabhängig (Lebensdauer ggf. unabhängig)

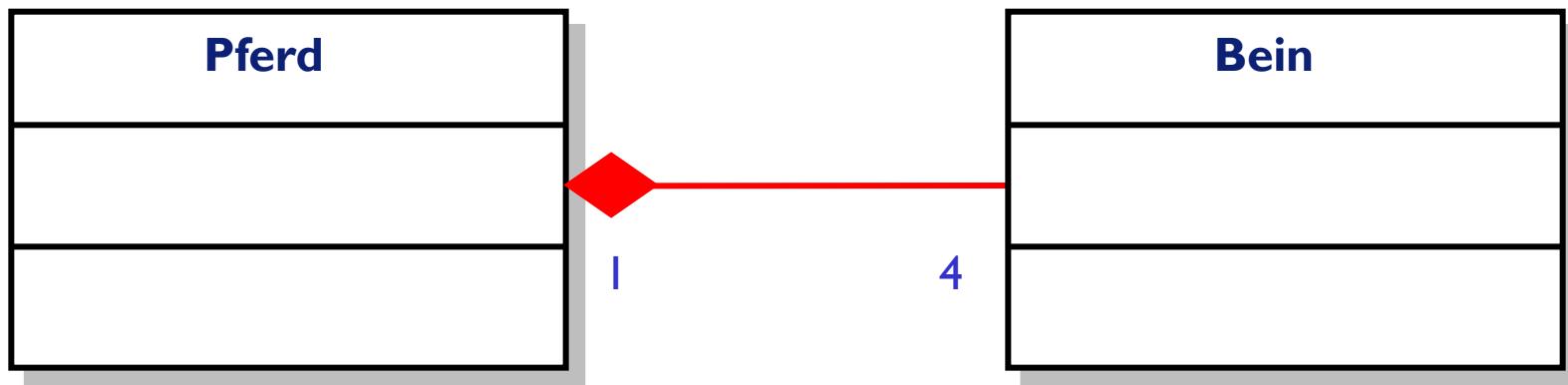


- Komposition (strenge Form der Aggregation): Zusammensetzung eines Objekts (Kompositionsobjekt) aus verschiedenen Einzelteilen (Komponenten)
 - Teile sind vom Ganzen existenzabhängig



Komposition

- Stärkste nicht auf Vererbung beruhende Form der Beziehung zwischen Objekten
- „Besteht-aus-Beziehung“
 - Beispiel: Ein Pferd und seine Beine



- Bei Erzeugung eines Pferde-Objekts bekommt es automatisch 4 Beine, die von anderen Klassen nicht verwendet werden können
- Stirbt das Pferd, sterben auch seine Beine

Vererbung

Motivation:

Bisher: gleichartige Objekte derselben Klasse / Objekte völlig verschiedener Klassen

Nunmehr: Betrachte Erzeugung einander ähnlicher Objekte

Beispiel: Personalverwaltung, Lohnbuchhaltung

- In vielen Unternehmen: unterschiedliche Arten von Arbeitnehmern
→ unterschiedliche Arten der Entgeltberechnung

Arbeiter: Stundenlohn * geleistete Stunden = Monatslohn

- Angestellte
- tarifliche Mitarbeiter: Monatslohn + Zuschlag für geleistete Überstunden
 - AT Mitarbeiter: kein Zuschlag für Überstunden, Bestandteil des Vertrags
 - Geschäftsführer: Monatslohn + erfolgsabhängige Zulage

Vererbung

Soll für jeden Mitarbeiter-Typ eine eigene Klasse geschrieben werden?

Nein, weil es viele Gemeinsamkeiten gibt:

Alle tragen einen Namen, haben eine Adresse, einen Familienstand, Lebenslauf usw.

- Mehrfachaufwand beim Aufbau und bei der Pflege (Änderungen)
- Gefahr von Inkonsistenzen (Adresse mit/ohne Tel.Nr.)

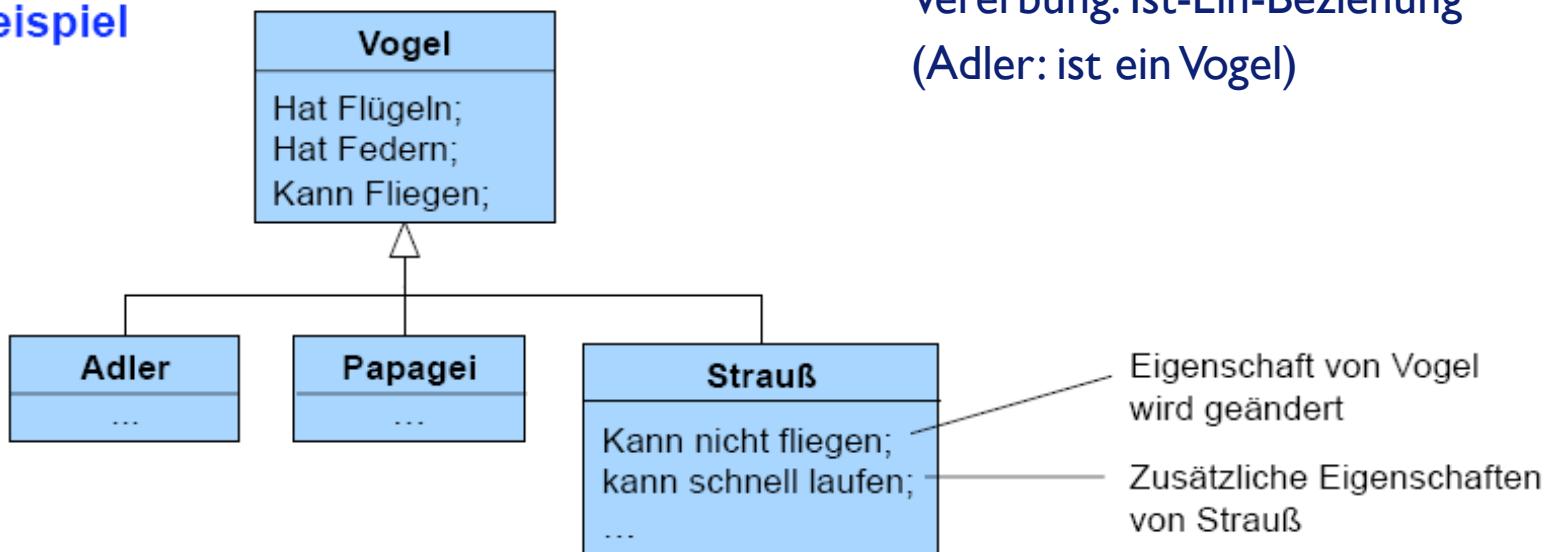
Lösung: **Vererbung**

- Führe „Basisklasse“ Mitarbeiter ein, die alle Attribute und Methoden enthält, die für *alle* Mitarbeiter relevant sind
- Besonderheiten der Mitarbeiter-Typen werden in speziellen Klassen geregelt

Vererbung

- **Vererbung:** Aus vorhandener Basisklasse A kann neue Klasse B abgeleitet werden
 - Klasse B erbt alle Attribute und Methoden der Basisklasse A
 - Klasse B kann um zusätzliche Attribute und Methoden erweitert werden
 - Klasse B kann vererbte Methoden aus A verändern (überschreiben)

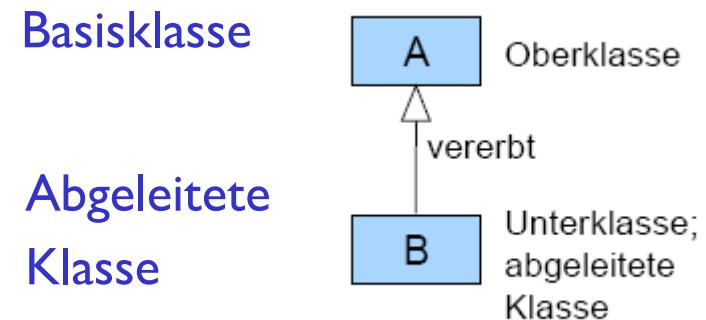
Beispiel



Vererbung

- Terminologie:
 - Basisklasse, Oberklasse oder Superklasse
 - Abgeleitete Klasse, Unterklasse oder Subklasse
- Darstellung:

Pfeil mit geschlossener, leerer Spitze;
zeigt auf Basisklasse
- Eine abgeleitete Klasse kann weiter spezialisiert werden
→ Vererbungshierarchie



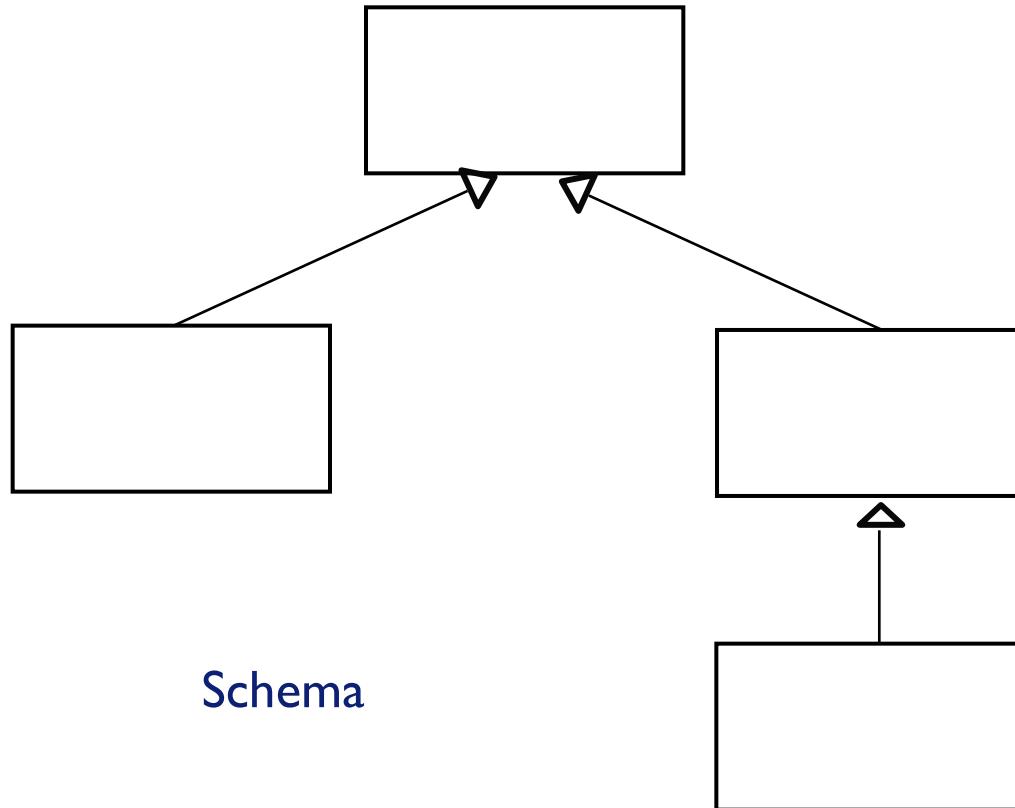
Quelle: Bittel

Grundbegriffe der OOP

Vererbung

Zeichnen Sie eine Vererbungshierarchie mit den Klassen

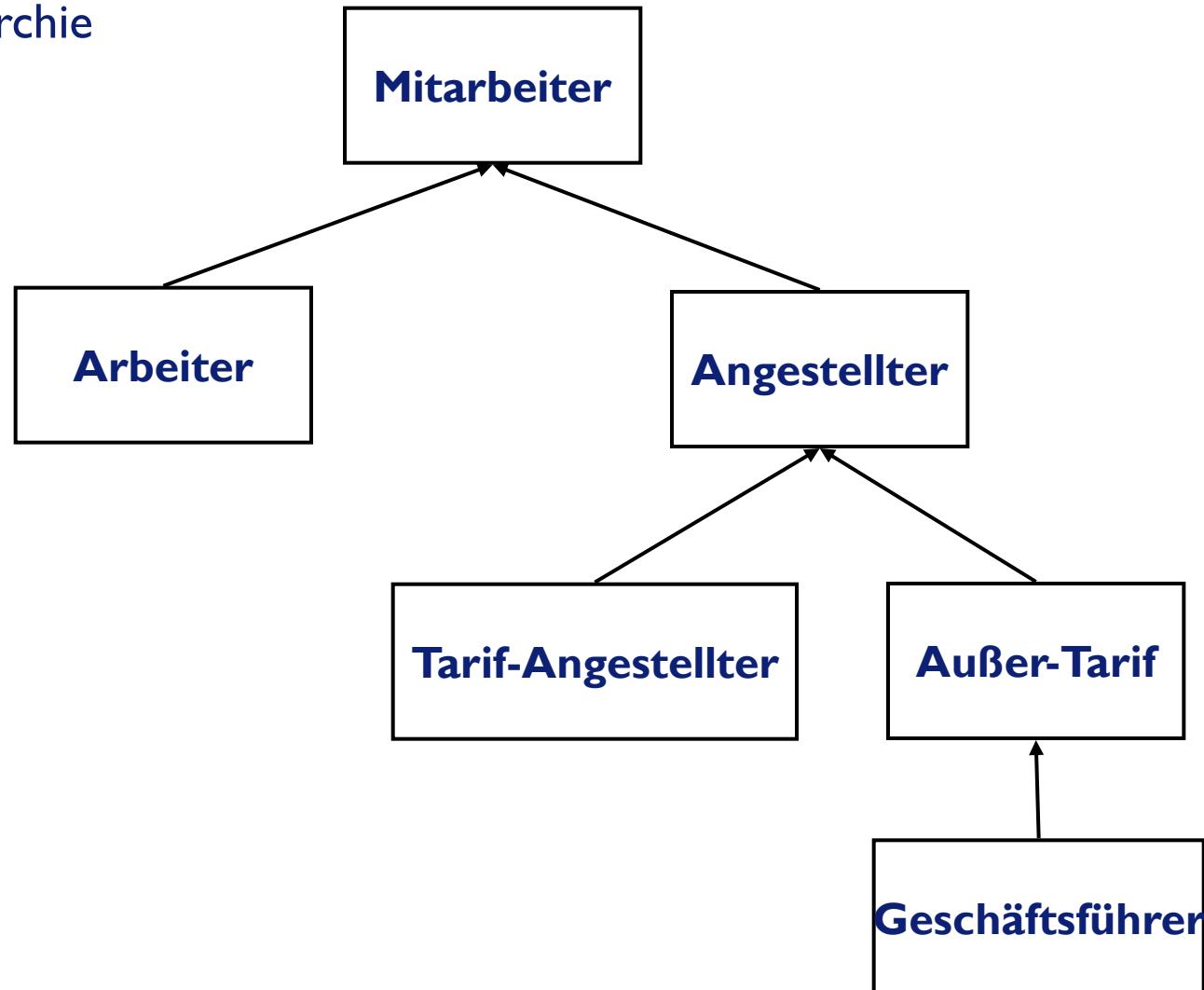
Arbeiter, Geschäftsführer, Tarif-Angestellter, Mitarbeiter,
Außertarif Angestellter, Angestellter



Grundbegriffe der OOP

Vererbung

Vererbungshierarchie



Vererbung

Je nach Sichtweise ist Vererbung eine Spezialisierung oder Generalisierung

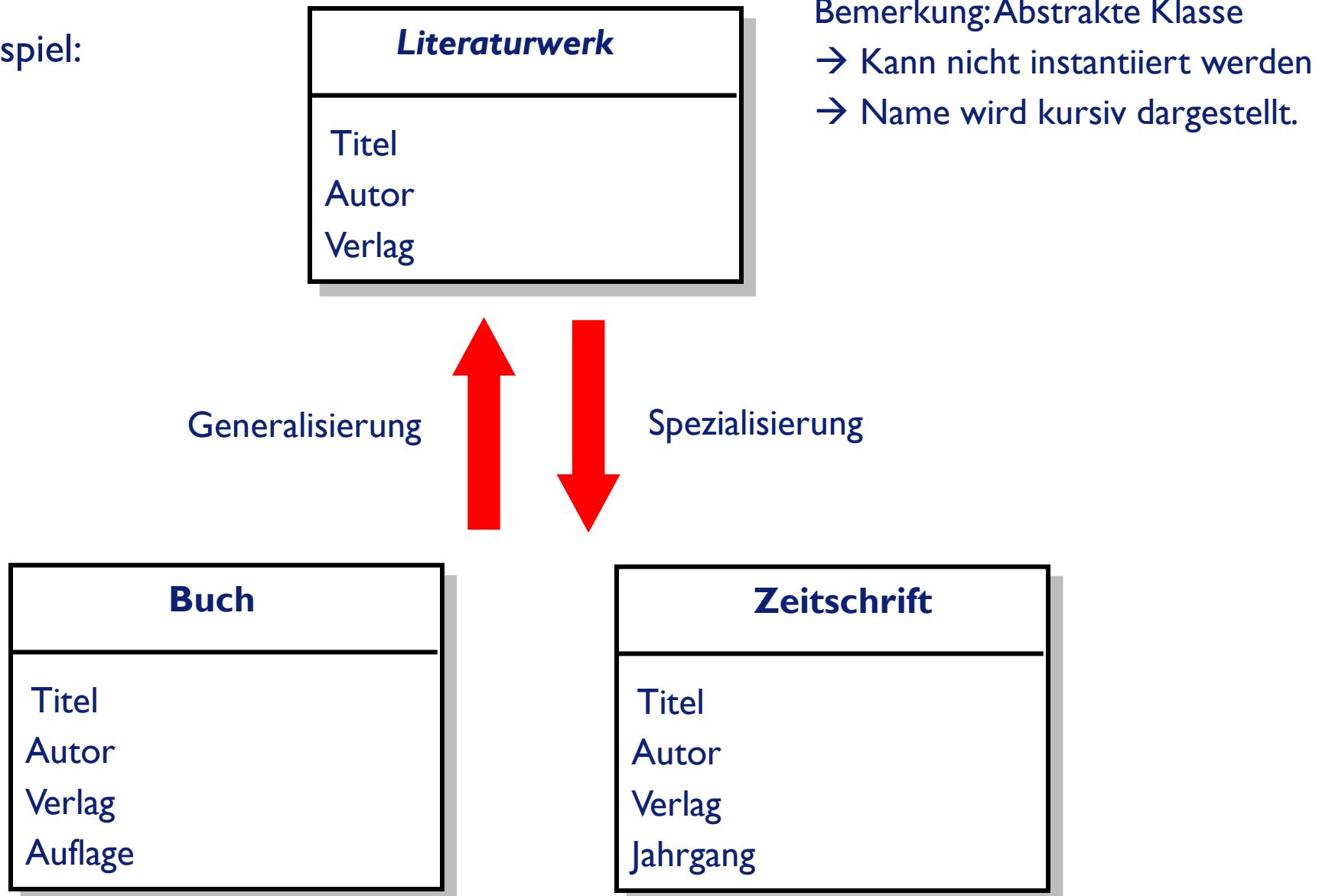
Spezialisierung: Aus Sicht der Basisklasse sind abgeleitete Klassen Spezialisierungen
Bsp: „Angestellter“ ist Spezialisierung eines „Mitarbeiters“

Generalisierung: Aus Sicht der abgeleiteten Klassen ist die Basisklasse eine
Verallgemeinerung (Generalisierung)
Bsp.: „Angestellter“ ist Verallgemeinerung von „Tarif-Angestellter“

Grundbegriffe der OOP

Vererbung

Beispiel:



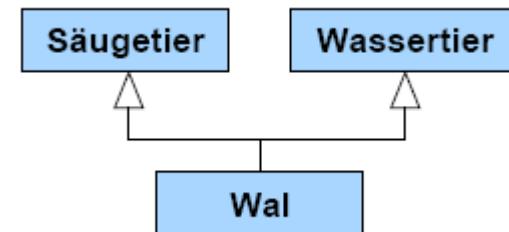
Mehrfachvererbung

Eine Klasse erbt von mehreren Basisklassen

- Klasse hat mehr als eine direkte Basisklasse
- Interessant, wenn Objekt die Eigenschaften mehrerer Basisklassen in sich vereinigen soll.

Beispiel:

Wal erbt gleichzeitig die Eigenschaften eines Säugetiers und eines Wassertiers.



Bemerkung:

Mehrfachvererbung kann zu Namenskonflikten führen, wenn die Basisklassen Elemente beinhalten, die den gleichen Namen haben.

Vererbung

Ziele der Vererbung:

- **Vermeidung von Redundanz** im Programmcode
(gemeinsame Klasseneigenschaften genau einmal in Basisklasse definiert)
- **Wiederverwendbarkeit** von bereits erzeugtem Programmcode
- **Bessere Modellierung** der Realität

Probleme mit Vererbung / Designfehler:

- Sehr starke Kopplung zwischen Klassen (und damit auch zwischen Objekten)
→ große Nachteile, falls Klassendesign nicht sorgfältig analysiert wurde

Grundbegriffe der OOP

Vererbung

Beispiel:



Gibt es Einwände gegen
eine so definierte Klasse?

Ist ein Zebra schwarz
oder weiss? Farbe macht
keinen Sinn → Muster

Grundbegriffe der OOP

Vererbung

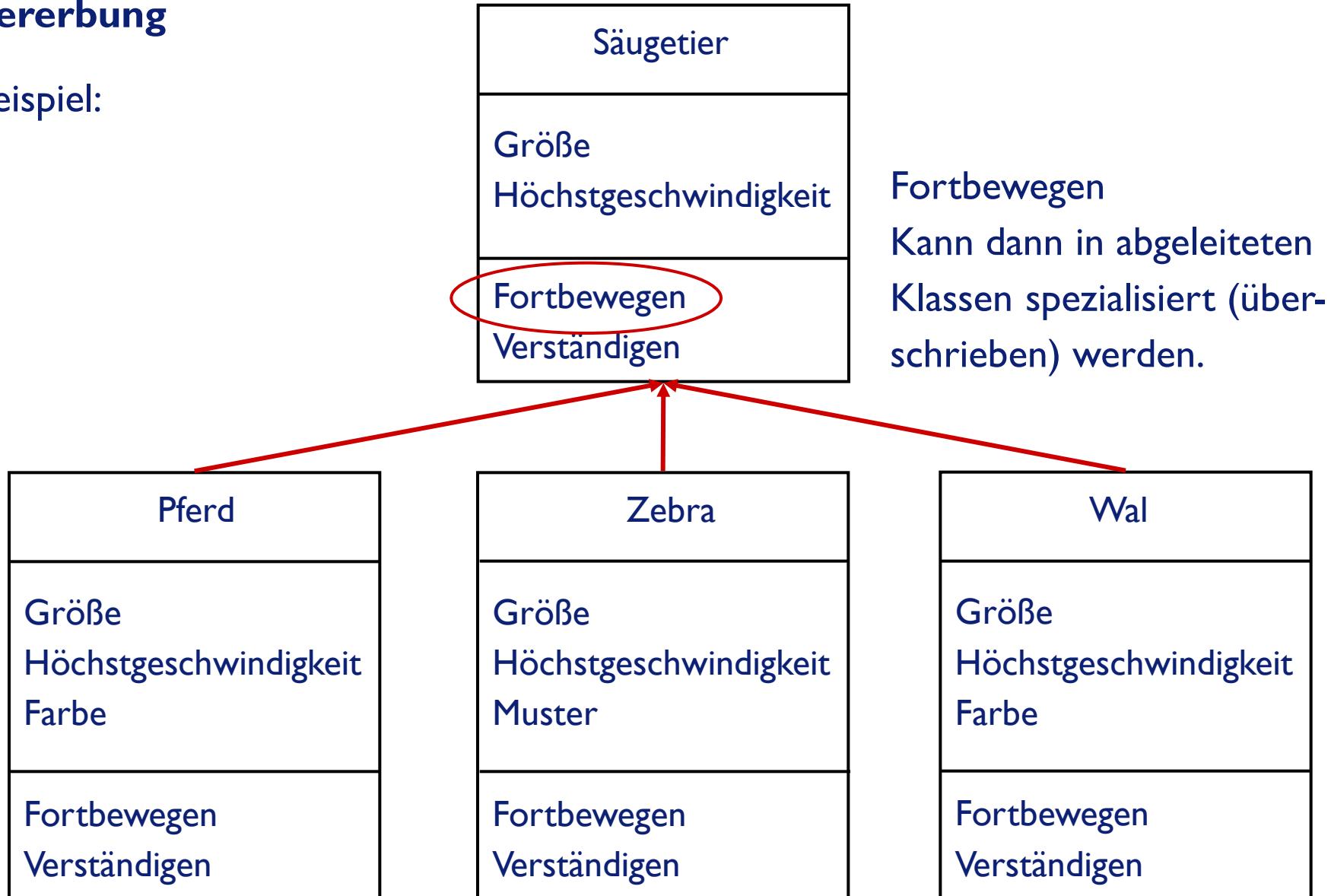
Beispiel:



Grundbegriffe der OOP

Vererbung

Beispiel:



Fortbewegen
Kann dann in abgeleiteten
Klassen spezialisiert (über-
schrieben) werden.

Designfehler

Designfehler wie der eben beschriebene können schwerwiegende Folgen haben.

Das Ändern einer Basisklasse spät in der Programmentwicklung ist mit erheblichem Aufwand verbunden

Warum?

- Ein Fehler in der Basisklasse pflanzt sich in die abgeleiteten Klassen fort ggf. durch die ganze Klassenhierarchie
- Basisklassen können in sehr vielen Programmteilen zur Anwendung kommen
→ an vielen Stellen im Programm muss Code geändert werden

Designfehler

Im Beispiel:

- An vielen Stellen gehen wir davon aus, dass die Methode Laufen (mit bestimmten Eigenschaften) zur Verfügung steht.
- Wenn Laufen durch das allgemeinere Fortbewegen ersetzt wird:
 - Es reicht nicht, dort wo Laufen aufgerufen wird, einfach den Namen zu ändern
 - Fortbewegen tut etwas anderes als Laufen
 - im Programm muss auf dieses andere Verhalten reagiert werden

Nachteile von Vererbung

- Designfehler können sich lawinenartig ins ganze Programm fortpflanzen
- Es werden auch unbenötigte / unerwünschte Teile der Basisklasse vererbt
→ Nachkommen werden immer umfangreicher

Programmierregel: Vererbung sollte möglichst sparsam und nur dort eingesetzt werden, wo sie wirklich sinnvoll ist.

Polymorphie

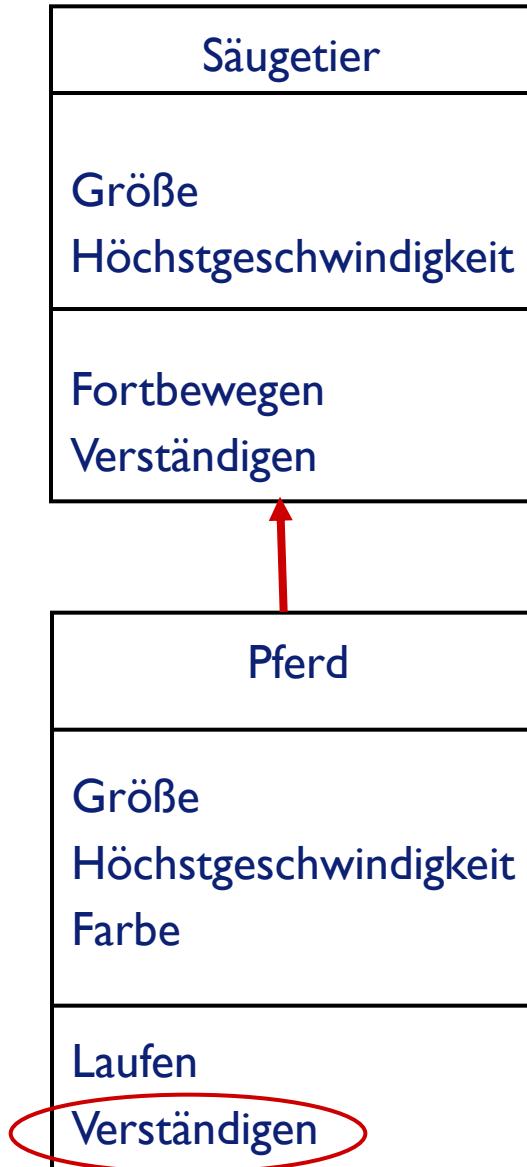
- Name kommt aus dem Griechischen und bedeutet Viel- oder Verschiedengestaltigkeit
- Gibt es auch z.B. in der Mineralienkunde:
Manche Mineralien (Kalziumcarbonat) nehmen je nach Druck und Temp. unterschiedliche Kristallformen (Gestalten) an

Polymorphie in der objektorientierten Programmierung:

- a) Überladung von Funktionen („Funktionspolymorphie“)
 - Funktionen / Methoden haben gleichen Namen, aber unterschiedl. Signatur (d.h. unterschiedliche Parametertypen und / oder Anzahl Parameter)
- b) Vererbung: abgeleitete Klassen überschreiben Methode der Basisklasse
 - Methoden haben gleiche Namen, gleiche Parameter, aber anderen Inhalt
 - Dennoch wird – je nach Objekttyp – die „richtige“ Funktion aufgerufen

Grundbegriffe der OOP

Polymorphie



Festlegen, auf welche Weise
sich Pferde-Objekte verstndigen.

→ Überschreiben der Methode
Verständigen und Festlegen der
Art und Weise des Wieherns

Persistenz

Typisch: Programm erzeugt Objekte, die an ihrem Lebensende zerstört werden

→ transiente (oder flüchtige) Objekte

Persistenz bedeutet Dauerhaftigkeit

→ persistente Objekte werden in geeigneter Form (z.B. Datenbank) gespeichert

In objektorientierter Programmierung:

Objektvariablen existieren, solange die Objekte vorhanden sind und „verfallen“ nicht nach Abarbeitung einer Methode.

Grundbegriffe der OOP

Objektorientierte Analyse (OOA) und objektorientiertes Design (OOD)



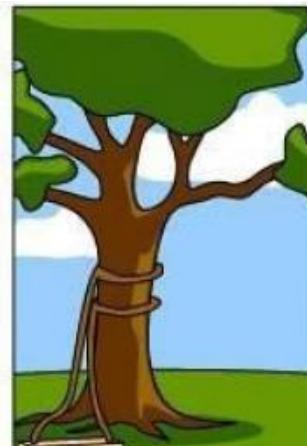
Was der Kunde erklärte



Was der Projektleiter verstand



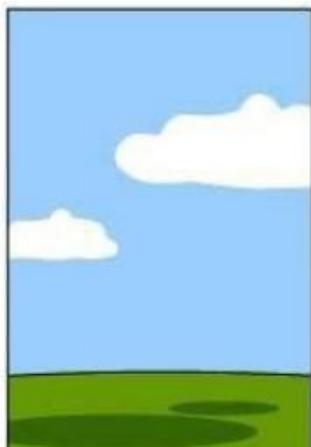
Wie es der Analytiker entwarf



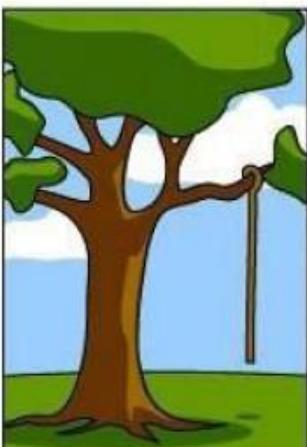
Was der Programmierer programmierte



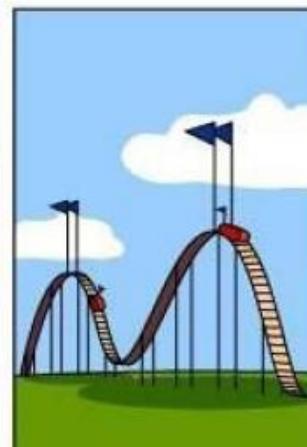
Was der Berater definierte



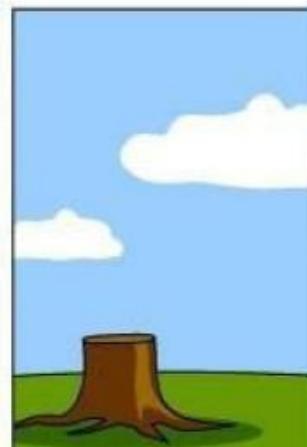
Wie das Projekt dokumentiert wurde



Was installiert wurde



Was dem Kunden in Rechnung gestellt wurde



Wie es gewartet wurde



Was der Kunde wirklich gebraucht hätte

Zusammenfassung

I. Motivation

Software wird immer komplexer (gestiegener Umfang, gestiegene Anforderungen); Ursachen:

- Komplexität des gegebenen Problems
 - Schwierigkeiten beim Management des Entwicklungsprozesses
 - Software-Flexibilität
- Gestiegene Fehlerhäufigkeit
- Lösung: neue Programmierkonzepte:

„Objektorientierte Programmierung“

Zusammenfassung

2. Prozedurale Programmierung

Hauptmerkmale:

- Funktion (oder Prozedur) ist der fundamentale Baustein für ein Programm
- Keine logische Verbindung von Daten und darauf operierenden Funktionen
- Daten und Funktionen können beliebig im Quelltext verstreut liegen
- Keine Datenkapselung

Haupt-Nachteile:

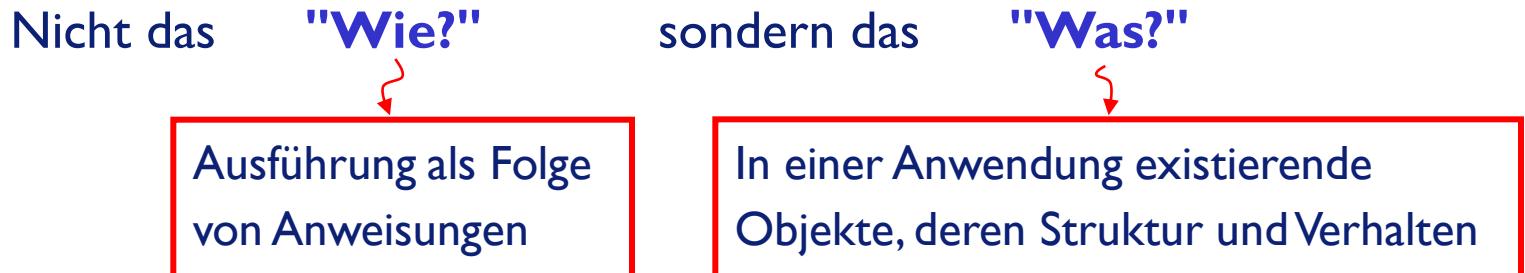
- Bei großer Programmkomplexität ist es schwierig, den Überblick zu behalten:
Welche Programmteile arbeiten mit welchen Datenstrukturen?
- Kein Schutz sensibler Daten vor fehlerhafter Veränderung
- Grosser Aufwand für Pflege und Anpassung komplexer Systeme
 - Geänderte Datenstruktur: Welche Funktionen arbeiten mit dem Format?
 - Entwickler muss viele unnötige Details von Funktionen und Datenstrukturen kennen
- Eingeschränkte Wiederverwendbarkeit von Programmteilen

Zusammenfassung

3. Objektorientierte Programmierung - Überblick

Hauptmerkmale:

- Im Zentrum des Programmdesigns:



- Verwendung von **Objekten** als besondere Programm- und Datenstruktur
- Objekte besitzen Eigenschaften und darauf definierte Operationen (Methoden)
- Objekte kapseln (verbergen) interne Eigenschaften und auch Methoden, die der Benutzer nicht kennen muss
- Jedes Objekt ist ein Exemplar einer Klasse
- Klassen sind durch Vererbungsbeziehungen miteinander verbunden

Zusammenfassung

3. Objektorientierte Programmierung - Überblick

Beispiele:

a.) Zeichenprogramm mit geometrischen Elementen (Figuren)

- Zu zeichnende Figuren (Linien, Kreise, usw.) sind durch Objekte repräsentiert.
- Jedes Objekt hat Eigenschaften (z.B. Größe, Position) und "weiß", wie es zu zeichnen oder drucken ist.

b.) Sortieren

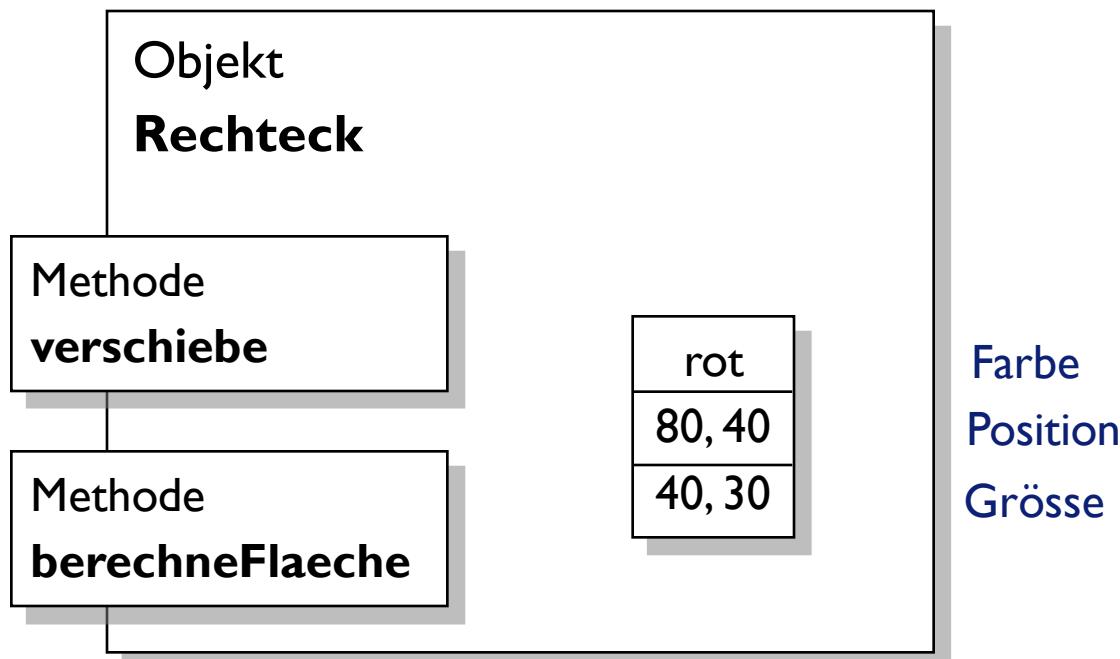
- Gegeben sei ein Feld von Objekten (Zeichenketten, Studenten, o. ä.)
- Jedes Objekt "weiß", wie es mit anderen Objekten verglichen werden kann
 - Zeichenkette: alphabetisch
 - Studenten: Vergleich der Matr.-Nr.

Zusammenfassung

4. Objekt

- Repräsentiert ein "Ding" (physikalisch oder konzeptionell)
- Hat einen Zustand: Wert der Objekt-Variablen (Attribute) zu einem Zeitpunkt
- Hat ein Verhalten: Menge der Methoden (Funktionen), über die Objekt verfügt
- Hat eine Identität: Eigenschaft durch die sich das Objekt von anderen auch bei gleichem Zustand unterscheidet → Speicherort.

Beispiel:



Zusammenfassung

4. Objekt

Interaktion zwischen Objekten durch Austausch von Nachrichten

Beispiel:

Nachricht an Rechteck Nr. 3:

"Verschieben um 3mm nach rechts und 5mm nach oben"

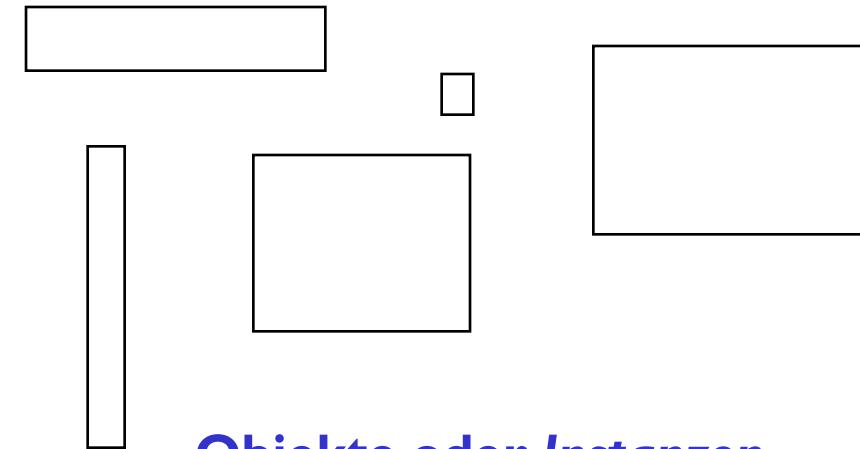
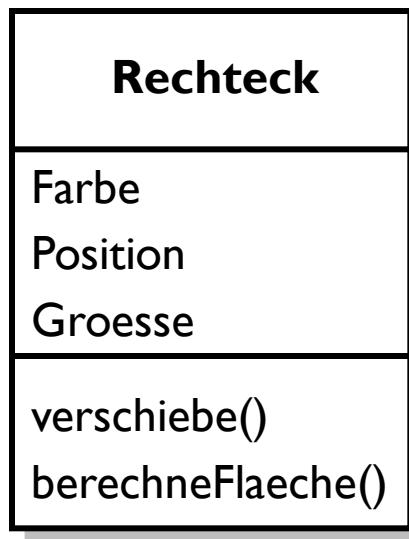
Praktisch:

- Aufruf der Methode "verschiebe()" des Objekts Rechteck
- Änderung des Zustandes Position

Zusammenfassung

5. Klasse

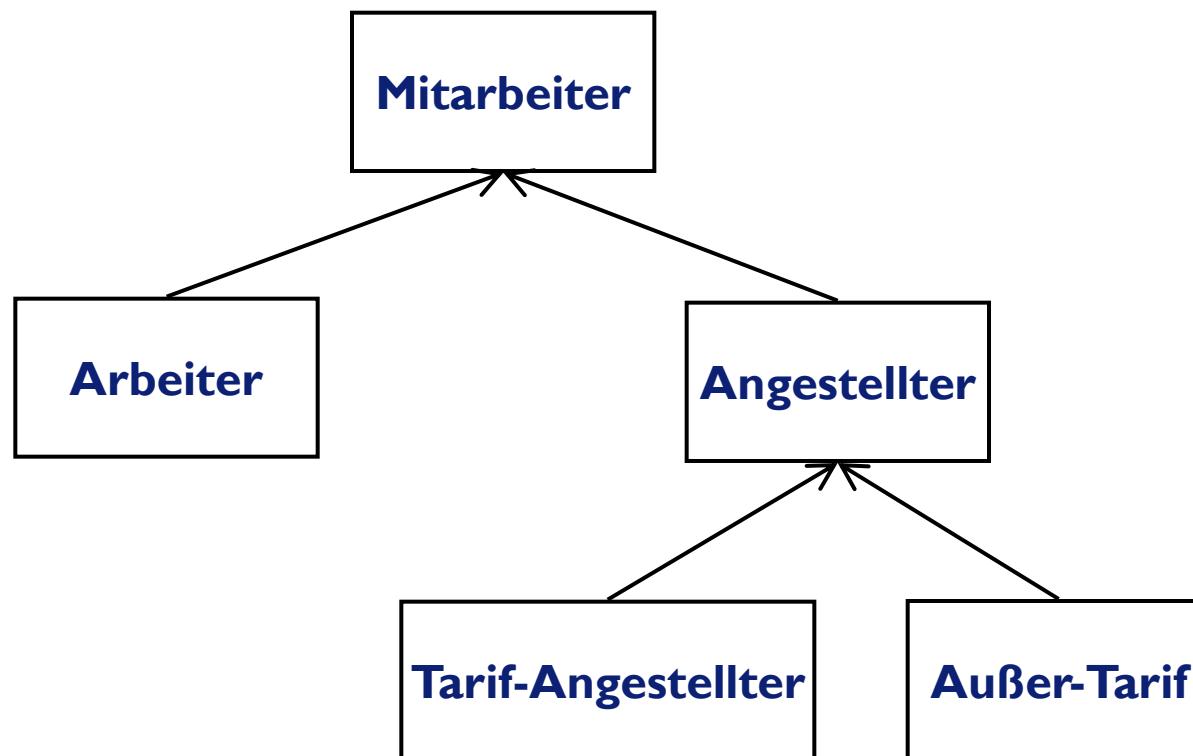
- Programmierung von Objekten bedeutet:
 - Vereinbarung von Variablen (Attributen)
 - Implementierung von Funktionen (Methoden)
- Wird nicht für jedes Objekt einzeln durchgeführt, sondern für dessen Klasse
- Eine Klasse fasst die gemeinsamen Eigenschaften und das gemeinsame Verhalten einer Menge von gleichartigen Objekten zusammen.
- Eine Klasse ist eine Art Schablone oder Vorlage für die Erzeugung von Objekten



Zusammenfassung

6. Vererbung

- Erweiterung existierender Klassen
- Hinzufügung neuer Eigenschaften und Methoden
- Abgeleitete Klassen können wieder erweitert werden → Vererbungshierarchie



Zusammenfassung

7. Vorteile der objektorientierten Programmierung

- **Kapselung, klare Schnittstellen:** Anwender von Objekten müssen nur dessen Schnittstelle kennen und keine internen Details der Implementierung
- Bessere Modularität, bessere Strukturierbarkeit besonders bei großen Systemen
- Leichte Wartbarkeit
- Leichte Erweiterung / Weiterentwicklung, hohes Mass an Wiederverwendbarkeit
- Möglichkeit zu ggf. hierarchischen Klassenbeziehungen
 - Teile-Ganzes-Beziehung („hat ein“)
 - Oberbegriff-Beziehung („ist ein“) → Vererbung, Polymorphie

Übersicht (1)

Objektorientierte Programmierung (OOP):

- Einführung
- Objekt und Klasse
- Attribute, Methoden
- Geheimnisprinzip, Kapselung
- Vererbung
- Klassifikation
- Beziehungen zwischen Klassen
- Polymorphie

Schnellkurs C++:

- Einführung
- **Bekannte Sprachmittel:**
 - Variablen, Datentypen, Operatoren, Kontrollstrukturen, Arrays, Strukturen
- **Neue Sprachmittel:**
 - Referenzen
 - Funktionen: Vorgabeargumente, Überladung, Templates
 - Namensräume
 - Ein- und Ausgabe
 - Strings
 - Typumwandlung

Objektorientierte Programmierung mit C++:

- Klassen, Vererbung, Polymorphie, Mehrfachvererbung

C++ Schnellkurs

Entstehung von C++

Design und Implementierung: Bjarne Stroustrup

Homepage: <http://www2.research.att.com/~bs/homepage.html>



- C++ ist eine objektorientierte Programmiersprache
- Wurde ab 1979 als Erweiterung der Programmiersprache C entwickelt
- Erste Version hieß „C with classes“, erst 1983 entstand der Name C++
- 1985 erscheint die erste kommerzielle Version von C++
- 1990 erscheint das Buch „The Annotated C++ Reference Manual“
- 1998 erfolgt (ANSI / ISO) Standardisierung (C++98); Vorteile:
 - Sprache ist an kein Entwicklungstool oder Betriebssystem gebunden
 - Standardbibliothek mit Ein-/Ausgabe Werkzeugen bei jedem Entwicklungstool gleich
- 2003: Nachbesserung der Norm von 1998 (C++03); 2011: C++11
- Weit verbreitet in Systemprogrammierung, Anwendungsprogrammierung
- Stroustrup: „Mehrere Millionen Anwender weltweit“

Entwurfsziele von C++

- Effizienz (Code und Laufzeit)
- Größere Typsicherheit
- Vernünftige Strukturierung großer Programme
 - Schutz der Daten durch Einschränkung der Zugriffsrechte
 - Klare Trennung zwischen Benutzerschnittstelle und Implementierung
 - Leichte Änderbarkeit von Programmteilen ohne Einfluß auf Gesamtsystem
 - Leichte Wiederverwendbarkeit, Erweiterbarkeit und Anpaßbarkeit des Codes
- Kompatibilität zu C
 - „C++ ist besseres C“ (B. Stroustrup); C ist eine Untermenge von C++
 - Gut geschriebene C Programme sind meist auch zulässige C++ Programme
 - Alle Unterschiede zwischen C und C++ können vom Compiler erkannt werden
- Objektorientiert
- C++: Prozedurale und objektorientierte Programmierung möglich

Einsatzmöglichkeiten von C++

(Unvollständige) Liste von Anwendungen, die in C++ geschrieben wurden:

<http://www2.research.att.com/~bs/applications.html>

Beispiele:

- **Amazon** : Large-scale e-commerce software
- **Google** : Suchmaschine etc.
- **Microsoft** : Windows XP, NT, MS Office, Explorer, Visual Studio
- **Siemens** : Medizinsysteme
- **Philips** : Medizinsysteme, Forschungssoftware
- ...

Vergleich von C++ und Java (1)

C++:

Compiler: Quelltext wird kompiliert (übersetzt in Maschinencode) auf dem Rechner des Entwicklers:

- + Optimale Leistung
- Wenn das Programm auf unterschiedlichen Betriebssystemen (Linux, Windows, Mac) oder Rechnern (Sun, SGI, AMD) laufen soll, muss man auch unterschiedliche Versionen verwalten

Java:

Interpreter: Java Byte-Code wird während der Laufzeit auf dem Rechner des Nutzers interpretiert und ausgeführt:

- Langsamer als compilierte Programme
- + Der Byte-Code kann (ohne Weiteres) auf andere Architekturen portiert werden

Vergleich von C++ und Java (2): Anwendungsspektrum

C++:

- Maschinennahe Hardware-Programmierung
- Betriebssysteme / Netzwerke
- Rechenzeit-intensive Bibliotheken und Anwendungen
- ...

Java:

- Webseiten
- Security-Anwendungen (Home-Banking)
- (E-)Learning
- Mehr und mehr auch für Rechenzeit-intensive Anwendungen
- ...

C Code, der kein C++ Code ist

Probleme, wenn in C-Code C++-Schlüsselworte verwendet werden

```
int main (void)
{
    int class = 2;      // Schluesselwort class verwendet
    // ...
}
```

C++-Schlüsselworte, die keine C-Schlüsselworte sind:

```
and, catch, explicit, namespace, or_eq, template, typename,
and_eq, class, export, new, private, this, using, asm,
compl, false, not, protected, throw, virtual, bitand,
const_case, friend, not_eq, public, true, wchar_t, bitor,
delete, inline, operator, reinterpret_cast, try, xor, bool,
dynamic_cast, mutable, or, static_cast, typeid, xor_eq
```

Alle Details zur Kompatibilität: s. Anhang B in Stroustrup's C++ Buch

Schlüsselworte in C++

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

C Code, der kein C++ Code ist

Probleme bei Zuweisung eines `void*` - Zeigers an Zeiger eines anderen Typs

- in C: ohne „cast“ erlaubt; in C++: explizite Typumwandlung erforderlich)

```
int main (void)
{
    // kompiliert in C, aber nicht in C++
    void* ptr;
    int *i = ptr; // implizite Umwandlung void* → int*
    int *j = malloc( sizeof(int) * 5 ); // ditto

    // kompiliert in C++
    int *i = (int *) ptr;
    int *j = (int *) malloc( sizeof(int) * 5 );

    ...
}
```

Siehe auch „Typumwandlung“

Siehe auch <http://david.tribble.com/text/cdiffs.htm>

C++ Schnellkurs

Ziel: Einführung in die wichtigsten allgemeinen Sprachelemente
Aber: noch keine objektorientierte Programmierung

Compiler und Linker

Code kann in einzelne Quellcode-Dateien (a.cpp, b.cpp, ...) aufgeteilt werden:



Modularisierung

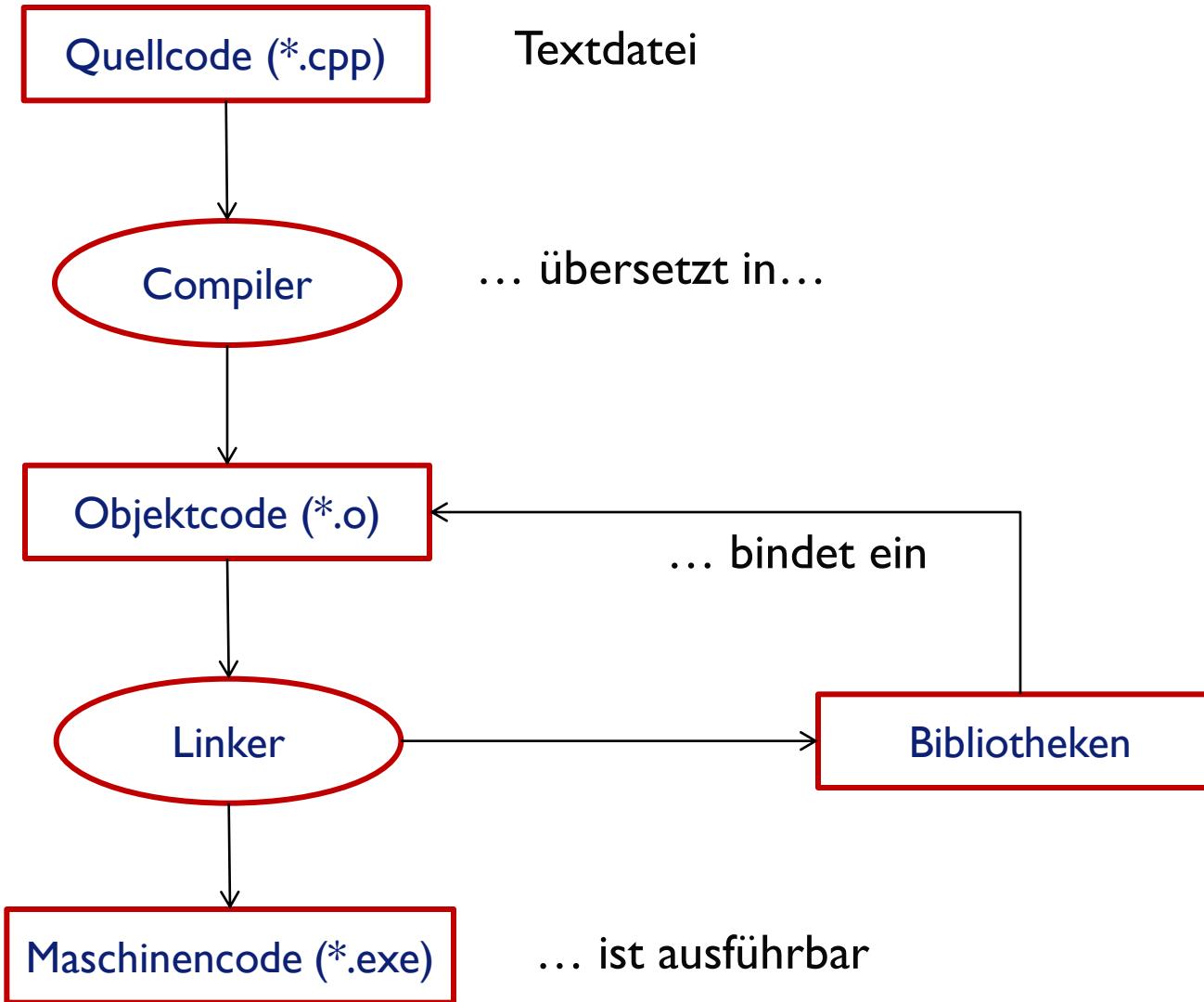
- **Compiler** macht aus jeder Quelldatei eine binäre Objektdatei.
- **Linker** (engl.: to link, verbinden) bindet die zum Programm gehörenden Objektdateien mit den Bibliotheksmodulen zusammen



ausführbare Datei

Das erste C++-Programm

Vom Programm zur ausführbaren Datei



Quelle: RRZN

Bibliotheken

- ... enthalten Hilfsmittel für die Programmierung
- ... ist eine Datei, die verschiedene Funktionen bereitstellt
- ... sind Werkzeugkästen zu verschiedenen Themen
- ... können im Quellcode (eigene Bibliotheken) oder im Maschinencode (fremde Bibliotheken) vorliegen

Das erste C++ Programm

C++ Programme bestehen aus:

- Präprozessor-Direktiven
- Eintrittsfunktion
- Anweisungen
- Deklarationen und Definitionen
- Kommentaren
- Funktionen (später)
- Namensräumen (neu in C++; später)

Das erste C++-Programm

```
*****  
*                                     // mehrzeiliger Kommentar  
* Hallo Welt Programm  
*  
* Autor : ...  
*  
*****  
#include <iostream>          // Praeprozessor-Direktiven  
#include <string>           // Header der String-Klasse einbinden  
  
using namespace std; // Wahl des Namensraumes (Details spaeter)  
  
int main(void)          // Definition der Eintrittsfunktion  
{  
    string gruss;         // Variablendefinition  
    gruss = "Hallo Welt\n"; // Anweisung  
    cout << gruss;        // Anweisung  
    system("PAUSE");      // C++ - Ausgabe; statt printf("%s\n", gruss);  
  
    return 0;              // Anweisung  
}
```

neue C++ - Klasse `string`: statt `char[]` bzw. `char*`

C++ - Ausgabe; statt `printf("%s\n", gruss);`

Präprozessor-Direktiven

- Anweisungen an den Compiler, die er ausführt, bevor er übersetzt
- Beginnen immer mit einem Hash (#)
- Stehen immer am Anfang eines Programms
- Pro Zeile wird eine Präprozessor-Anweisung geschrieben
- Enden nicht mit einem Semikolon

Präprozessor-Direktiven: include

- Wichtigste Direktive: #include
 - kopiert den Inhalt einer Textdatei in die aktuelle Datei
 - wird benutzt, um über Header-Dateien Deklarationen von verwendeten Code-Elementen (z.B. Funktionen oder Typen) einzukopieren.
- Einbindung eigener Header-Dateien oder verfügbarer Bibliotheken:

`#include "abc.h"`: bindet Datei abc.h aus aktuellem Verzeichnis ein

`#include <iostream>`: bindet iostream-Bibliothek ein

- iostream enthält Funktionen für die Ein- und Ausgabe

- Header einbinden → Funktionen müssen nicht einzeln deklariert werden

Eintrittsfunktion: main()

- ...ist bei jedem C++-Programm der Startpunkt der Programmausführung
- ...muß einmal in einem C++-Programm vorhanden sein
- Rückgabetyp: int
- Parameterliste: Compiler-abhängig, die meisten Compiler akzeptieren
 - int main (void)
 - int main ()
 - int main (int argc, char *argv[])

Beispiel für die Verwendung der Parameter: s. nächste Seite

Das erste C++-Programm

Eintrittsfunktion

Beispiel für die Verwendung der Parameter

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

int main(int argc, char *argv[])
{
    printf("Anzahl der Argumente: %d\n", argc);
    for (int i=0; i<argc; i++)
        printf("Argument %d: %s\n", i+1, argv[i]);
    system("PAUSE");
    return 0;
}
```

in diesem Beispiel
nicht erforderlich

Aufruf mit Parameterliste:

12 hallo

Ausgabe?

Eintrittsfunktion

Beispiel für die Verwendung der Parameter

Ausgabe:

Anzahl der Argumente: 3

Argument 1: D:\Dev-Cpp\Projekt2.exe

Argument 2: 12

Argument 3: hallo

Drücken Sie eine beliebige Taste . . .

Anweisungen

- ... enden immer mit einem Semikolon
- ... beschreiben einen Arbeitsschritt in C++
 - Zuweisungen, Kontrollanweisungen (Schleifen, Bedingungen), Funktionen...

```
int main(int argc, char *argv[])
{
    ...
    gruss = "Hallo Welt\n";           Zuweisung
    cout << gruss;                  Ausgabe (Funktion)
    ...
    return 0;                      Rückgabe an
                                    Betriebssystem
}
```

Definition und Deklaration

Die wichtigsten vom Programmierer definierbaren Elemente sind

- Variablen zum Zwischenspeichern von Daten
- Funktionen zur Lösung von Aufgaben / Teilproblemen
- Datentypen zur Repräsentation neuer Arten von Daten

Was ist der Unterschied zwischen Deklaration und Definition?

- **Deklaration:** macht Compiler mit Namen (Bezeichner) und Typ bekannt
 - Typ bestimmt die zulässigen Aktionen (z.B. arithmet. Operationen)
 - Keine Anlegung von Speicher / Erzeugung von Maschinencode!
- **Definition:** Anlegung eines Speicherobjekts im Programm/ Maschinencode
 - Variable: Speicher für den Wert der Variable
 - Funktion: Speicher u.a. für Maschinencode (kompil. Anweisungsteil)
- Jede Definition ist auch eine Deklaration; Umkehrung gilt nicht immer

Das erste C++-Programm

Definition und Deklaration

Beispiel:

Eine Variable **var** soll in zwei verschiedenen Quelldateien verwendet werden.

datei1.cpp

```
int variable;      // Definition der globalen Variablen

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

datei2.cpp

```
int variable;      // Definition der Variablen

void funktion_eins ( ... ) { ... }
```

Linker beschwert sich
über mehrere Varianten
dieselben Elementes

Das erste C++-Programm

Definition und Deklaration

Beispiel:

Lösung:

Mit Hilfe des Schlüsselwortes **extern** wird variable in datei2.cpp **deklariert**.

datei1.cpp

```
int    variable;           // Definition der globalen Variablen

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

datei2.cpp

```
extern int    variable; // Deklaration der Variablen

void funktion_eins ( ... ) { ... }
```

Das erste C++-Programm

Definition und Deklaration

Beispiel zu Funktionen:

Eine Funktion soll in zwei verschiedenen Quelldateien verwendet werden.

datei1.cpp

```
void funktion_eins (int parameter);      // Funktionsdeklaration

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

datei2.cpp

```
void funktion_eins (int parameter)      // Funktionsdefinition
{ ... }
```

Das erste C++-Programm

Definition und Deklaration

- C: Deklaration von Variablen am Anfang des Programm
- C++: Deklaration von Variablen an jeder Stelle im Programm möglich
 - Variablen sind ab dem Zeitpunkt der Deklaration bekannt
(beachte jedoch den Gültigkeitsbereich!)

```
void f();

int main(int argc, char *argv[])
{
    f();                                // Funktionsaufruf
    int a[4];                            // Deklaration und Definition
    for ( int i=0; i<4; i++ )           // Deklaration und Definition
    {
        a[i] = i;
    }
    return 0;                            //... hier ist i nicht mehr sichtbar
}
```

Kommentare

- Kommentare dienen dem besseren Verständnis
- Einzeilige Kommentare beginnen mit //

 - Nachfolgender Text wird nicht vom Compiler gelesen

- Ein- oder mehrzeilige Kommentare auch mit /* ... */ (wie in C)

```
int main(int argc, char *argv[])
{
    int a,b; // Seitenlängen
    // Berechnung der Fläche
    int s = a*b;
    /* Alternative
    ...
    */
    return 0;
}
```

Layout eines Programms

Schlechtes Beispiel:

```
#include <iostream>
using namespace std;
int main(int argc,char *argv[]){cout<<"Hallo Welt";return 0;}
```

Warum ist dieses Beispiel unbefriedigend?

Was kann besser gemacht werden?

Das erste C++-Programm

Layout eines Programms

Ziel: bessere Übersichtlichkeit für den Programmierer
→ Leichteres Suchen von Fehlern

Einige Grundregeln:

- Pro Zeile eine Anweisung
- Zusammenhängende Blöcke gemeinsam einrücken, übersichtlich klammern
- Leerzeichen für bessere Übersichtlichkeit
- Kommentare zur Gliederung

The diagram illustrates the layout of a C++ program. On the left, blue annotations explain the code structure. A bracket labeled "Klammern" points to the opening brace of the main function. Another bracket labeled "2-4 Zeichen eingerückt" points to the opening brace of the main function, indicating the indentation level. Blue arrows point from these annotations to the corresponding parts of the code.

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    cout << "Hallo Welt";
    return 0;
}
```

C++ Schnellkurs:

Grundkonzepte C++

Grundkonzepte C++

- Variable
- Datentypen, Typumwandlung
- Operatoren
- Kontrollstrukturen
- Funktionen
- Arrays, Aufzählungen, Strukturen
- Zeiger

Die meisten dieser Grundkonzepte sind in C++ ähnlich wie in C. Einige Unterschiede (z.B. Typumwandlung) werden erläutert.

Variablen und Datentypen

Programme verarbeiten Daten, daher kommt der Datenrepräsentation eine besondere Bedeutung zu.

Grundbegriffe:

Literal: Zeichenfolge zur Darstellung der Werte von Basistypen

- Nicht benannt; Verwendung i.a. in rechtsseitigen Ausdrücken
- Bsp: 10 -1200 12.34 'a' "hallo" 12.e-34

Variable: Zwischenspeicher für Daten, Behälter für einen Wert

- Hat Namen (Bezeichner), Adresse im Hauptspeicher und Typ
- Können verändert werden

Konstante: Wert, der *nicht* verändert werden kann (Ggs. zu Variable)

- Meist benannt (unbenannte Konstante → Literal)

Variablen

Variable: Zwischenspeicher für Daten

- **Variablenname:**
 - Symbolisiert einen Wert, der im Speicher abgelegt ist
 - Ist ein Platzhalter
- **Variablenwert:**
 - Wird an einer bestimmten Speicherstelle abgelegt
 - Ist veränderbar
- **Datentyp:**
 - Legt das Format des zu speichernden Wertes fest
 - Legt die Verarbeitungsmöglichkeiten des Wertes fest
- Definition, Zuweisung von Werten und Verwendung: Wie in C
 - `int i; float a = 3.0; char zeichen = 'a';`
- Ort der Variablendefinition legt Gültigkeitsbereich und Lebensdauer fest

Variablen: Gültigkeitsbereich, Lebensdauer

1. Lokale Variablen

- Variablen, die in Funktionen definiert sind
- Erzeugung beim Aufruf der Funktion
- Löschen beim Verlassen der Funktion

2. Variablen in Klassen

- Variablen, die als Elemente von Klassen definiert sind
- Lebensdauer ist an die Lebensdauer der Klasse/Objekte gebunden
→ Näheres später im Kapitel über Klassen

3. Globale Variablen

- Variablen, die außerhalb von Fkt. und Klassen definiert sind
- Im gesamten nachfolgenden Code verfügbar
- Werden erst beim Beenden des Programms aufgelöst

Spezielle Variablen

Durch bestimmte Schlüsselwörter können Variablen mit besonderen Eigenschaften definiert werden. Die wichtigsten sind:

- **const** (→ Konstante)

- Beispiele:

```
const float pi = 3.1415927;
```

```
const char dquote = '\"';
```

- **static**

- Beispiele:

```
static float wert;
```

```
static int zaehler;
```

Spezielle Variablen: Konstante

Realisierung durch Schlüsselwort „**const**“:

- Mit vorangestelltem Schlüsselwort **const** definierte Variablen behalten ihren Wert bei, können also nicht verändert werden.
- Müssen bei der Definition bereits initialisiert werden.

Beispiel für einen Fehler:

```
int main(void)
{
    const int a; <----- Error here
    a = 12; <----- Error here
    printf("%d\n", a);

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Compiler-Ausgabe:

- **uninitialized const 'a'**
- **assignment of read-only variable 'a'**

Spezielle Variablen: Static

Schlüsselwort „static“ hat je nach Variablentyp unterschiedliche Auswirkung:

A. Lokale Variablen

- Mit vorangestelltem Schlüsselwort static definierte lokale Variablen werden beim ersten Aufruf ihrer Funktion erzeugt und bleiben bis zum Programmende - auch über mehrere Funktionsaufrufe hinweg - erhalten.
- Anwendung, um Funktionswerte über das Ende der Funktion hinweg zu speichern.

Spezielle Variablen: Static

Schlüsselwort „static“ hat je nach Variablentyp unterschiedliche Auswirkung:

B. Variablen in Klassen

Wird hier nur der Vollständigkeit halber mit aufgeführt - muss noch nicht verstanden werden → Näheres später im Kapitel über Klassen

Als static deklarierte Membervariablen existieren nur ein Mal pro Klasse und werden nicht wie die anderen Membervariablen für jedes neue Objekt neu angelegt.

Spezielle Variablen: Static

Schlüsselwort „static“ hat je nach Variablentyp unterschiedliche Auswirkung:

C. Globale Variablen

Als static deklarierte globale Variablen sind nur innerhalb ihrer Code-Datei (.cpp-Datei) gültig und nicht in anderen Quelltextdateien des Programms.

Beispiel:

Jede Variable **global** wird exklusiv von den
in der Datei definierten Funktionen verwendet

```
// Datei 1:  
static int global;
```

```
// Datei 2:  
static int global;
```

Dieser Gebrauch von „static“ sollte vermieden werden

→ stattdessen: (unbenannte) Namensbereiche verwenden



Datentypen

- Sind Baupläne für die Art der Variablen
- Geben über das Format des gespeicherten Wertes Auskunft
- Legen Regeln für die Interpretation und Verwendung eines Wertes fest
- Legen den Speicherbedarf für eine Variable fest
- Sind in C++ für Ganz- und Dezimalzahlen sowie einzelne Zeichen definiert

Kategorien von Datentypen in C++:

- einfache Datentypen, die fester Bestandteil der Sprache sind
- komplexe Datentypen aus der Standardbibliothek
- komplexe selbst definierte Datentypen

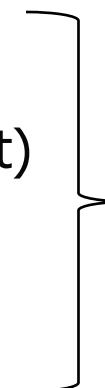
Datentypen

Einfache Datentypen

- sind fest in die Sprache integriert
- besitzen ein eigenes Schlüsselwort

Wichtige einfache Datentypen:

- **char**
- **int, short** (i.e. `short int`), **long** (i.e. `long int`)
 - Jeweils **signed / unsigned**
- **float, double, long double**
- **bool** (neu in C++ bzw. ab C99 als `_Bool`)



typ.Wertebereiche:
wie in C

Datentypen

Einfache Datentypen: „bool“

- Mögliche Werte (Literale): `true`, `false`
- Drückt Wahrheitswerte aus

```
bool flag;  
...  
flag = true;      // Schalter auf wahr setzen  
...  
flag = false;     // Schalter auf falsch setzen
```

- In arithmetischen und logischen Ausdrücken wird `bool` zu `int` konvertiert:

`bool` → `int`

`false` → 0

`true` → 1

`int` → `bool`

0 → `false`

≠0 → `true`

Datentypen

Einfache Datentypen: Operatoren

- Einfache Datentypen besitzen eigene Operatoren

Beispiele:

`bool: &&, ||, !`

`int: +, -, *, /, %, ++, --, <, <=, >=, >, ==, !=,`
`&, |, ^, ~, >>, <<`

- Operatoren: siehe später

Datentypen

Komplexe vordefinierte Datentypen

Zur Speicherung komplexerer Daten gibt es in C++ eine Reihe vordefinierter Klassen in der Standardbibliothek.

Bemerkung: Wir sprechen hier bereits von Klassen statt Datentypen, weil zusätzlich zu den Daten auch Funktionen zur Verarbeitung der Daten bereitgestellt werden. Näheres später im Kapitel über Klassen.

Beispiele:

- string: Für die Arbeit mit Zeichenfolgen. Header-Datei: <string>
- complex: komplexe Zahlen, Header-Datei: <complex>
- vector: Vektoren, Header-Datei <vector>

Datentypen

Komplexe selbstdefinierte Datentypen

- Funktioniert wie in C
- Möglichkeiten sind z.B. Aufzählungen, Arrays oder Strukturen

Datentypen: Typumwandlung

Man unterscheidet zwischen impliziter und expliziter Typumwandlung:

a. Implizite Typumwandlung

- Wird vom Compiler automatisch vorgenommen
- Betrifft Umwandlungen zwischen
 - einfachen Datentypen
 - einem Klassentyp und einem abgeleiteten Klassentyp (s. später)

Datentypen: Typumwandlung

a. Implizite Typumwandlung

Beispiel:

```
double temperatur = 23;
```

Umwandlung des Integer-Literal
als 23 in das double-Literal 23.0

```
int offset = 2;
```

```
double rand;
```

```
rand = 2.5 + offset;
```

Umwandlung der Integer-Variablen offset in double Datentyp.
Damit findet Angleichung an den anderen Operanden statt.

Datentypen: Typumwandlung

b. Explizite Typumwandlung

Vom Programmierer ausdrücklich (explizit) gewünschte Umwandlung

Operatoren:

Es stehen 5 Cast-Operatoren zur Verfügung

- universell einsetzbarer Operator: ()
- 4 für spezielle Konvertierungen einsetzbare neue C++-Operatoren:

`static_cast<Ziel-Typ> (Variable)`

`const_cast<Ziel-Typ> (Variable)`

`reinterpret_cast<Ziel-Typ> (Variable)`

`dynamic_cast<Ziel-Typ> (Variable)`

Ausführung zur
Übersetzungszeit

Ausführung zur Laufzeit

Datentypen: Typumwandlung

- **(Ziel-Typ) Variable**

Wurde von Vorgänger-Sprache C übernommen und wird gerne wegen der einfachen Syntax verwendet.

Einsatzgebiet: Universell einsetzbar, wo Konvertierung möglich ist.

Beispiele:

```
erg = (double) n1 / n2;  
ptr = (int *) malloc(30 * sizeof(int));
```

Es wird jedoch sehr empfohlen, die **neuen C++-Operatoren** zu verwenden:

- Erleichterte Text-Suche
- Unterscheiden zwischen verschiedenen Anwendungsfällen
- Größere Sicherheit wegen besserer Überprüfbarkeit durch den Compiler

Datentypen: Typumwandlung

- `static_cast<Ziel-Typ> (Variable)`
- Verändert die Daten selbst
- Konvertierung wird **zur Übersetzungszeit geprüft**

Einsatzgebiet:

Typumwandlungen zwischen Datentypen, die auch der Compiler implizit ineinander umwandeln dürfte (z.B. `int` → `double`).

typische Anwendungsbeispiele:

- Erzwingung von Gleitkomma-Divisionen für `int`-Operanden
- Umwandlung eines `void`-Zeigers in z.B. `int`-Zeiger

Datentypen: Typumwandlung

- `static_cast<Ziel-Typ> (Variable)`

Beispiel:

```
int n1=2, n2=3;  
  
double erg;  
  
erg = n1/n2;  
  
cout << erg << endl;
```

Ausgabe?

→ 0

Problem: Ganzzahl-Division ergibt wieder eine ganze Zahl. D.h. vom korrekten Ergebnis 2/3 wird der Nachkommaanteil abgeschnitten.

Datentypen: Typumwandlung

- `static_cast<Ziel-Typ> (Variable)`

Beispiel:

```
int n1=2, n2=3;  
  
double erg;  
  
erg = n1/n2;  
  
cout << erg << endl;
```

Lösung?

Datentyp eines Operators in double umwandeln. Dann wird der zweite Operator angepasst und ebenfalls umgewandelt.

Datentypen: Typumwandlung

- `static_cast<Ziel-Typ> (Variable)`

Beispiel:

```
int n1=2, n2=3;  
  
double erg;  
  
erg = static_cast<double>(n1)/n2;  
  
cout << erg << endl;
```

Datentypen: Typumwandlung

- `static_cast<Ziel-Typ> (Variable)`

anderes Beispiel: Zeiger-Umwandlung

```
int *ptr;  
  
ptr = malloc(30*sizeof(int));
```

Problem?

Compilerfehler:

```
invalid conversion from 'void *' to 'int *'
```

Datentypen: Typumwandlung

- `static_cast<Ziel-Typ> (Variable)`

anderes Beispiel: Zeiger-Umwandlung

```
int *ptr;  
  
ptr = malloc(30*sizeof(int));
```

Lösung?

→ Cast auf int-Zeiger

Wie sieht das aus?

Datentypen: Typumwandlung

- `static_cast<Ziel-Typ> (Variable)`

anderes Beispiel: Zeiger-Umwandlung

```
int *ptr;
```

```
ptr = static_cast<int *>( malloc(30*sizeof(int)) );
```

Lösung?

→ Cast auf int-Zeiger

Wie sieht das aus?

Datentypen: Typumwandlung

- `const_cast<Ziel-Typ> (Variable)`
- Bewirkt dasselbe wie `(Ziel-Typ) Variable`, sofern sich der Ziel-Typ und der Typ von Variable höchstens um ein `const` unterscheiden
- Konvertierung wird [zur Übersetzungszeit](#) geprüft

Einsatzgebiet:

Entfernt eine const-Deklaration aus einem Typ

Beispiel für eine Anwendung: nächste Seite

Grundkonzepte C++

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

void f(int *p) {
    cout << *p << endl;
}

int main(void) {
    const int a = 10;
    const int *b = &a;

    f(b);

    return 0;
}
```

Was sagt der Compiler dazu?

invalid conversion from 'const int*' to 'int *'

Grundkonzepte C++

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

void f(int *p) {
    cout << *p << endl;
}

int main(void) {
    const int a = 10;
    const int *b = &a;

    f(b);

    return 0;
}
```

Lösung?

Grundkonzepte C++

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

void f(int *p) {
    cout << *p << endl;
}

int main(void) {
    const int a = 10;
    const int *b = &a;

    int *c;
    c = const_cast<int *>(b);
    f(c);

    return 0;
}
```

Kürzer:

```
f( const_cast<int *>(b) );
```

Mögliche Lösung:

- (Ggf. neue Variable c einführen, die nicht konstant ist)
- **const_cast auf b anwenden, um die Eigenschaft const zu eliminieren**
- (Ergebnis nach c kopieren)

Ohne den Cast (d.h. würde man `c = b;` schreiben) gäbe es eine Fehlermeldung:

```
invalid conversion from 'const int*' to 'int *'
```

Besser: Konsistenter Code

- Argument von `f` als konstanter Zeiger
- Oder `b` als nicht konstanter Zeiger

Datentypen: Typumwandlung

- `reinterpret_cast<Ziel-Typ> (Variable)`
- Verändert die Art, wie Daten an einer bestimmten Adresse gelesen („interpretiert“) werden
- Bitmuster des Originaltyps bleibt erhalten und wird nur neu interpretiert
- Konvertierung wird [zur Übersetzungszeit](#) geprüft

Einsatzgebiet:

Umwandlung nicht verwandter Typen

Konvertierung wird ohne jegliche Prüfung vorgenommen und ist daher besonders riskant (nur in hardwarenahem Code zu empfehlen...)

Datentypen: Typumwandlung

- `dynamic_cast<Ziel-Typ> (Variable)`
- Prüft zur Laufzeit, ob gegebener Zeiger auf gewünschten Objekttyp zeigt
 - Falls nicht: Null-Pointer wird zurückgeliefert
- Dadurch wird zur Laufzeit sichergestellt, daß die Objekte, mit denen gearbeitet wird, auch vom erwarteten Typ sind

Einsatzgebiet:

Umwandlung zwischen Klassentypen einer gemeinsamen
Klassenhierarchie → siehe später

Operatoren

(Keine wesentlichen Unterschied zwischen C und C++)

Operatoren werden unterschieden nach

- Anzahl der Operanden
 - unär (ein Operand)
 - binär (zwei Operanden)
 - ternär (drei Operanden)
- Stellung des Operators in einem Ausdruck
 - infix (zwischen Operanden)
 - prefix (vor den Operanden)
 - postfix (nach den Operanden)
- Assoziativität des Operators
 - Links-Assoziativität
 - Rechts-Assoziativität

Operatoren (1)

- Arithmetische Operatoren (binär)
 - Operanden ganzzahlig oder reell: +, -, *, /
 - Operanden ganzzahlig: % (modulo)
- Inkrement- und Dekrement-Operatoren (unär, pre- oder postfix)
 - ++ (increment), -- (decrement)
- Zuweisung (binär)
 - =, +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=
- Bedingungsoperator (ternär)
 - ? : Bedingte Wertzuweisung
- Relationale Operatoren
 - <, <=, >, >=, ==, !=
- Logische Operatoren
 - && (logisches AND) , || (logisches OR), ! (logisches NOT)

Operatoren (2)

- Bitmanipulation(binär)
 - & (bitweises AND), | (bitweises OR), ^ (bitweises XOR), << (shift left), >> (shift right)
- Zeiger-Operatoren
 - & (Adressoperator, Adresse von), * (Verweisoperator, Wert an Adresse)
- Weitere bzw. zusätzliche Operatoren von C++:
 - . Dereferenzierung von Klassenelementen (auch in C)
 - -> Dereferenzierung von Zeigern auf Klassenelemente (auch in C)
 - :: Bereichsauflösung (Namensraum)
 - new Operator zur dynamischen Speicherverwaltung
 - typeid Typidentifikation zur Laufzeit
 - & Referenz

Operatoren: Assoziativitt

- **Rechtsassoziativer Operator:** wird von rechts nach links ausgewertet
 - Beispiel: $a = b = c$ gleichbedeutend mit $a = (b = c)$
 - Zuweisungsoperator, Bedingungsoperator, Inkrement, Dekrement, logisches NOT, bitweises NOT, unres Minus, unres Plus, Adresse von (&), Indirektion (*), sizeof, new, delete, Typumwandlung (type)
- **Linksassoziativer Operator:** wird von links nach rechts ausgewertet
 - Beispiel: $a - b - c$ gleichbedeutend mit $(a - b) - c$
 - brige Operatoren

Operatoren: Auswertungsreihenfolge

Höchste
Priorität



Niedrigste
Priorität

Operatoren	Assoziativität
() [] -> .	von links
! ~ + - - - * & (Typ) sizeof	von rechts
* / %	von links
+	von links
<< >>	von links
< <= > >=	von links
== !=	von links
&	von links
^	von links
	von links
&&	von links
	von links
? :	von rechts
= += -= *= /= %= &= ^= = <<= >>= ,	von rechts
	von links

Kontrollstrukturen

Auch bei den Kontrollstrukturen gibt es keine wesentlichen Unterschiede zwischen C und C++.

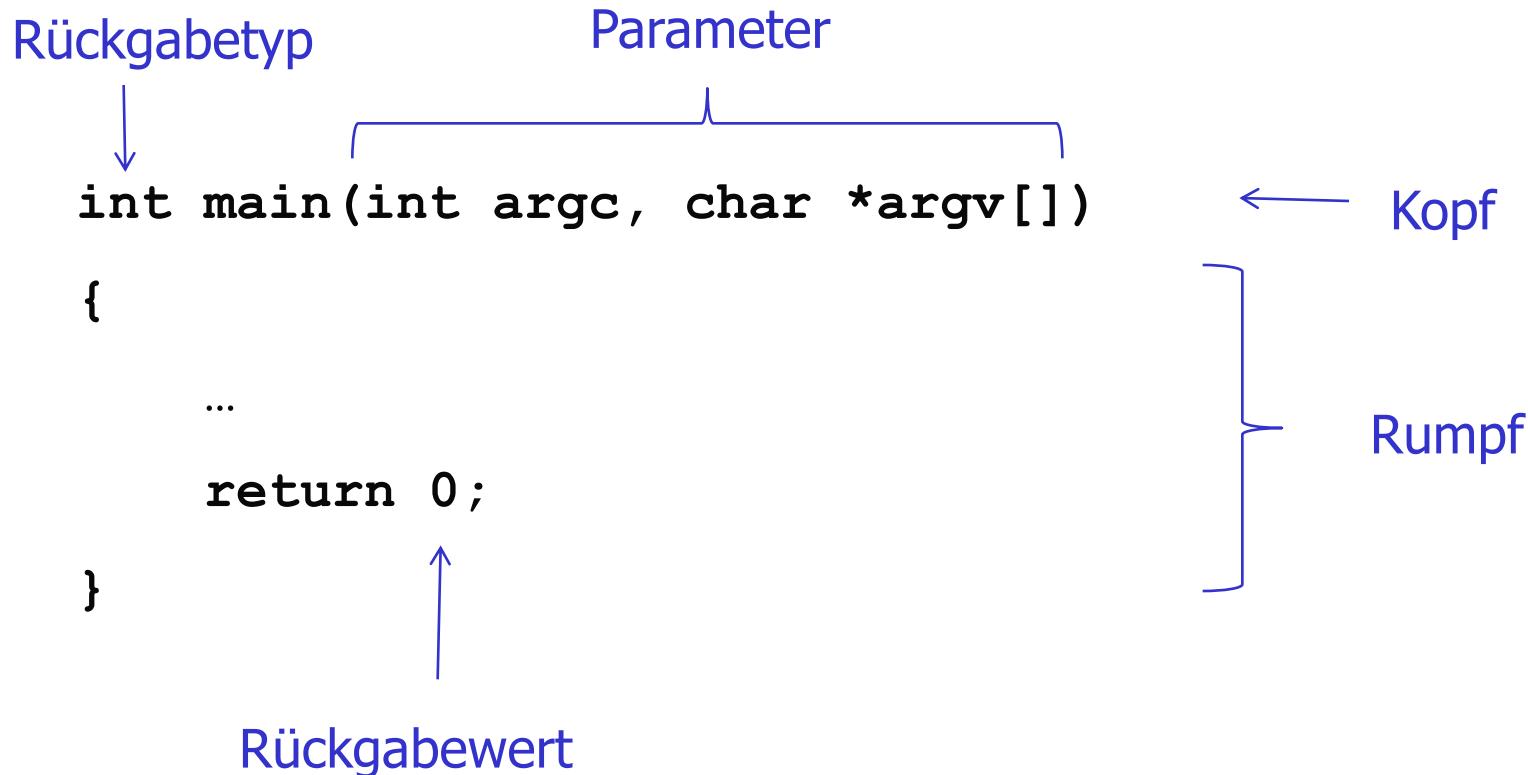
Es gibt

- Verzweigungen (if-else, switch)
- Schleifen (for, while, do-while)

wobei die Syntax aus C übernommen wurde.

Funktionen

- ... fassen Anweisungen, die eine Teilaufgabe erledigen, zusammen
- ISO C++ verlangt Rückgabetyp (z.B. **void**)



Funktionen

Kopf:

- Hat eindeutigen Namen, mit dem die Funktion aufgerufen wird
- Übergabe von Parametern: in runden Klammern
 - Falls keine Parameter übergeben werden: leere Klammern

Rumpf:

- Anweisungen / Programmcode innerhalb von geschweiften Klammern
(kennzeichnen den Anfang und das Ende eines Blocks von Anweisungen)

Rückgabewert:

- Rückgabe mit Schlüsselwort `return`

Funktionen

Definition und Verwendung von Funktionen weitgehend identisch zu C

- Aber: ISO C++ verlangt Rückgabetyp (z.B. **void**)
- Weitere Unterschiede werden im Verlauf der Vorlesung besprochen.

Zur Terminologie:

Rückgabetyp Funktionsname (Parameterliste)

{

Anweisungen

}

Frage: Was ist der Unterschied zwischen Parametern und Argumenten?

- **Parameter** sind die im Funktionskopf definierten Variablen
- **Argumente** sind die Werte der Parameter, die beim Funktionsaufruf übergeben werden.

Arrays, Aufzählungen, Strukturen

Ziel: Definition eigener Datentypen

Syntax und Verwendung dieser Elemente wie in C

1. Array

Dient der gemeinsamen Verwaltung mehrerer Variablen *eines* Datentyps

- Faßt Elemente desselben Datentyps zusammen
- ein- oder mehrdimensional

Beispiel:

```
#define ARRAY_SIZE 100

int main (void) {

    int array[ARRAY_SIZE];

    for (int i = 0; i<ARRAY_SIZE; i++) {

        array[i] = i;

    }
```

Arrays, Aufzählungen, Strukturen

2. Aufzählung

Datentyp, dessen mögliche Werte durch explizite Aufzählung festgelegt sind

Beispiel:

```
enum wochentag {montag, dienstag, mittwoch, donnerstag,  
                 freitag, samstag, sonntag};
```

Interne Repräsentation: durch Integer-Konstanten

- erstem Wert (hier 'montag') wird 0 zugewiesen
 - jedem Folgewert wird der Wert des Vorgängers +1 zugewiesen
- dienstag wird intern durch 1 repräsentiert, mittwoch durch 2 usw.

Arrays, Aufzählungen, Strukturen

2. Aufzählung

Integer-Präsentation kann ausdrücklich gesetzt werden:

Beispiel:

```
enum wochentag {montag=2, dienstag=4, mittwoch, donnerstag,  
                freitag, samstag, sonntag};
```

Grundkonzepte C++

```
enum wochentag {montag, dienstag, mittwoch, donnerstag,  
                freitag, samstag, sonntag};
```

```
int main (void) {  
    "enum" kann in C++ fehlen  
    wochentag tag = donnerstag;
```

```
switch (tag) {  
    case dienstag:  
        cout << "Dienstag" << endl;  
        break;  
    case mittwoch:  
        cout << "Mittwoch" << endl;  
        break;  
    default:  
        cout << "Weder Dienstag noch Mittwoch" << endl;  
}
```

```
cout << "Wert von tag = " << tag << endl;
```

```
return 0;
```

```
}
```

Ausgabe?

Weder Dienstag noch Mittwoch

Wert von tag = 3

Arrays, Aufzählungen, Strukturen

3. Strukturen

Zusammenfassung von Variablen unterschiedlicher Datentypen

- Schlüsselwort **struct**
- Elemente einer Struktur innerhalb geschweifter Klammern
- Anweisung wird mit Semikolon abgeschlossen
- Zugriff auf Strukturelemente („Membervariablen“): **Punktoperator**
 - Verbindet Strukturelement mit Strukturvariabler

Beispiel: Struktur für die Verwaltung von Personendaten

Grundkonzepte C++

Beispiel:

```
struct person {
```

```
    string name;
```

```
    string vorname;
```

```
    int geburtsjahr;
```

}

Strukturelemente

```
} ;
```

Semikolon

```
int main (void) {
```

"struct" kann in C++ fehlen

```
struct person dichter;
```

Punktoperator

```
dichter.name = "Rilke";
```

```
dichter.vorname = "Rainer Maria";
```

```
dichter.geburtsjahr = 1875;
```

```
cout << "Name = " << dichter.name << endl;
```

```
cout << "Vorname = " << dichter.vorname << endl;
```

```
cout << "Geburtsjahr = " << dichter.geburtsjahr << endl;
```

```
return 0;
```

}

Arrays, Aufzählungen, Strukturen

3. Strukturen

Die aus C übernommene Struktur wurde in C++ erweitert, so dass sie nun nicht nur Membervariablen, sondern auch Funktionen als untergeordnete Elemente enthalten kann.

```
struct person {
    string name;
    string vorname;
    int geburtsjahr;

    void ausgabe () {
        cout << "Name = " << name << endl;
        cout << "Vorname = " << vorname << endl;
        cout << "Geburtsjahr = " << geburtsjahr << endl;
    }
};

int main (void) {
    struct person dichter;

    dichter.name      = "Rilke";
    dichter.vorname   = "Rainer";
    dichter.geburtsjahr = 1854;
    dichter.ausgabe();

    return 0;
}
```

Idee dahinter:

Bündelung von Daten und darauf operierenden Funktionen zu einer Einheit.

Wird so aber nur selten verwendet.

Stattdessen Einsatz des verallgemeinerten Konzepts der **Klasse** (siehe später)

Zeiger

Zeiger in C++ sind (wie in C) Variablen, die Speicheradressen enthalten.

Das Zeigerkonzept ist sehr leistungsfähig, aber auch fehleranfällig

Vorteile:

- Speicherersparnis: Ein Zeiger benötigt üblicherweise 4 Byte und kann damit Objekte, die viel größer sind, verwalten.
- Ermöglichen "call-by-reference" Parameterübergabe, d.h. Funktionen können Objekte außerhalb ihres Gültigkeitsbereiches manipulieren.
- Ermöglichen Implementierung dynamischer Datenstrukturen, z.B. Bäume
- Ermöglichen generische Implementierungen, z.B. durch void-Pointer

Nachteil: Durch Fehler oder Unachtsamkeiten sind Programmabstürze, Speicherlecks, usw. möglich.

Zeiger

Definition, Initialisierung und Verwendung eines Zeigers: wie in C

Beispiel:

```
int i = 10;
```

```
int *ptr1 = &i;
```

```
int *ptr2 = ptr1;
```

```
int feld[3] = {20, 21, 22};
```

```
ptr2 = feld;
```

```
printf("*ptr1 = %d, *ptr2 = %d\n", *ptr1, *ptr2);
```

Ausgabe?

*ptr1 = 10, *ptr2 = 20

Zeiger

Definition, Initialisierung und Verwendung eines Zeigers: wie in C

Beispiel:

```
int i = 10;
```

```
int *ptr1 = &i;
```

```
int *ptr2 = ptr1;
```

```
int feld[3] = {20, 21, 22};
```

```
ptr2 = feld;
```

```
printf("*ptr1 = %d, *ptr2 = %d\n", *ptr1, *ptr2);
```

OOP-Terminologie:

Zeiger **ptr1** zeigt auf ein
Objekt der **Klasse** Integer.

Details: später

Zeiger

Neue Objekte können wie in C erzeugt werden:

Neu in C++: Verwendung des `new`-Operators (mehr zu `new`: später)

```
double *ptr;  
  
ptr = new double;  
  
*ptr = 2.0;  
  
printf("%.2f\n", *ptr);
```

Mit dem `new`-Operator wird ein neues Objekt der Klasse `double` erzeugt. Dessen Adresse wird `ptr` zugewiesen, so dass über Dereferenzierung (`*`) auf dessen Inhalt zugegriffen werden kann.

C++ Schnellkurs:

C++: Neue Sprachmittel

C++: Neue Sprachmittel

- Referenzen
- Funktionen: Vorgabeargumente, Überladung, Templates, inline
- Namensräume
- Dynamische Speicherverwaltung
- Ein- und Ausgabe
- Strings
- Klassen (später)

Referenzen

Referenzen

- verweisen wie Zeiger auf Objekte und werden ähnlich verwendet
- sind eine Art **Pseudonym** für Objekte, d.h. man kann sie verwenden wie das Objekt selber, benötigt dafür aber nur Speicherplatz für eine Adresse
- werden auch als **Aliase** bezeichnet (alias: englisch für Pseudonym)

Unterschiede zu Zeigern:

- Referenz muß initialisiert werden
- Referenz kann nur einmal mit Objekt initialisiert werden. Spätere "Umlenkung" des Verweises auf ein anderes Objekt ist nicht möglich.
- Referenz kann wie ganz normale Variablen verwendet werden, d.h. die Zeiger-typische Dereferenzierungssyntax entfällt.
- Referenz kann nicht NULL sein, da sie immer auf ein Objekt verweist
- Referenzen können nicht in Arrays gespeichert werden

Referenzen

Technisch ausgedrückt:

Eine Referenz entspricht einem **const**-Zeiger, der bei jedem Zugriff automatisch dereferenziert wird.

Zeiger-typische Dereferenzierungssyntax entfällt

Referenz kann nur einmal mit Objekt initialisiert werden.

Referenzen

Vorteile von Referenzen gegenüber Zeigern:

- Referenzen können wie normale Variablen angesprochen werden - eine umständliche **Dereferenzierung ist nicht notwendig**
- Der Einsatz von Referenzen ist **sicherer**, da sie immer auf ein genau definiertes Objekt verweisen.

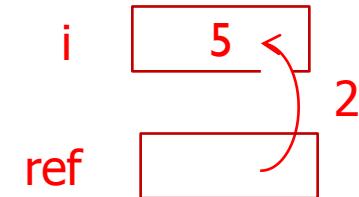
Zeiger dagegen können beliebig verändert werden, so dass an manchen Punkten im Programm möglicherweise unklar ist, wohin sie zeigen.

Referenzen

Definition einer Referenz: Verwendung des Symbols &

Typ & Referenz = Bezeichnung

```
int i = 5, k = 9;  
int& ref = i;      // Deklaration und Initialisierung einer  
                   // Referenz auf i mit dem Namen 'ref'  
  
ref = 2;          // i hat den Wert 2  
ref++;           // i hat den Wert 3  
ref = k;          // i hat den Wert 9
```



Beachte:

Initialisierung muss direkt bei der Deklaration durchgeführt werden.

→ späteres Initialisieren / "Umlenken" auf anderes Objekt nicht möglich.

Referenzen

Problem hier?

```
int i;  
  
int &ref;  
  
ref = i;
```

Compiler-Fehlermeldung:

'ref' declared as reference but not initialized

Initialisierung wurde nicht direkt bei der Deklaration durchgeführt.

Referenzen

Weiteres Beispiel:

```
int main(int argc, char* argv[])
{
    int x = 5;
    int& y = x;
    y = 9;           // x = 9

    int a[100];
    int& b = a[55]; // b ist Referenz auf a[55] (leichtere Ansprechbarkeit)
    b = y;          // a[55] = 9

    y = 12;
    cout << "x = " << x << endl << "y = " << y << endl;
    cout << "b = " << b << endl << "a[55] = " << a[55] << endl;

    return 0;
}
```

Ausgabe?

```
x = 12
y = 12
b = 9
a[55] = 9
```

Achtung: Hier wird die Referenz **b** nicht umgelenkt,
sondern es findet eine Wertzuweisung statt!

Bemerkung: Interessant ist die Verwendung von Referenzen vor allem bei
größeren Objekten.

Referenzen als Aufrufparameter

- C/C++: Call by reference über Zeiger möglich
- C++: Call by reference auch über Referenzen möglich

```
// call by value
void inc1( int v )
{
    v++; // inkrementiert v, nicht x
}
```

```
int main( int argc, char* argv[] )
{
    int x = 5;

    inc1(x); // x = 5

    return 0;
}
```

Referenzen als Aufrufparameter

- C/C++: Call by reference über Zeiger möglich
- C++: Call by reference auch über Referenzen möglich

```
// call by value
void inc1( int v )
{
    v++; // inkrementiert v, nicht x
}

// call by reference (Zeiger)
void inc2( int *p )
{
    (*p)++; // inkrementiert x
}
```

```
int main( int argc, char* argv[] )
{
    int x = 5;

    inc1(x);      // x = 5
    inc2(&x);    // x = 6

    return 0;
}
```

Referenzen als Aufrufparameter

- C/C++: Call by reference über Zeiger möglich
- C++: Call by reference auch über Referenzen möglich

// call by value

```
void inc1( int v )
{
    v++; // inkrementiert v, nicht x
}
```

// call by reference (Zeiger)

```
void inc2( int *p )
{
    (*p)++; // inkrementiert x
}
```

// call by reference (Referenz)

```
void inc3( int& v )
{
    v++; // inkrementiert x
}
```

```
int main( int argc, char* argv[] )
{
    int x = 5;

    inc1(x);      // x = 5
    inc2(&x);    // x = 6
    inc3(x);      // x = 7

    return 0;
}
```

Aber: bei Funktionsaufruf nicht ersichtlich, daß Variable verändert werden kann!

Referenzen zur Effizienzsteigerung

Übergabe großer Datenstrukturen (Objekte) an Funktionen

- Kopieren großer Speicherbereiche wird vermieden
- Trotzdem „übersichtliche“ Programmierung ohne Zeigersyntax
- Rein lesender Zugriff sollte über „const“ erfolgen

```
enum MODEL { ENTERPRISE, VOYAGER, ... };
```

```
struct SpaceShip
{
    ... // viele Datenelemente
    MODEL model;
    ... // viele Datenelemente
};
```

```
MODEL getModel( const SpaceShip& s )
{
    return s.model;
}
```

```
#include "spaceShip.h"

int main( int argc, char* argv[] )
{
    SpaceShip s;
    ...
    if ( getModel(s) == VOYAGER )
    ...
    return 0;
}
```

Funktionen: Erweiterungen in C++

Vorgabeargumente

Beim Aufruf einer Funktion wird jedem Parameter ein Argument übergeben

Ausnahme: Die Funktion definiert Parameter mit Vorgabeargumenten

Beispiel:

```
string repeat(int anzahl, char c = '-')
{
    string s = "";
    for (int i = 0; i < anzahl; i++)
    {
        s += c;
    }
    return s;
}
```

Bemerkung:

Anhängen eines neuen Zeichens an
s durch den += Operator.

Funktionen: Erweiterungen in C++

Vorgabeargumente

Beim Aufruf einer Funktion wird jedem Parameter ein Argument übergeben

Ausnahme: Die Funktion definiert Parameter mit Vorgabeargumenten

Beispiel:

```
string repeat(int anzahl, char c = '-')
{
    string s = "";
    for (int i = 0; i < anzahl; i++)
    {
        s += c;
    }
    return s;
}
```

Mögliche Aufrufe :

```
string line = repeat(10);
string line = repeat(10, '*');
```

Funktionen: Erweiterungen in C++

Vorgabeargumente

- Wenn ein Parameter mit Vorgabewert angegeben wird, müssen alle folgenden Parameter auch einen Vorgabewert besitzen
- Wird bei einem Funktionsaufruf ein Parameter weggelassen, müssen auch die folgenden weggelassen werden
- Vorbelegung entweder in Deklaration *oder* in Implementierung angeben (empfohlen: in der Deklaration)

```
float f( int x, float y = 7.5, char s = 'a' ) ; // Deklaration  
  
float f( int x, float y, char s ) {...}           // Implementierung  
  
float g( int x, float y = 0, char s) ; // Fehler!
```

Funktionen: Erweiterungen in C++

Überladung von Funktionen

C++ erlaubt Definition mehrerer Funktionen gleichen Namens,
falls die Funktionen sich in Anzahl oder Typ der Parameter unterscheiden.

Motivation:

- Es gibt viele Situationen, in denen dieselbe Aufgabe für unterschiedliche Arten von Parametern gelöst werden muss.
- Lösung durch Definition unterschiedlicher Funktionen
- Ohne Überladung müssten diese Funktionen unterschiedliche Namen tragen.

Beispiel:

```
string repeat(int anzahl, char c = '-') // Variante 1
{
    string s = "";

    for (int i = 0; i < anzahl; i++)
    {
        s += c;
    }
    return s;
}

string repeat(int anzahl, string str) // Variante 2
{
    string s = "";

    for (int i = 0; i < anzahl; i++)
    {
        s += str;
    }
    return s;
}
```

Zugehörige main()-Funktion:
→ siehe nächste Seite

Ausgabe:

Version 1 : hhhhhhhhhh

Version 2 : -----

Version 3 : hallohallohallohallohallohallohallohallohallohallohallo

Drücken Sie eine beliebige Taste . . .

```
s = repeat(10, 'h');

cout << "Version 1 : " << s << endl;

s = repeat(10); // hier wird Default-Parameter verwendet

cout << "Version 2 : " << s << endl;

s = repeat(10, "hallo"); // Variante 2 der Funktion

cout << "Version 3 : " << s << endl;

system("PAUSE");

return EXIT_SUCCESS;
```

Funktionen: Erweiterungen in C++

Überladung von Funktionen

- Signatur (Funktionsname + Parameterliste) muß verschieden sein
- Keine Überladung, wenn bei gleicher Signatur nur der Rückgabetyp unterschiedlich ist
- Typen T und T& werden nicht unterschieden

Überladung:

- Bei unterschiedlichen *Parametertypen*, aber gleicher Funktionalität (nicht unbedingt gleicher Funktionsrumpf)

Vorgabeargumente:

- Bei unterschiedlicher *Parameteranzahl*, aber gleicher Funktionalität (gleicher Funktionsrumpf)

Funktionen: Erweiterungen in C++

Überladung von Funktionen: Beispiel

```
char maximum( char a, char b )           // maximum für 'char'  
{   return ( a >= b ) ? a : b; }  
  
int maximum( int a, int b )             // maximum für 'int'  
{   return ( a >= b ) ? a : b; }  
  
double maximum( double a, double b )    // maximum für 'double'  
{   return ( a >= b ) ? a : b; }  
  
maximum( '1', '3' );  
maximum( 1, 2 );  
maximum( 1.5, 2.1 );  
// maximum ( 1.1, 2 );    // Fehler  
maximum ( 1.1, static_cast<double>(2) );
```

Funktionen: Erweiterungen in C++

Funktionstemplates

Motivation: Dieselbe Aufgabe für **verschiedene Datentypen** erledigen.

Beispiele: Maximum berechnen, Variablen vertauschen, sortieren,...

→ Int-Variablen, Double-Variablen, String-Variablen, ...

- Mit Templates (engl. für "Schablonen") können Funktionen für einen beliebigen, später (bei der Benutzung) festzulegenden Datentyp geschrieben werden
- **Templates: Typunabhängige Vorlagen für Funktionen**
- Für den noch unbestimmten Datentyp wird ein Platzhalter eingeführt.
- Compiler erzeugt daraus durch Typbindung echte Funktionen

Funktionen: Erweiterungen in C++

Funktionstemplates: Allgemeine Form einer Template-Funktion:

```
template < typename TypBezeichner >
Rückgabetyp funktionsname( Parameterliste) { ... } ;
```

TypBezeichner: beliebiger Name, der in Funktionsdef. als Datentyp verwendet wird.

Häufig: T (für Template)

Beispiel:

```
template < typename T > // T: Platzhalter
```

```
T maximum( T a, T b )
```

```
{    return ( a >= b ) ? a : b; }
```

```
maximum( '1', '1' ); // T → char:     char maximum( char a, char b )
```

```
maximum( 1.5, 2.1 ); // T → double: double maximum(double a, double b)
```

```
maximum(2, 3.3); // Fehler; stattdessen: maximum<double>(2, 3.3);
```

Compiler definiert
entsprechende Funktion



Funktionen: Erweiterungen in C++

Schlüsselwort „inline“

- Schlüsselwort `inline` vor Funktionskopf (Empfehlung an den Compiler): anstelle eines Funktionsaufrufes fügt Compiler den gesamten Funktionscode an der Aufrufstelle ein (wenn er der Empfehlung inline folgt!)
- Definition der inline-Funktion muß i.a. im Header stehen
- Vorteile:
 - Overhead für Parameterübergabe und Unterprogrammaufruf fallen weg
 - Kurze, strukturierende Funktionsschreibweise ohne Geschwindigkeitsverlust

```
struct Bsp           bsp.h
{
    int a;
    ...
};

inline void setA( Bsp* p, int v)
{   p->a = v; }
```

```
#include "bsp.h"

...
int main(...)
{
    Bsp s;
    ...
    setA( &s, 7 );
    ...
}
```

main.cpp

Namensräume

Ziel: Lösung/Vermeidung von Namenskonflikten

Beispielproblem

- Programmierer hat in seinem Programm Klasse mit Namen **x** definiert
- Möchte nun eine Bibliotheksfunktion verwenden und bindet mit include die entsprechende Header-Datei ein.
- Beim Kompilieren zeigt sich, dass es Namenskonflikt gibt, da in der eingebundenen Bibliothek bereits eine Klasse mit Namen **x** definiert ist.

Vorschläge zur Lösung dieses Namenskonflikts?

Namensräume

Vorschläge zur Lösung des Namenskonflikts?

- Selber auf den Namen X verzichten
 - Auf die Bibliothek verzichten
 - Header-Datei kopieren und Element X daraus entfernen (wenn es nicht verwendet werden soll).
- alles unbefriedigend

Ähnliche Probleme können auftreten, wenn mehrere Programmierer ihre Quelltexte zu einem gemeinsamen Programm zusammenführen wollen.

Namensräume

Zur Lösung solcher Probleme wurden Namensräume eingeführt

Definition eines Namensraumes:

```
namespace demospace
{
    struct X
    {
        // ...
    }
}
```



The word **demospace** is circled in red, and a dashed arrow points from this circle to the text "Name des Namensraumes" located to the right of the code.

Namensräume

Zur Lösung solcher Probleme wurden Namensräume eingeführt

Definition eines Namensraumes:

```
namespace demospace  
{  
    struct X  
    {  
        // ...  
    }  
}
```

Alle Elemente aus diesem Namensraum können nicht mehr alleine nur über ihren Namen angesprochen werden.

Statt dessen Verwendung des so genannten

vollqualifizierten Namens

Namensräume

Zur Lösung solcher Probleme wurden Namensräume eingeführt

Definition eines Namensraumes:

```
namespace demospace  
{  
    struct X  
    {  
        // ...  
    }  
}
```

Alle Elemente aus diesem Namensraum können nicht mehr alleine nur über ihren Namen angesprochen werden.

Statt dessen Verwendung des so genannten

vollqualifizierten Namens

Scope-Operator

Beispiel:

demospace::X

Namensräume

Effekt: Compiler und Linker können problemlos zwischen Elementen aus verschiedenen Namensräumen unterscheiden.

`meinNamensraum::X`

`andererNamensraum::X`

Namensräume: Beispiel (1)

```
void f();
```

mike.h

joe.h

```
void f();
```

```
#include "mike.h"

void f()
{
...
}
```

mike.cpp

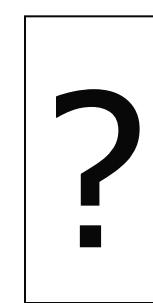
joe.cpp

```
#include "joe.h"

void f()
{
...
}
```

```
#include "mike.h"
#include "joe.h"

int main(...)
{
    f(); // mike's f()
    f(); // joe's f()
    return 0;
}
```



C++: Neue Sprachmittel

Namensräume: Beispiel (1)

```
namespace mike
{
    void f();
}
```

mike.h

joe.h

```
namespace joe
{
    void f();
}
```

```
#include "mike.h"

namespace mike
{
    void f()
    {
        ...
    }
}
```

mike.cpp

joe.cpp

```
#include "mike.h"
#include "joe.h"

int main(...)
{
    mike::f();
    joe::f();
    return 0;
}
```

```
#include "joe.h"

namespace joe
{
    void f()
    {
        ...
    }
}
```

Scope-Operator

C++: Neue Sprachmittel

Namensräume: Beispiel (1) - alternativ

```
namespace mike
{
    void f();
}
```

mike.h

joe.h

```
namespace joe
{
    void f();
}
```

```
#include "mike.h"

void mike::f()
{
    ...
}
```

mike.cpp

joe.cpp

```
#include "joe.h"

void joe::f()
{
    ...
}
```

```
#include "mike.h"
#include "joe.h"

int main(...)
{
    mike::f();
    joe::f();
    return 0;
}
```

Scope-Operator

Namensräume

Problem:

Verwendung der vollqualifizierten Namen ist lästig,
falls es gar keine Konflikte gibt.

Beispiel:

```
#include <iostream>
#include <string>

int main(void)
{
    std::string gruss;
    gruss = "Hallo Welt";
    std::cout << gruss << std::endl;

    return 0;
}
```

Namensräume

Lösung: Verwendung der `using`-Deklaration, welche die Verwendung eines Namensraumes bekanntgibt, so dass alle enthaltenen Elemente über ihren einfachen Namen angesprochen werden können.

```
#include <iostream>
#include <string>

using namespace std;           // Namensraum bekannt geben

int main(void)
{
    string gruss;
    gruss = "Hallo Welt";
    cout << gruss << endl;

    return 0;
}
```

Namensräume: Beispiel (2)

```
#include "joe.h"

void f() {...} // globales f()

int main(...)
{
    f();           // globales f()
    {
        using joe::f;
        f();       // joe's f()
        ::f();    // globales f()
    }
    f();           // globales f()

    return 0;
}
```

joe.h

```
namespace joe
{
    void f();
}
```

joe.cpp

```
#include "joe.h"

namespace joe
{
    void f()
    {
        ...
    }
}
```

Dynamische Speicherverwaltung

Was bedeutet "dynamisch" hier?

Programmierer kann Speicherplatz je nach Bedarf anfordern und freigeben

- Dynamischer Speicher wird auf dem „Heap“ angelegt

Zur dynamischen Speicherverwaltung stehen zur Verfügung:

A. C++-Operatoren

- new
- delete

B. Aus C übernommene Funktionen

- malloc()
- free()

Dynamische Speicherverwaltung

1. Speicherplatz anfordern

- Falls der angeforderte Speicher bereitgestellt werden kann, liefern sowohl `malloc()` als auch `new` die Anfangsadresse des Bereiches zurück.
- Sonst: `new` löst sogenannte Exception aus (später), `malloc` liefert NULL

Beispiel: Speicheranforderung für 10 double-Elemente

```
double *ptr;  
  
ptr = (double *) malloc (10*sizeof(double)) ;  
  
ptr = new double[10]; // Alternative mit new-Operator
```

Dynamische Speicherverwaltung

2. Speicherplatz freigeben

- Mit `malloc()` allokierte Speicherplatz muss mit `free()` wieder freigegeben werden. Sonst bleibt Speicherpl. bis Programm-Ende reserviert.
- Entsprechendes gilt für `new` und `delete`
- „Jedes `malloc()` hat ein `free()`, jedes `new` hat ein `delete!`“
- Wird Speicher nicht wieder freigegeben, spricht man von "Speicherlecks"

Beispiel:

```
double *ptr;  
  
ptr = (double *) malloc (sizeof(double));  
  
free(ptr);  
  
ptr = new double; // Alternative mit new-Operator  
  
delete ptr;
```

Beispiel mit malloc() und free()

```
int main(void)
{
    double *ptr;
    int elemente;

    cout << "Wie viele Elemente soll Array enthalten?" << endl;
    cin >> elemente;

    ptr = (double *) malloc (elemente*sizeof(double));

    for (int i=0; i<elemente; i++)
    {
        *(ptr+i) = i+1;
    }

    for (int i=0; i<elemente; i++)
    {
        cout << *(ptr+i) << endl;
    }

    free(ptr);

    return 0;
}
```

Beispiel mit new und delete

```
int main(void)
{
    double *ptr;
    int elemente;

    cout << "Wie viele Elemente soll Array enthalten?" << endl;
    cin >> elemente;

    ptr = new double[elemente];      // Anlegen eines Feldes

    for (int i=0; i<elemente; i++)
    {
        *(ptr+i) = i+1;
    }

    for (int i=0; i<elemente; i++)
    {
        cout << *(ptr+i) << endl;
    }

    delete[] ptr;                  // da Feld: delete[] statt delete

    return 0;
}
```

Dynamische Speicherverwaltung

Tipps und Hinweise:

- **new** : für alle Datentypen (Standardtypen: z.B. bei Feldern variabler Länge)
- Zur Freigabe wird Zeiger auf den Anfang des Speicherbereiches benötigt
 - Speicher kann nicht mehr freigegeben werden, wenn dieser Zeiger verloren geht („Speicherleck“)
- Dereferenzierung eines Zeigers, der auf keine korrekte Speicheradresse zeigt, führt i.d.R. zu schwerwiegenden Fehlern
 - Zeigern, die auf keine korrekte Adresse zeigen, sollte NULL zugewiesen werden.
 - Vor einer Dereferenzierung prüfen, ob Zeiger auf NULL steht.
- **delete** kann gefahrlos auf NULL-Zeiger angewendet werden

Ein- und Ausgabe: Streamkonzept

- Datenstrom (stream) bezieht sich auf Ein-/ Ausgabe von allen Dateien, d.h. physikalische Datei auf Datenträger, Konsole, Geräte...
- In C: `stdout`, `stdin`, `stderr`; `<stdio.h>`
 - Daten werden mit `printf`, `scanf` etc. bearbeitet (typunsicher)
- In C++: `cin`, `cout`, `cerr`, `clog`; `<iostream>` (IO-Bibliothek)
 - Streams als Objekte zur Eingabe und Ausgabe
 - `cin` ist mit Standardeingabe (Tastatur) verbunden
 - `cout` ist mit Standardausgabe (Konsole) verbunden
 - `cerr` für Fehlermeldungen (Default: Standardausgabe)
 - `clog` für Log-(Kontroll-)meldungen (Default: Standardausgabe)

Ein- und Ausgabe: Umleitungsoperator <<

- Spitzen zeigen in die Richtung, in die der Strom umgeleitet wird
- Auszugebende Elemente werden mit dem Umleitungsoperator getrennt
- ... darf nicht mit dem Shift-Operator verwechselt werden
- Beispiele:

```
cout << "3 * 4 = " << result << endl; (Verkettung von <<)
```

```
int z;  
cout << "Please enter a decimal number: " << endl;  
cin >> z;
```

`cout << result;` gleichbedeutend mit

`operator<< (cout, result);`

`ostream& operator<< (ostream os, <dataType>)`



Rückgabe von Referenz auf Streamobjekt
→ Verkettung möglich



output stream
(z.B. cout)



Datentyp (Operator wird für
jeweiligen Datentyp überladen)
254

Ein- und Ausgabe: Beispiel

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

int main (int argc, char* argv[])
{
    int zahl01, zahl02;
    char zeichen;

    cout << "Bitte geben Sie zwei Ganzzahlen ein: ";
    cin  >> zahl01 >> zahl02;
    cout << "Bitte geben Sie ein Zeichen ein: ";
    cin  >> zeichen;

    return 0;
}
```

- Datenformat wird automatisch berücksichtigt und konvertiert!
- Ein-/Ausgabe von Variablen (Objekten) selbstdefinierter Datentypen durch geeignete Überladung des Umleitungsoperators << (später)

Ein- und Ausgabe: Manipulatoren

- Bisher: *Was* wird ausgegeben? Jetzt: *Wie* wird ausgegeben?

C	C++	Ausgabe
<code>printf("Hello");</code>	<code>cout << "Hello";</code>	Hello
<code>printf("%d", 100);</code>	<code>cout << dec << 100;</code>	100
<code>printf("%X", 100);</code>	<code>cout << hex << 100;</code>	64
<code>printf("%o", 100);</code>	<code>cout << oct << 100;</code>	144
<code>printf("%e", 100.11);</code>	<code>cout << scientific << 100.11;</code>	1.001100e+002
<code>printf("%f", 100.11);</code>	<code>cout << fixed << 100.11;</code>	100.11
<code>printf("%+f", 100.11);</code>	<code>cout << fixed << showpos << 100.11;</code>	+100.11
<code>printf("%.3e", 100.11);</code>	<code>cout << setprecision(3) << 100.11; //(*)</code>	100.110
<code>printf("%8f", 100.11);</code>	<code>cout << setw(8) << 100.11; //(*)</code>	100.11
<code>printf("%+9.5f", 100.11);</code>	<code>cout << showpos << setw(9) << showpoint << setprecision(5) << fixed << 100.11; //(*)</code>	+100.11000

- Mit (*) gekennzeichnete Manipulatoren erfordern Header-Datei <iomanip>
- Siehe z.B. <http://www.cs.fsu.edu/~myers/c++/notes/formatting.html>

Strings: Einführung

- In C: kein eigener Datentyp, sondern `char`-Arrays, die mit '\0' enden
`char st[] = { 'A', 'B', 'C', '\0' }` „C-String“ (unhandlich)
`char st[] = "ABC";`
- C++: Typ `string` (Klasse)
 - Funktionen, Methoden, Operatoren für die Arbeit mit Zeichenketten
 - Deklarationen in Headerdatei `<string>`, Namensraum `std`

```
#include <string>
```

```
std::string st = "ABC";
```

- Mit Hilfe des Datentyps wird automatisch genügend Speicher bereitgestellt

```
string s0; string s1("Beispiel"); // Erzeugung
```

```
s0 = "Beispiel"; s1 = s0;           // Zuweisung
```

Zeichenketten in C versus Strings in C++

```
char st1[5], st2[15], st3[30]; // Größen fix, nicht veränderbar
                                // Platz für '\0' einkalkulieren!
strcpy( st1, "Karl" );          // da st1 = "Karl" nicht geht!
strcpy( st2, " studiert hier"); // Grenzen selber kontrollieren
strcat( st1, st2 );            // st1 = st1 + st2 geht nicht!
```

```
#include <iostream>
#include <string>

int main()
{
    string s1 = "Karl", s2;           // Erzeugung
    s2 = " studiert hier";          // Zuweisung
    s1 = s1 + s2;                  // Verkettung
    std::cout << s1 << std::endl;   // Ausgabe
    return 0;
}
```

Klassen

Klassen sind eine Verallgemeinerung des Struktur-Datentyps aus C

Klassen fassen

- Variablen und
- darauf operierende Funktionen

zu einer Einheit zusammen.

Beispiel:

- Personendaten und
- Funktionen, die die Personendaten verarbeiten (ausgeben, verändern,...)

Einzelheiten im nächsten Kapitel!

Zusammenfassung (1)

- Entstehung von C++ als Erweiterung von C
 - C++ ist keine reine objektorientierte Sprache, sondern Hybridsprache
 - Entwurfsziele von C++:
 - Effizienz (Code und Laufzeit)
 - Größere Typsicherheit
 - Vernünftige Strukturierung großer Programme
 - Leichte Änderbarkeit, Wartbarkeit, Zuverlässigkeit
 - Kompatibilität zu C
 - Objektorientiert
- Keine großen Unterschiede zwischen C und C++ bzgl. der Grundkonzepte:
 - Variablen / Datentypen (neuer Datentyp bool, neue Typumwandlung)
 - Operatoren, Kontrollstrukturen, Funktionen, Zeiger
 - Arrays / Aufzählungen / Strukturen (Struktur kann Funktion als untergeordnetes Element erhalten)

Zusammenfassung (2)

- C++ ermöglicht folgende **neue Sprachmittel**:
 - Referenzen (Pseudonym / Alias für Objekte, ähnlich wie Zeiger)
 - Funktionen: Vorgabeargumente, Überladung, Templates, inline
 - Namensräume
 - dynamische Speicherverwaltung: `new`, `delete`
 - Ein- / Ausgabe über Streams (z.B. `cin`, `cout`)
 - Strings (neue Stringbibliothek `<string>`)
 - Klassen (siehe später)

Übersicht (1)

Objektorientierte Programmierung (OOP):

- Einführung
- Objekt und Klasse
- Attribute, Methoden
- Geheimnisprinzip, Kapselung
- Vererbung
- Klassifikation
- Beziehungen zwischen Klassen
- Polymorphie

Schnellkurs C++:

- Einführung
- **Bekannte Sprachmittel:**
 - Variablen, Datentypen, Operatoren, Kontrollstrukturen, Arrays, Strukturen
- **Neue Sprachmittel:**
 - Referenzen
 - Funktionen: Vorgabeargumente, Überladung, Templates
 - Namensräume
 - Ein- und Ausgabe
 - Strings
 - Typumwandlung

Objektorientierte Programmierung mit C++:

- Klassen, Vererbung, Polymorphie, Mehrfachvererbung

Übersicht (2)

Objektorientierte Programmierung mit C++:

- **Klassen**
 - Klasse als Datentyp
 - Member: Instanzvariablen / -methoden, Klassenvariablen / -methoden, Konstruktor, Destruktor, this-Zeiger
 - Sichtbarkeit und Kapselung, Zugriffsspezifizierer, friends
 - Designempfehlungen: const, get/set,
 - Klassentemplates
 - Operatorüberladung für Klassenobjekte
 - Objektverwaltung: Erzeugen, Vergleichen, Kopieren, Auflösen, Komposition...
- **Vererbung**
 - Syntax und Einsatz
 - Basisklassen-Unterobjekt
 - Verdecken, Überschreiben, Überladen
 - Zugriffsmodifizierer, Zugriffsrechte

Übersicht (3)

Objektorientierte Programmierung mit C++:

- Polymorphie
 - Frühe und späte Bindung
 - Virtuelle Funktionen, virtueller Destruktor
 - Abstrakte Methoden
 - Abstrakte Klassen
- Mehrfachvererbung
- Fehlerbehandlung (exception handling)

Objektorientierte Programmierung mit C++

Objektorientierte Programmierung mit C++

- Einführung der Konzepte an praktischen C++-Beispielen
- Motivation des Klassenkonzepts
- Die Klasse als Erweiterung des strukturierten Datentyps

Inhalt:

- Klassen
 - Die Klasse als Datentyp
 - Member einer Klasse
 - Sichtbarkeit und Kapselung
 - Designempfehlungen
 - Operatorüberladung für Klassenobjekte
 - Objektverwaltung
- Vererbung
- Polymorphie

Objektorientierte Programmierung mit C++

Ziele von C++ bei der Realisierung großer Projekte:

- Hohe Zuverlässigkeit
- Überschaubarkeit
- Leichte Änderbarkeit
- Gute Wartbarkeit

Objektorientierte Programmierung mit C++

Lösungen in C++:

- Sicherheit durch Einschränkung von (häufig Fehler verursachenden) Zugriffsmöglichkeiten (Datenkapselung, Geheimnisprinzip, „private“)
- Leichteres Verständnis & Änderbarkeit durch Minimierung der Schnittstellen zwischen den Programmteilen (Zugriffsfunktionen, „public“)
- Automatisierung von immer wieder nötigen Vorgängen, deren Vergessen zu schwer zu findenden Laufzeitfehlern führt (Konstruktor, Destruktor)

Wichtigstes Konzept der OOP: Klasse

Objektorientierte Programmierung mit C++:

Klassen

Klassen

Klassen sind eine Verallgemeinerung des Struktur-Datentyps aus C

Klassen fassen

- Variablen und
- darauf operierende Funktionen

zu einer Einheit zusammen.

Beispiel:

- Personendaten und
- Funktionen, die die Personendaten verarbeiten (ausgeben, verändern,...)

Die Klasse als Datentyp

Von der Struktur zur Klasse: Das Problem der Struktur

```
#include "abc.h"
...
ABC abc;

int main(...)
{
    abc.a = 7;
    ...
}
```

main.cpp

```
#include "abc.h"
...
void f1()
{
    abc.b = 3;
    ...
}
```

f1.cpp

```
struct ABC      abc.h
{
    int a;
    int b;
    int c;
};
```

```
#include "abc.h"
...
void fn()
{
    int y = abc.c;
    ...
}
```

fn.cpp

```
#include "abc.h"
...
void f3()
{
    abc.c = 2;
    ...
}
```

f3.cpp

```
#include "abc.h"
...
void f2()
{
    int x = abc.b;
    ...
}
```

f2.cpp

Quelle: Microconsult

Die Klasse als Datentyp

Von der Struktur zur Klasse: Das Problem der Struktur

```
#include "abc.h"
...
ABC abc;

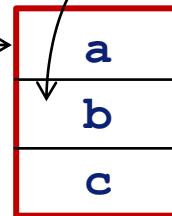
int main(...)
{
    abc.a = 7;
    ...
}
```

main.cpp

```
#include "abc.h"
...
void f1()
{
    abc.b = 3;
    ...
}
```

f1.cpp

```
struct ABC abc.h
{
    int a;
    int b;
    int c;
};
```



Probleme:

- Änderung der Datenelemente?
 - Was passiert bei Abhängigkeiten?
 - Lösung: Zugriffsfunktionen
- } Datenkonsistenzproblem!

Die Klasse als Datentyp

Von der Struktur zur Klasse: Zugriffsfunktionen

```
#include "abc.h"                                abc.cpp
...
// Zugriffsfunktionen

int getA(ABC& p) {return p.a;}
int getB(ABC& p) {return p.b;}
int getC(ABC& p) {return p.c;}

void setA(ABC& p, int v) {p.a=v;}
void setB(ABC& p, int v) {p.b=v;}
void setC(ABC& p, int v) {p.c=v;}
```

```
struct ABC      abc.h
{
    int a;
    int b;
    int c;
};
```

Vorteile:

- falls sich Aufbau eines Datenelements ändert:
nur die Zugriffsfunktion muß geändert werden
- bei Abhängigkeiten zwischen Datenelementen:
Zugriffsfunktionen sorgen für Datenkonsistenz

```
#include "abc.h"
...
ABC abc;

int main(...)
{
    setA(abc,7);
    ...
}
```

main.cpp

Die Klasse als Datentyp

Von der Struktur zur Klasse: Zugriffsfunktionen

Beispiel: Änderung eines Datenelements / Realisierung in Zugriffsfunktion
→ Datenkonsistenz bei jedem Zugriff auf Datenstruktur gewährleistet

```
struct ABC          abc.h
{
    enum { MAX_C=100 };

    int a;
    int b;
    int c;
    // 0 <= c <= MAX_C
};
```

```
int getC( ABC& p ) { return p.c; }

...
bool setC( ABC& p, int v )
{
    if ( (v < 0) || (v > ABC::MAX_C) )
    {
        cout << "invalid value!" << endl;
        return false;
    }
    else
    {
        p.c = v;
        return true;
    }
}
```

Aber: direkter Zugriff auf Datenelement c immer noch möglich!

Von der Struktur zur Klasse: private

Schlüsselwort **private** in Deklaration:

- Strukturelement kann nicht mehr direkt angesprochen werden
- Aufruf muß über Zugriffsfunktion erfolgen

```
struct ABC          abc.h
{
private:
    enum { MAX_C=100 };

    int a;
    int b;
    int c;
    // 0 <= c <= MAX_C
};
```

```
int main( ... )
{
    ABC abc;
    int x;

    abc.c = 117; // Fehler: Zugriff auf
                  // privates Datenelem.
    ...
}
```

Die Klasse als Datentyp

Von der Struktur zur Klasse: private

Schlüsselwort **private** in Deklaration:

- Strukturelement kann nicht mehr direkt angesprochen werden
- Aber: Auch die Funktion **setC** kann nicht auf **c** zugreifen!

```
struct ABC           abc.h
{
private:
    enum { MAX_C=100 };

    int a;
    int b;
    int c;
    // 0 <= c <= MAX_C
};
```

```
bool setC( ABC& p, int v )
{
    p.c = v; // Fehler: Zugriff auf
              // privates Element
}
```

abc.cpp

```
int main( ... )
{
    ABC abc;

    setC( abc, 5 );

}
```

Quelle: Microconsult

Die Klasse als Datentyp

Von der Struktur zur Klasse: private

Lösung: Funktion `setC` wird zu Element der Struktur (Elementfkt., Methode)

- Muß `public` sein, um Zugriff von außen zu ermöglichen

```
struct ABC  
{  
private:  
    enum { MAX_C=100 };  
  
    int a;  
    int b;  
    int c; // 0 <= c <= MAX_C  
  
public:  
    bool setC( ABC& p, int v )  
    {  
        p.c = v;  
    }  
};
```

abc.h

```
int main( ... )  
{  
    ABC abc;  
  
    abc.setC( abc, 5 );  
}
```

Funktion `setC` wird am Objekt `abc` aufgerufen

Die Klasse als Datentyp

Von der Struktur zur Klasse: private

Vereinfachung der Elementfunktion (Details später, „this-Zeiger“)

```
struct ABC                                abc.h
{
private:
    enum { MAX_C=100 } ;

    int a;
    int b;
    int c; // 0 <= c <= MAX_C

public:
    bool setC( int v )
    {
        c = v;
    }
};
```

```
int main( ... )
{
    ABC abc;

    abc.setC( 5 );
}
```

Funktion **setC** wird für
Objekt **abc** aufgerufen

Von der Struktur zur Klasse

Beispiel demonstriert:

Leichtere Änderbarkeit, Datenkonsistenz und größere Überschaubarkeit durch:

- Verbergen der internen Datenelemente einer Struktur
(Geheimnisprinzip, „private“)
- Zugriff auf Datenelemente über zugehörige Elementfunktionen
(Kapselung, Schnittstellen, „public“; Methoden sind Elemente der Struktur!)

Daher: Fasse

- Variablen und
- darauf operierende Funktionen

zu einer Einheit zusammen → **Klasse** (Verallgemeinerung der C-Struktur)

Klassen

Klasse: benutzerdefinierter Datentyp, bestehend aus

- Datenelementen und
- (Element-)Funktionen
 - durch die der Zugriff auf die Datenelemente mittels Zugriffsrechten geeignet gestaltet werden kann

Klasse: Variablen und Funktionen zu einer Einheit zusammengefasst

Analogie: einfache Datentypen (z.B. int):

- Neben der eigentlichen Variablen gibt es eine Gruppe von Operationen, die auf die Variable angewendet werden können.
- Beispiel `int`: Arithmetische Operationen, wie `+`, `-`, `*`, `/`
Vergleichsoperationen, wie `<`, `>`, `>=`, `<=`
Bitmanipulationen, wie `&`, `|`

Die Klasse als Datentyp

Zur Klassendefinition

Schema:

```
class Klassename
{
    Zugriffsspezifizierer_A:
        Elemente

    Elementfunktionen

    Zugriffsspezifizierer_B:
        Elemente

    Elementfunktionen
};
```

Beispiel:

```
class ABC
{
private:
    enum { MAX_C=100 };

    int a;
    int b;
    int c; // 0 <= c <= MAX_C
    ...

public:
    bool setC( int v )
    {
        c = v;
    }
    ...
};
```

Klassendefinition

Allgemeine Syntax der Klassendefinition (ohne Vererbung):

```
class Klassename
{
    Zugriffsspezifizierer_A:           // z.B. (private), public
        Elemente
        Konstruktoren                  // Objektinitialisierung
        Destruktoren                   // Objektauflösung
        Elementfunktionen

    Zugriffsspezifizierer_B:
        Elemente
        ...
};
```

Klassendefinition enthält:

- Deklaration der Datenelemente
- Deklaration (Prototypen) bzw. Definition der zugehörigen Elementfunktionen

Die Klasse als Datentyp

Klassendefinition: Beispiel

```
class kreis
```

```
{
```

private: ← Zugriffsspezifizierer: private (kann weggelassen werden, da Default)

```
    double radius;
```

```
    double centerX, centerY;
```

Klassenname

Elemente

public: ← Zugriffsspezifizierer: public, d.h. ohne Zugriffsbeschränkung

```
kreis ()
```

```
{
```

```
    centerX = 0.0;
```

```
    centerY = 0.0;
```

```
    radius = 1.0;
```

```
}
```

```
~kreis () {};
```

```
double flaeche()
```

```
{
```

```
    return radius * radius * M_PI;
```

```
};
```

Konstruktor (Initialisierung)
Wird einmal ausgeführt, wenn
Objekt der Klasse erzeugt wird
(wird später behandelt)

Destruktor: Aktionen vor
Objektauflösung (später)

Elementfunktion

Die Klasse als Datentyp

Klassendefinition: Beispiel

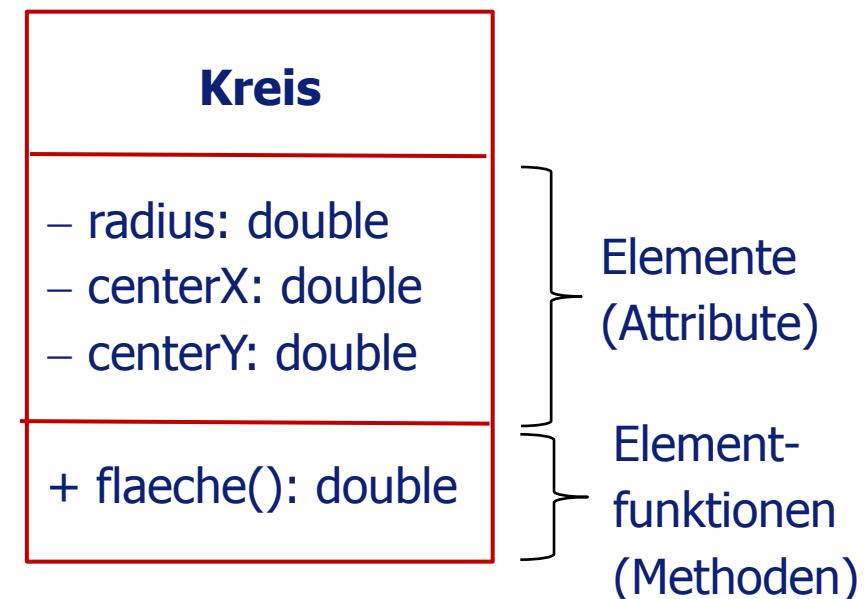
```
class kreis
{
private:
    double radius;
    double centerX, centerY;

public:
    kreis ()
    {
        centerX = 0.0;
        centerY = 0.0;
        radius   = 1.0;
    }

    ~kreis () {};

    double flaeche()
    {
        return radius * radius * M_PI;
    }
};
```

UML:



Klasse: Bestandteile

- Elemente (Member):
 - Klassenelemente (Attribute)
- Elementfunktion (Memberfunktion, Methode):
 - Funktion, die als Element einer Klasse definiert ist
- Konstruktor:
 - Elementfunktion, die beim *Erzeugen* eines Objektes aufgerufen wird
- Destruktor:
 - Elementfunktion, die beim *Auflösen* eines Objektes aufgerufen wird
- Zugriffsspezifizierer:
 - Public: uneingeschränkt zugänglich
 - Protected: zugänglich in eigener und in abgeleiteten Klassen (später)
 - Private: zugänglich nur innerhalb der eigenen Klasse

Unterschied zwischen Klasse und (erweiterter) Struktur

Unterschied zum eben besprochenen, C++-erweiterten Datentyp Struktur?

- i.e. C-Struktur plus Elementfunktionen, public / private – Schlüsselworte
 - Sehr geringe Unterschiede
 - Die Struktur in C++ ist ein Sonderfall der Klasse

Unterschied:

- Elemente einer Struktur sind per Default von überall im Programm her benutzbar bzw. veränderbar („public“)
 - Elemente einer Klasse sind per Default vor Zugriff "von außen" geschützt („private“)
- Daher sollte mit Klassen gearbeitet werden (sicherer)

Klasse und Struktur

Warum gibt es überhaupt beide Konzepte (Klassen und Strukturen) in C++?

Klassen sind zentrales Konzept der objektorientierten Programmierung (OOP)

→ unverzichtbar

Strukturen sind ein Relikt aus der Sprache C, das vor allem aus Kompatibilitätsgründen (Lauffähigkeit alter C-Programme, Gewohnheit der C-Programmierer) beibehalten und konzeptuell an die OOP angeglichen wurde.

Objekt

- Ein **Objekt** ist die Instanziierung einer Klasse
 - Bsp.: Klasse Mensch (abstrakter Begriff)
Objekt Willi (*realer* Mensch)
- Objektdeklaration:
 - Als Variable einer Klasse (Stack):

```
Mensch Willi;  
kreis k;
```

Ein **Objekt** der Klasse Kreis mit dem Namen k wird auf dem Stack angelegt

- Dynamisch mit new (Heap):

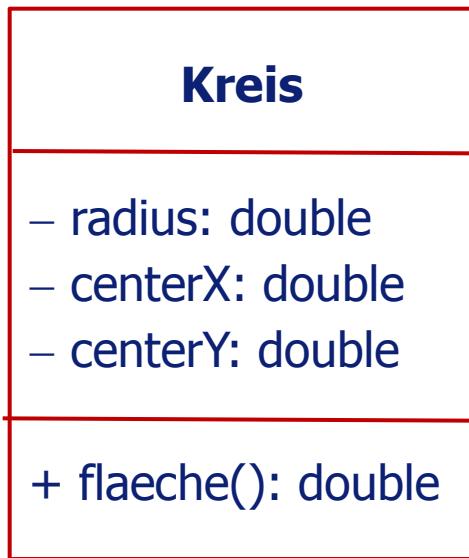
```
Mensch* pWilli = new Mensch;  
kreis* pk = new kreis;
```

(dyn. alloziertes) **Objekt** der Klasse Kreis wird auf dem Heap angelegt; von new gelieferte Adresse im Zeiger pk gespeichert

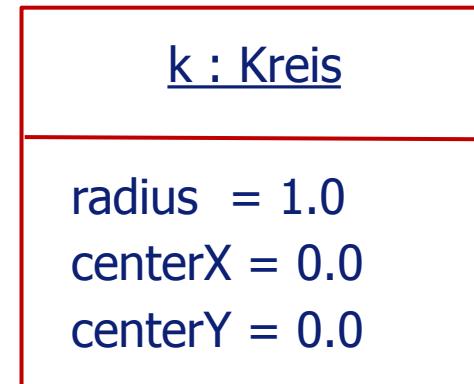
- Speicher für die Elemente des Objektes wird reserviert
- Objektinitialisierung → später („Konstruktor“)

Die Klasse als Datentyp

Objekt



Instanziierung



Klasse: „Schablone“

Objekt: Konkreter Kreis mit
Radius 1 um den Ursprung

Instanziierung: Speicher für die Elemente des Objekts wird bereitgestellt

- radius, centerX, centerY

Die Klasse als Datentyp

Selbstdefinierte Datentypen: Warum Einheit Variable – Funktion?

Einfache Datentypen sind für komplexe Programme unzureichend

→ Konzept der selbst definierten Datentypen (z.B. Arrays, Aufzählungen, C-Strukturen)

Problem:

Diese sind unvollständig, da sie keine Definition typspezifischer Operationen auf den Daten erlauben.

Beispiel: Punkte in 3D können mit Hilfe einer Struktur definiert werden

```
struct point {  
    int x, y, z;  
};
```

Problem: Die Arbeit mit diesen Strukturen ist mühsam, da keine Operationen definiert werden können, die direkt mit dieser Datenstruktur verknüpft sind.

Die Klasse als Datentyp

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

```
    struct point a, b, c;
```

```
    a.x = 1;
```

```
    a.y = 2;
```

```
    a.z = 3;
```

```
    b.x = 2;
```

```
    b.y = 3;
```

```
    b.z = 4;
```

// Verknuepfung zweier Punkte, z.B. Addition?

```
    c.x = a.x + b.x;
```

```
    c.y = a.y + b.y;
```

```
    c.z = a.z + b.z;
```

// Ausgabe von c?

// muss muehsam elementweise durchgefuehrt werden

```
printf("a + b = (%d, %d, %d)\n", c.x, c.y, c.z);
```

```
system("PAUSE");
```

```
return EXIT_SUCCESS;
```

```
}
```

```
struct point {
    int x, y, z;
};
```

Warum Einheit Variable – Funktion?

Das Klassenkonzept erlaubt genau diese gemeinsame Definition von

- Datenstrukturen und
- darauf definierten Operationen

Einwand:

Wir könnten doch auch ohne das Klassenkonzept Funktionen (z.B. `add()` und `ausgabe()`) definieren, die Arbeit für uns erledigen.

Die Klasse als Datentyp

```
struct point add (struct point a, struct point b) {  
    struct point c;  
  
    c.x = a.x + b.x; c.y = a.y + b.y; c.z = a.z + b.z;  
    return c;  
}  
  
void ausgabe (struct point a) {  
    printf("a = (%d, %d, %d)\n", a.x, a.y, a.z);  
    return;  
}  
  
int main(int argc, char *argv[])  
{  
    struct point a, b, c;  
  
    a.x = 1; a.y = 2; a.z = 3;  
    b.x = 2; b.y = 3; b.z = 4;  
  
    c = add(a, b);  
    ausgabe(c);  
  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

```
struct point {  
    int x, y, z;  
};
```

Warum Einheit Variable – Funktion?

Das Klassenkonzept erlaubt genau diese gemeinsame Definition von

- Datenstrukturen und
- darauf definierten Operationen

Frage:

Worin liegt der Vorteil, Daten und Funktionen in einen Datentyp zusammenzuziehen?

Warum Einheit Variable – Funktion?

Schnelle Antwort:

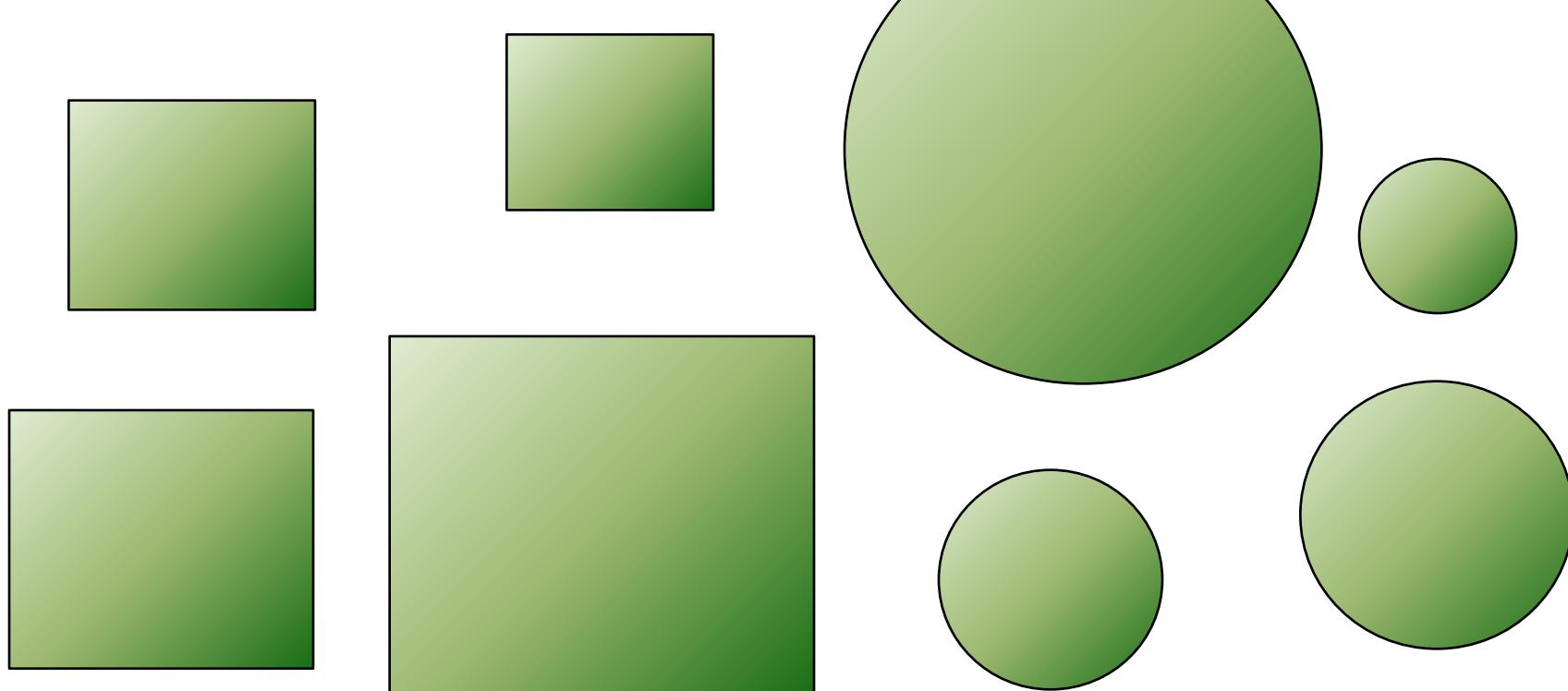
- Die Praxis hat gezeigt, dass die Trennung von Daten und Funktionen fehleranfällig und bei komplexen Programmen schwer handhabbar ist.
- Das Klassenkonzept führt zu besser strukturiertem Quellcode, der vom Compiler leichter überprüft werden kann.
- Das Klassenkonzept ermöglicht viele weitere Verbesserungen, die im Verlauf der Vorlesung besprochen werden.

Die Klasse als Datentyp

Warum Einheit Variable – Funktion?

Etwas ausführlichere Antwort durch ein Beispiel:

Programm, das mit Hilfe einiger Funktionen Umfang und Fläche von Kreisen und Quadraten berechnet.



Die Klasse als Datentyp

```
double umfang_quadrat(double s)
{
    return 4 * s;
}
double flaeche_quadrat(double s)
{
    return s * s;
}
```

```
double umfang_kreis(double r)
{
    return 2 * r * M_PI;
}
double flaeche_kreis(double r)
{
    return r * r * M_PI;
}
```

```
int main()
{
    double radius, seite;

    cout << endl << " Geben Sie den Radius ein: ";
    cin >> radius;

    cout << " Umfang: " << umfang_kreis(radius) << endl;
    cout << " Flaeche: " << flaeche_kreis(radius) << endl;

    cout << endl << " Geben Sie die Seitenlaenge ein: ";
    cin >> seite;

    cout << " Umfang: " << umfang_quadrat(seite) << endl;
    cout << " Flaeche: " << flaeche_quadrat(seite) << endl;
    return 0;
}
```

zulässiges C++-Programm!

Die Klasse als Datentyp

```
double umfang_quadrat(double s)
{
    return 4 * s;
}
double flaeche_quadrat(double s)
{
    return s * s;
}
```

```
double umfang_kreis(double r)
{
    return 2 * r * M_PI;
}
double flaeche_kreis(double r)
{
    return r * r * M_PI;
}
```

```
int main()
{
    double radius, seite;

    cout << endl << " Geben Sie den Radius ein: ";
    cin >> radius;

    cout << " Umfang: " << umfang_kreis(radius);
    cout << " Flaeche: " << flaeche_kreis(radius);

    cout << endl << " Geben Sie die Seitenlaenge ein: ";
    cin >> seite;

    cout << " Umfang: " << umfang_quadrat(seite);
    cout << " Flaeche: " << flaeche_quadrat(seite);

    return 0;
}
```

Programmablauf:

Geben Sie den Radius ein: 3

Umfang: 18.8496

Flaeche: 28.2743

Geben Sie die Seitenlaenge ein: 3

Umfang: 12

Flaeche: 9

Drücken Sie eine beliebige Taste . . .

Die Klasse als Datentyp

Warum Einheit Variable – Funktion?

Beobachtungen:

- Die Daten (radius, seite) sind in der main()-Funktion definiert
- Zwischen den Daten und den Funktionen gibt es ohne das Zutun des Programmierers keine Beziehung.
- Erst durch die Funktionsaufrufe wird (vom Programmierer) eine Beziehung hergestellt

```
int main()
{
    double radius, seite;

    cout << endl << " Geben Sie den Radius ein: ";
    cin >> radius;
    cout << " Umfang: " << umfang_kreis(radius) << endl;
    cout << " Flaeche: " << flaeche_kreis(radius) << endl;

    cout << endl << " Geben Sie die Seitenlaenge ein: ";
    cin >> seite;
    cout << " Umfang: " << umfang_quadrat(seite) << endl;
    cout << " Flaeche: " << flaeche_quadrat(seite) << endl;
    return 0;
}
```

Warum Einheit Variable – Funktion?

Mögliche Fehlerquelle:

- Programmierer vergisst, welche Daten zu welchen Funktionen gehören
- Beispiel: Aufruf von umfang_kreis (seite)
 - Solche Fehler können bei kleinen Programmen und übersichtlichen Funktionsnamen noch vermeidbar sein.
 - Bei großen Programmpaketen, vielen Programmierern und unübersichtlicher Namensgebung ist das eine große Fehlerquelle!

Lösung: **Klassen** → Daten und zugehörige Funktionen verbunden

- Funktionen "wissen", auf welche Daten sie angewendet werden sollen
- Fehlerhafte Funktionsaufrufe mit den falschen Daten sind nicht möglich.

Die Klasse als Datentyp

Warum Einheit Variable – Funktion?

Klassendefinition für die Quadratberechnung:

```
class quadrat {  
public:  
    double seite;  
  
    double umfang()  
    {  
        return 4 * seite;  
    }  
  
    double flaeche()  
    {  
        return seite * seite;  
    }  
};
```

Elemente sollten „private“ sein; hier zur Vereinfachung „public“

Zum Quadrat gehörende Daten (Elemente), hier nur die Seitenlänge

Zum Quadrat gehörende Funktion zur Berechnung des Umfangs. (Elementfunktion)

Zum Quadrat gehörende Funktion zur Berechnung der Fläche.

Die Klasse als Datentyp

Warum Einheit Variable – Funktion?

Klassendefinition für die Kreisberechnung:

```
class kreis
{
public:
    double radius; // Elemente sollten „private“ sein; hier zur Vereinfachung „public“
    double umfang()
    {
        return 2 * radius * M_PI;
    }
    double flaeche()
    {
        return radius * radius * M_PI;
    }
};
```

Zum Kreis gehörende Daten,
hier nur der Radius

Zum Kreis gehörende Funktion
zur Berechnung des Umfangs.

Zum Kreis gehörende
Funktion zur Berechnung
der Fläche.

Die Klasse als Datentyp

```
class quadrat
{
public:
    double seite;

    double umfang()
    {
        return 4 * seite;
    }

    double flaeche()
    {
        return seite * seite;
    }
};
```

```
class kreis
{
public:
    double radius;

    double umfang()
    {
        return 2 * radius * M_PI;
    }

    double flaeche()
    {
        return radius * radius * M_PI;
    }
};
```

```
int main() {
    kreis c;
```

Ein **Objekt** der Klasse Kreis mit dem Namen c wird angelegt.

Die Klasse als Datentyp

```
class quadrat
{
public:
    double seite;

    double umfang()
    {
        return 4 * seite;
    }

    double flaeche()
    {
        return seite * seite;
    }
};
```

```
class kreis
{
public:
    double radius;

    double umfang()
    {
        return 2 * radius * M_PI;
    }

    double flaeche()
    {
        return radius * radius * M_PI;
    }
};
```

```
int main() {
    kreis c;
    quadrat s;
```

Ein **Objekt** der Klasse Quadrat mit dem Namen s wird angelegt.

Die Klasse als Datentyp

```
class quadrat
{
public:
    double seite;

    double umfang()
    {
        return 4 * seite;
    }

    double flaeche()
    {
        return seite * seite;
    }
};
```

```
class kreis
{
public:
    double radius;

    double umfang()
    {
        return 2 * radius * M_PI;
    }

    double flaeche()
    {
        return radius * radius * M_PI;
    }
};
```

```
int main() {
    kreis c;
    quadrat s;

    cout << endl << " Geben Sie den Kreis-Radius ein: ";
    cin >> c.radius;
```

Das Kreis-Objekt bekommt den
eingegebenen Radius zugewiesen.

Die Klasse als Datentyp

```
class quadrat
{
public:
    double seite;

    double umfang()
    {
        return 4 * seite;
    }

    double flaeche()
    {
        return seite * seite;
    }
};
```

```
class kreis
{
public:
    double radius;

    double umfang()
    {
        return 2 * radius * M_PI;
    }

    double flaeche()
    {
        return radius * radius * M_PI;
    }
};
```

```
int main() {
    kreis c;
    quadrat s;

    cout << endl << " Geben Sie den Kreis-Radius ein: ";
    cin >> c.radius;
    cout << " Umfang: " << c.umfang() << endl;
    cout << " Flaeche: " << c.flaeche() << endl;
```



Umfang und Fläche des Kreis-Objektes werden durch Aufruf der Elementfunktionen bestimmt.

Die Klasse als Datentyp

```
class quadrat
{
public:
    double seite;

    double umfang()
    {
        return 4 * seite;
    }

    double flaeche()
    {
        return seite * seite;
    }
};
```

```
class kreis
{
public:
    double radius;

    double umfang()
    {
        return 2 * radius * M_PI;
    }

    double flaeche()
    {
        return radius * radius * M_PI;
    }
};
```

```
int main() {
    kreis c;
    quadrat s;

    cout << endl << " Geben Sie den Kreis-Radius ein: ";
    cin >> c.radius;
    cout << " Umfang: " << c.umfang() << endl;
    cout << " Flaeche: " << c.flaeche() << endl;
    cout << endl << " Geben Sie die Quadrat-Seitenlaenge ein: ";
    cin >> s.seite;
    cout << " Umfang: " << s.umfang() << endl;
    cout << " Flaeche: " << s.flaeche() << endl;
    return 0;
}
```

Die Klasse als Datentyp

```
class quadrat
{
    public:
        double seite;

    double umfang()
    {
        return 4 * seite;
    }

    double flaeche()
    {
        return seite * seite;
    }
};
```

```
class kreis
{
    public:
        double radius;

    double umfang()
    {
        return 2 * radius * M_PI;
    }

    double flaeche()
    {
        return radius * radius * M_PI;
    }
};
```

Konsequenzen:

1. Vermeidung von Namenskonflikten
 - Elementfunktionen mit gleichem Namen (z.B. `umfang()`) sind aufgrund der Kapselung in Klassen möglich.
 - Bei prozeduraler Programmierung hingegen mussten unterschiedliche Funktionsnamen verwendet werden, damit Compiler & Programmierer diese auseinanderhalten konnten.

Die Klasse als Datentyp

```
class quadrat
{
    public:
        double seite;

        double umfang()
        {
            return 4 * seite;
        }

        double flaeche()
        {
            return seite * seite;
        }
};
```

```
class kreis
{
    public:
        double radius;

        double umfang()
        {
            return 2 * radius * M_PI;
        }

        double flaeche()
        {
            return radius * radius * M_PI;
        }
};
```

Konsequenzen:

2. Elementfunktionen greifen auf Elemente zu
 - Elementfunktionen kennen die benötigten Elemente ihrer Klasse und greifen ohne Zutun des Programmierers darauf zu
 - Damit sind fehlerhafte Aufrufe, bei denen falsche Parameter übergeben werden (s. Beispiel oben) nicht möglich.

Die Klasse als Datentyp

```
int main() {  
    kreis c;  
    quadrat s;  
  
    cout << endl << " Geben Sie den Kreis-Radius ein: ";  
    cin >> c.radius;  
    cout << " Umfang: " << c.umfang() << endl;  
    cout << " Flaeche: " << c.flaeche() << endl;  
    cout << endl << " Geben Sie die Quadrat-Seitenlaenge ein: ";  
    cin >> s.seite;  
    cout << " Umfang: " << s.umfang() << endl;  
    cout << " Flaeche: " << s.flaeche() << endl;  
    return 0;  
}
```

Konsequenzen:

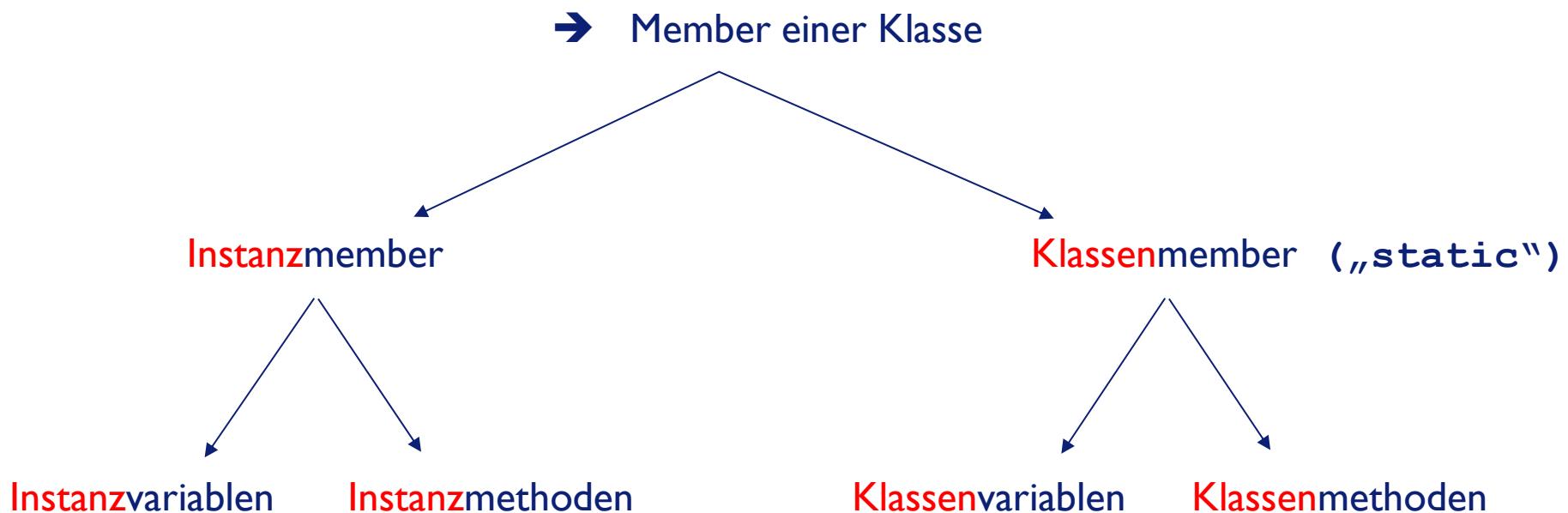
3. Zugriff auf die Objekte ist sicher und bequem
 - Beim Aufruf einer Elementfunktion (z.B. umfang()) kann man absolut sicher sein, dass die richtige Funktion aufgerufen wird.
 - Compiler kann Aufruf einer falschen Funktion leicht erkennen.

Member einer Klasse

Instanzmember versus Klassenmember

Unterscheidung:

- Elemente einer Klasse an Instanz (konkretes Objekt) gebunden: **Instanzmember**
- Elemente *nicht* an Instanz, sondern nur an Klasse gebunden: **Klassenmember**



Motivation: jedes Objekt hat eigene Identität → unterscheidet Eigenschaften

- abhängig von konkretem Objekt (Objekteigenschaft) → **Instanzmember**
- unabhängig von konkretem Objekt (Klasseneigenschaft) → **Klassenmember**

Member einer Klasse

Instanzmember versus Klassenmember

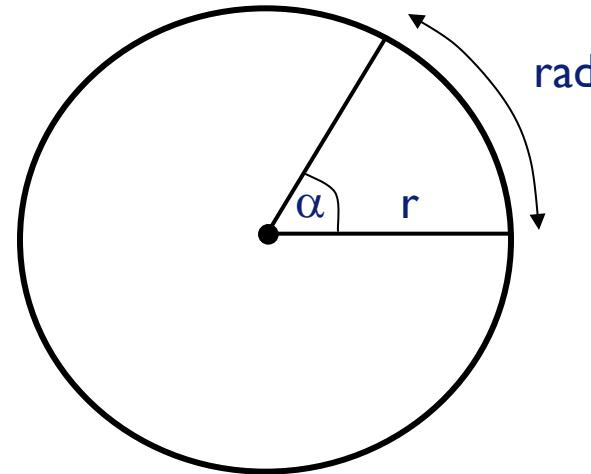
Beispiel: Klasse **Kreis**

Felder:

- Radius
- Pi

Methoden:

- Umfang = $2 \cdot \pi \cdot r$
- Fläche = $\pi \cdot r^2$
- Umrechnung Bogenmass in Winkelmass : $\alpha = \text{rad} \cdot 180 / \pi$ (auf Einheitskreis def.)



Objekt-spezifisches Feld – kann für jede Kreis-Instanz eigenen Wert annehmen

- **abhängig** von der tatsächlichen Realisierung (d.h. vom jeweiligen Kreisobjekt)
- **Instanzvariable**

Member einer Klasse

Instanzmember versus Klassenmember

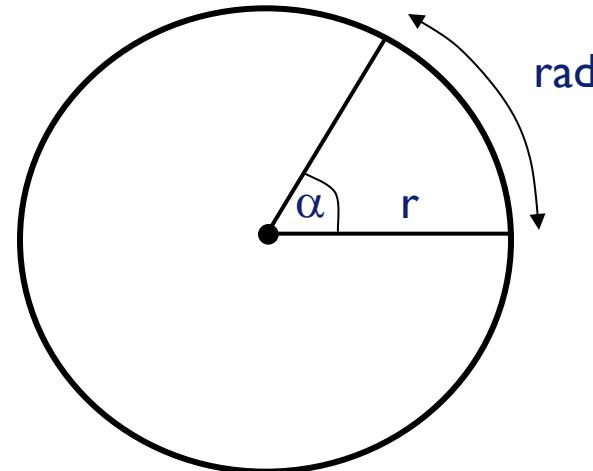
Beispiel: Klasse **Kreis**

Felder:

- Radius
- **Pi**

Methoden:

- Umfang $= 2 \cdot \pi \cdot r$
- Fläche $= \pi \cdot r^2$
- Umrechnung Bogenmass in Winkelmass : $\alpha = \text{rad} \cdot 180 / \pi$ (auf Einheitskreis def.)



Allgemeines oder globales Feld, das für alle Kreis-Instanzen gleichen Wert hat

→ **unabhängig** von einer tatsächlichen Realisierung

→ **Klassenvariable**

Member einer Klasse

Instanzmember versus Klassenmember

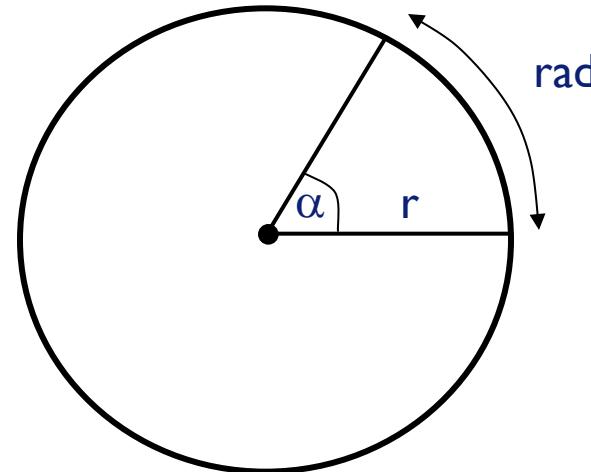
Beispiel: Klasse **Kreis**

Felder:

- Radius
- Pi

Methoden:

- Umfang = $2 \cdot \pi \cdot r$
- Fläche = $\pi \cdot r^2$
- Umrechnung Bogenmass in Winkelmass : $\alpha = \text{rad} \cdot 180 / \pi$ (auf Einheitskreis def.)



Objekt-spezifische Methoden – können für jede Instanz anderes Verhalten zeigen

- **abhängig** von der tatsächlichen Realisierung (d.h. vom jeweiligen Kreisobjekt)
- **Instanzmethoden**

Member einer Klasse

Instanzmember versus Klassenmember

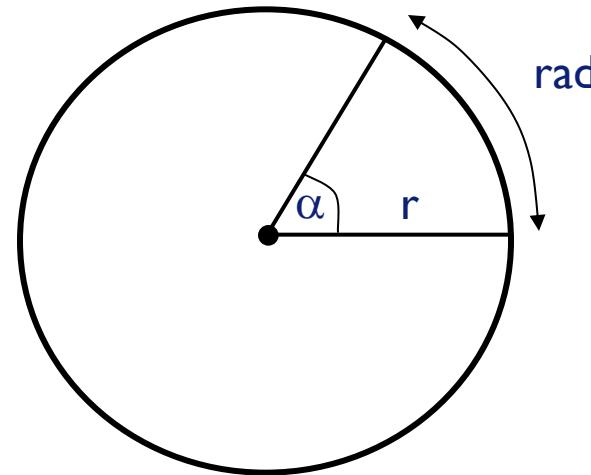
Beispiel: Klasse **Kreis**

Felder:

- Radius
- Pi

Methoden:

- Umfang $= 2 \cdot \pi \cdot r$
- Fläche $= \pi \cdot r^2$
- **Umrechnung Bogenmass in Winkelmass**: $\alpha = \text{rad} \cdot 180 / \pi$ (auf Einheitskreis def.)



Allgemeine oder globale Methode, die für alle Instanzen gleiches Verhalten zeigt
(weil sie auf dem Einheitskreis definiert ist)

- **unabhängig** von einer tatsächlichen Realisierung
- **Klassenmethode**

Instanzvariablen

Deklaration: Innerhalb der Klassendef.: `double radius; // Kreisradius`

- Instanzvariablen gehören zu den **Objekten** der Klasse
 - beschreiben den Zustand eines Objekts
 - sind das Herzstück der objektorientierten Programmierung
- Beim Erzeugen eines neuen Objekts legt der Compiler Speicher an, um diesem Objekt einen eigenen Satz von Instanzvariablen zu geben.

Zugriff:

- innerhalb der Klasse: über den Namen **radius**
- außerhalb der Klasse: über Objekt und Namen

```
kreis k;  
double x = k.radius;
```

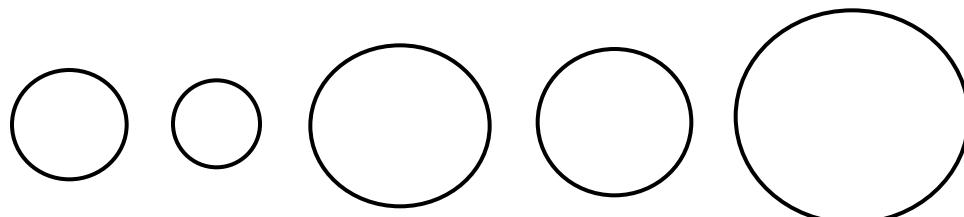
Objekt: Punktoperator `".."`

```
kreis* pk = new kreis;  
double x = pk->radius;
```

Zeiger auf Objekt: Pfeiloperator `"->"`
(Dereferenzierung & Elementzugriff)

Instanzvariablen

Beispiel: Klasse Kreis mit 5 Instanzen



Mit Hilfe der Instanzvariablen **radius** kann jedem Kreis eine eindeutige Identität gegeben werden.

Fragen:

- Wie viele Kopien der Instanzvariablen **radius** gibt es?
- Wie wird (technisch) erreicht, dass verschiedene Kreise mit gleichem Radius ihre eigene Identität haben?

Member einer Klasse

Instanzvariablen: Beispiel

```
class ABC           abc.h
{
private:
    int a;
    int b;
    int c;

public:
    int setA(int v){a = v;}
    int setB(int v){b = v;}
    int setC(int v){c = v;}
};
```

```
int main( ... )           main.cpp
{
    ABC abc1, abc2;
    ABC* pAbc = new ABC;
    ...
    abc1.setA(5); // abc1.a = 5
    abc2.setB(3); // abc2.b = 3;
    pAbc->setC(7); // pAbc->c = 7;
    ...
}
```

abc1

a
b
c

abc2

a
b
c

*pAbc

a
b
c

Instanzvariablen

Initialisierung der Instanzvariablen

- Initialisierung: Zuweisung "sinnvoller" Anfangswerte
→ „Konstruktor“ (siehe später)

Instanzmethoden

- Instanzmethoden sind eines der wichtigsten Merkmale der OOP, weil sie das Verhalten des Objekts (der Instanz) definieren
 - Instanzmethoden operieren auf dem Objekt (der Instanz) (nicht Klasse)
- Instanzmethoden können jedes Element einer Klasse verwenden

Definition: a) innerhalb der Klassendefinition

```
const double PI = 3.14159;  
class kreis  
{  
public:  
    double radius;  
    double flaeche () // berechnet die Kreisflaeche  
    {  
        return PI * radius * radius;  
    }  
}
```

Funktion erhält automatisch die **inline** – Empfehlung!

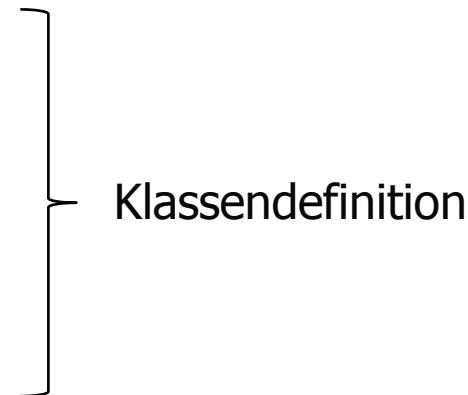
Member einer Klasse

Instanzmethoden

Definition: b) außerhalb der Klassendefinition

```
const double PI = 3.14159;

class kreis
{
public:
    double radius;
    double flaeche (); // Deklaration
};
```



Klassenname + Scope-Operator + Funktionsname

```
double kreis::flaeche () // Definition
{
    return PI * radius * radius;
}
```

Member einer Klasse

Instanzmethoden

Definition: b) außerhalb der Klassendefinition: **inline**

```
const double PI = 3.14159;  
  
class kreis  
{  
  
public:  
  
    double radius;  
  
    inline double flaeche();  
  
};
```

inline nur Empfehlung
an den Compiler!

Schlüsselwort **inline** i.a. nur vor Funktions*definition*;
Funktionsdefinition muß i.a. im Header stehen (s. später)

```
inline double kreis::flaeche () ; // Definition  
{  
    return PI * radius * radius;  
}
```

Instanzmethoden

Aufruf:

- innerhalb der Klasse: über den Namen

flaeche()

- außerhalb der Klasse: über Objekt und Namen

kreis k;

double x = k.flaeche();

Objekt: Punktoperator **". "**

kreis* pk = new kreis;

double x = pk->flaeche();

Zeiger auf Objekt: Pfeiloperator **"->"**
(Dereferenzierung & Elementzugriff)

Instanzmethoden

Beispiel:

```
kreis k;           // Neues Kreis-Objekt erzeugen

k.set_radius(2.0); // Radius des Kreises festlegen

double f = k.flaeche(); // Kreisflaeche berechnen
```

Wie würde dieser Ausdruck bei prozeduraler Programmierung aussehen?

Unterschied:

- OOP: Das Objekt steht im Zentrum, die Funktion hängt nur dran
- Prozedurale Programmierung: Funktion steht im Zentrum, das Objekt wird nur an sie übergeben.

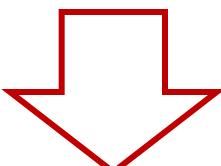
Instanzmethoden

Beispiel:

```
kreis k;           // Neues Kreis-Objekt erzeugen

k.set_radius(2.0); // Radius des Kreises festlegen

double f = k.flaeche(); // Kreisflaeche berechnen
```



```
double f = flaeche(k); // prozedurale Programmierung
```

Instanzmethoden

Beispiel:

```
kreis k;           // Neues Kreis-Objekt erzeugen  
  
k.set_radius(2.0); // Radius des Kreises festlegen  
  
double f = kreis::flaeche();
```

Frage: Funktioniert das auch? Begründung?

→ Nein, weil die Instanzmethoden außerhalb der Klasse nur an einem Objekt aufgerufen werden können. Ansonsten wäre nicht klar, für welches Objekt die Fläche berechnet werden soll.

Instanzmethoden

Beispiel:

```
kreis k;           // Neues Kreis-Objekt erzeugen

k.set_radius(2.0); // Radius des Kreises festlegen

double f = k.flaeche(); // Kreisflaeche berechnen
```



Warum brauchen wir `flaeche()` kein Argument zu übergeben?

Weil das Objekt (`k`), auf dem wir arbeiten, implizit in der Syntax enthalten ist.

- Wie funktioniert das technisch?
- Mit dem **this-Zeiger**

Der this-Zeiger

- Jedes neu erzeugte Objekt bekommt eigene Kopie der Instanzvariablen
 - Damit erst kann das Objekt individuelle Eigenschaften haben
- Die Instanzmethoden werden jedoch nicht in jedes Objekt hineinkopiert
- Sie stehen nur ein Mal im Arbeitsspeicher und werden von allen Objekten gemeinsam verwendet.
 - Woher "weiß" eine aufgerufene Instanzmethode, auf welche Instanzvariablen sie angewendet werden soll?

Der **this**-Zeiger

- Objekt bekommt bei Erzeugung einen Zeiger auf sich selbst zugewiesen
→ **this**-Zeiger
- Bei Aufruf einer Instanzmethode wird dieser **this**-Zeiger als *implizites* (also nicht ausdrücklich angegebenes) Argument übergeben.
- Werden in einer Instanzmethode Instanzvariablen verwendet, so werden automatisch die Variablen des Objektes verwendet, auf das **this** zeigt.

Beispiel:

```
double flaeche()
{
    return radius * radius * PI;
}
```

Der **this**-Zeiger

- Objekt bekommt bei Erzeugung einen Zeiger auf sich selbst zugewiesen
→ **this**-Zeiger
- Bei Aufruf einer Instanzmethode wird dieser **this**-Zeiger als *implizites* (also nicht ausdrücklich angegebenes) Argument übergeben.
- Werden in einer Instanzmethode Instanzvariablen verwendet, so werden automatisch die Variablen des Objektes verwendet, auf das **this** zeigt.

Beispiel:

```
double flaeche()
{
    return this->radius * this->radius * PI;
}
```

Technische Realisierung: Variable **radius** des Objektes, auf das **this** zeigt, wird verwendet (auch ohne ausdrückliche Angabe)

Member einer Klasse

Der this-Zeiger: Beispiel

Früheres Beispiel

```
class ABC                                abc.h  
{  
...  
public:  
    bool setC( ABC* p, int v );  
};
```

```
bool ABC::setC( ABC* p, int v )  
{  
    p->c = v;  
}
```

Programmierter Code

Vom Compiler
generierter Code

```
int main( ... )  
{  
    ABC abc;  
...  
    abc.setC( &abc, 5 );  
}
```

```
bool ABC::setC( ABC* const this, ABC* p, int v )  
{  
    p->c = v; // äquivalent zu this->c = v;  
}
```

Member einer Klasse

Der this-Zeiger: Beispiel

Früheres Beispiel: Vereinfachung der Elementfunktion

```
class ABC
{
...
public:
    bool setC( int v );
};
```

abc.h

abc.h

```
bool ABC::setC( int v )
{
    this->c = v;
}
```

Programmierter Code

```
int main( ... )
{
    ABC abc;
...
    abc.setC( 5 );
}
```

&abc

Vom Compiler
generierter Code

```
bool ABC::setC( ABC* const this, int v )
{
    this->c = v;
}
```

Member einer Klasse

Der this-Zeiger: Beispiel

Früheres Beispiel: Weitere Vereinfachung: "this->" weglassen

```
class ABC
{
    ...
public:
    bool setC( int v );
};
```

```
bool ABC::setC( int v )
{
    c = v;
}
```

abc.h
abc.h

Programmierter Code

```
int main( ... )
{
    ABC abc;
    ...
    abc.setC( 5 );
}
```

&abc

Vom Compiler
generierter Code

```
bool ABC::setC( ABC* const this, int v )
{
    this->c = v;
}
```

Der this-Zeiger

In einigen Fällen ist die Verwendung des this-Zeigers zwingend erforderlich:

- Auflösung einer Verdeckung von Instanzvariablen durch gleichnamige Parameter oder lokale Variablen.

Der this-Zeiger

In einigen Fällen ist die Verwendung des this-Zeigers zwingend erforderlich:

Beispiel:

```
void set_radius(double radius)
{
    // Hier soll die Instanzvariable radius auf
    // den im Parameter radius uebergebenen Wert
    // gesetzt werden.
}
```

Wie lösen wir den Namenskonflikt?

Der this-Zeiger

In einigen Fällen ist die Verwendung des this-Zeigers zwingend erforderlich:

Beispiel:

```
void set_radius(double radius)
{
    radius = radius;
}
```

Wie lösen wir den Namenskonflikt?

→ so jedenfalls nicht ...

Der this-Zeiger

In einigen Fällen ist die Verwendung des this-Zeigers zwingend erforderlich:

Beispiel:

```
void set_radius(double radius)
{
    this->radius = radius;
}
```

Wie lösen wir den Namenskonflikt?

→ Verwendung des this-Zeigers

Spezielle Instanzmethode: Der Konstruktor

Der **Konstruktor** ist eine (Instanz)Methode einer Klasse, die durch den vom Compiler generierten Code automatisch aufgerufen wird, wenn ein neues Objekt dieser Klasse **erzeugt** wird.

Aufgabe: **Objekt in einen "sinnvollen Anfangszustand" versetzen**, z.B.

- Initialisierung der Membervariablen
- Reservierung von dynamischem Speicher
- Öffnung externer Ressourcen (Dateien, Datenbankverbindungen, ...)
- ...

Der Konstruktor: Eigenschaften

- trägt denselben Namen wie die Klasse

Beispiel: Klasse **kreis** mit Konstruktor **kreis()**

- hat keinen Rückgabetyp (es wird auch nicht **void** angegeben)

Beispiel:

```
kreis() {radius = 1.0;}
```

- kann nicht direkt aufgerufen werden
- wird häufig überladen, um Objekte mit unterschiedlichen Anfangszuständen erzeugen zu können.

Beispiel:

```
kreis() {radius = 1.0;}
```

```
kreis(double r) {radius = r;}
```

Der Konstruktor: Objektinitialisierung

Konstruktor versetzt erzeugtes Objekt in sinnvollen Anfangszustand

Beispiel:

```
class zeit
{
    private:
        int std;
        int min;
        int sec;

    public:
        zeit (int h, int m, int s)
        {
            std = h;
            min = m;
            sec = s;
        }
        int getSec() { return sec; } 
```

The code defines a class `zeit` with private members `std`, `min`, and `sec`. It has a public constructor `zeit (int h, int m, int s)` that initializes these members. It also has a public member function `int getSec() { return sec; }`. A red brace groups the constructor code (from its definition to the closing brace) with the label "Konstruktor" in red. Another red brace groups the access function code with the label "Zugriffsfunktion" in red.

Der Konstruktor: Objektinitialisierung

```
class zeit
{
    private:
        int std;
        int min;
        int sec;

    public:
        zeit (int h, int m, int s)
        {
            std = h;
            min = m;
            sec = s;
        }
        int getSec() { return sec; }
};

int main(int argc, char *argv[])
{
    zeit t(2, 33, 15);
    cout << "Anzahl Sekunden: " << t.getSec() << endl;
    return 0;
}
```

Ausgabe: ?

Erzeugung des Objekts `t` der Klasse `zeit` durch Aufruf des überladenen Konstruktors

Der Konstruktor: Objektinitialisierung

```
class zeit
{
    private:
        int std;
        int min;
        int sec;

    public:
        zeit (int h, int m, int s)
        {
            std = h;
            min = m;
            sec = s;
        }
        int getSec() { return sec; }
};

int main(int argc, char *argv[])
{
    zeit t(2, 33, 15);
    cout << "Anzahl Sekunden: " << t.getSec() << endl;
    return 0;
}
```

Ausgabe:

Anzahl Sekunden: 15

Drücken Sie eine beliebige Taste . . .

Der Konstruktor: Konstruktorliste

Alternative Initialisierung über die so genannte Konstruktorliste:

- Fasst Schritte der Speicherplatzreservierung & Initialisierung zusammen
- Bringt Laufzeitvorteile bei größeren oder sehr vielen Objekten

Angabe einer Initialisierungsliste vor dem Codeblock des Konstruktors:

```
zeit (int h, int m, int s) : std(h), min(m), sec(s)
{
}
Name der Instanzvariablen   zugewiesener Wert
```

- *Konstante* Instanzvariablen *müssen* über die Initialisierungsliste initialisiert werden! (können trotzdem für jedes Objekt verschieden sein)

Konstruktor: Vorgabeargumente

- Konstruktor kann vorbelegte Parameter haben:

```
class zeit
{
    private:
        int std, int min, int sec;

    public:
        zeit( int h = 0, int m = 0, int s = 0 )
        { std = h; min = m; sec = s; }

};
```

```
int main(...)
{
    zeit t1(1,2,3); // 1 Stunde, 2 Minuten, 3 Sekunden
    zeit t2(5,7);   // 5 Stunden, 7 Minuten, 0 Sekunden
    zeit t3(2);     // 2 Stunden, 0 Minuten, 0 Sekunden
    zeit t4;         // Standard-Konstruktor-Funktionalität:
                    // 0 Stunden, 0 Minuten, 0 Sekunden
    ...
}
```

Konstruktor: Überladung

- Konstruktor kann überladen werden:

```
class zeit
{
    private:
        int std, int min, int sec;

    public:
        zeit( int h = 0,
              int m = 0,
              int s = 0 );
        zeit( const char *p );
};


```

```
zeit::zeit( int h, int m, int s )
{
    std = h; min = m; sec = s;
}

zeit::zeit( const char *p )
{
    sscanf(p, "%d:%d:%d",
           &std, &min, &sec);
}
```

```
int main(...)
{
    zeit t1(1,2);          // 1 Stunde, 2 Minuten, 0 Sekunden
    zeit t2("4:33:50");   // 4 Stunden, 33 Minuten, 50 Sekunden
    ...
}
```

Der Konstruktor: Aufruf

- Automatischer Aufruf bei der Erzeugung (Instanziierung) eines Objekts
- Bei Überladung wählt der Compiler den passenden anhand der übergebenen Argumente aus.
- Nach Erzeugung eines Objekts kann dessen Konstruktor nicht noch einmal aufgerufen werden.

Beispiele:

```
string str1; // Standardkonstruktor (naechste Seite)  
string str2("Hallo");  
string str3(str2);  
string* str4 = new string;
```

Standardkonstruktor

Ein Konstruktor ohne Aufrufparameter heißt Standardkonstruktor.

Beispiel:

```
kreis ()  
{  
    radius = 1.0;  
}
```

Objekterzeugung mit Standardkonstruktor (ohne Klammern!):

```
kreis k;  
kreis* k = new kreis;
```

Bei dynamisch erzeugten Objekten („new“) dürfen Funktionsklammern verwendet werden:
`kreis* k = new kreis();`

Standardkonstruktor

Frage:

Wie würde die folgende Anweisung interpretiert werden (Aufruf des Standardkonstruktors mit Klammern bei nicht dynamisch erzeugtem Objekt)?

```
kreis k();
```

Antwort:

Deklaration einer Funktion ohne Parameter mit Namen k, welche ein Kreis-Objekt zurück gibt.

Compiler-generierter Standardkonstruktor

Falls die Klasse keinen expliziten Konstruktor hat, generiert der Compiler automatisch einen (leeren!) Ersatzkonstruktor

→ Objekterzeugung möglich; Objekt ist aber nicht initialisiert

Kein expliziter Konstruktor in Klasse Kreis

```
class kreis
{
    private:
        double radius;

    public:
        void setRadius( double r );
};
```

```
int main(...)
{
    kreis k;
    k.setRadius( 1.0 );
    ...
    return 0;
}
```

Kreis k erzeugt,
aber nicht initialisiert

Kreis k initialisiert

Standardkonstruktor

Falls expliziter Konstruktor (mit Aufrufparametern) in der Klasse definiert ist
→ keine automatische Generierung eines Standardkonstruktors!

Expliziter Konstruktor mit Aufrufparametern
in Klasse Kreis vorhanden:

```
class kreis
{
private:
    double radius;

public:
    kreis( double r )
    {
        radius = r;
    }

    void setRadius( double r );
};
```

```
int main(...)
{
    kreis k;           // Fehler!
    kreis k( 1.0 ); // o.k.

    ...

    return 0;
}
```

Standardkonstruktor

Falls expliziter Konstruktor (mit Aufrufparametern) in der Klasse definiert ist

→ Standardkonstruktor explizit definieren oder

→ Vorgabeargumente

```
class kreis
{
    private:
        double radius;

    public:
        kreis() { radius = 1.0; }
        kreis( double r )
        {
            radius = r;
        }
        void setRadius( double r );
};
```

```
int main(...)
{
    kreis k;           // o.k.
    kreis k( 1.0 ); // o.k.

    ...
    return 0;
}
```

Standardkonstruktor

Falls expliziter Konstruktor (mit Aufrufparametern) in der Klasse definiert ist
→ Standardkonstruktor explizit definieren oder
→ Vorgabeargumente

```
class kreis
{
    private:
        double radius;

    public:
        kreis( double r = 1.0 )
        {
            radius = r;
        }
        void setRadius( double r );
};
```

```
int main(...)
{
    kreis k;           // o.k.
    kreis k( 1.0 );  // o.k.

    ...
    return 0;
}
```

Standardkonstruktor

In welchen Fällen existiert ein Standardkonstruktor?

- Wenn Programmierer einen Konstruktor ohne Parameter angegeben hat, unabhängig davon, ob es weitere Konstruktoren mit Parametern gibt.
- Wenn Programmierer *keinen* Konstruktor angegeben hat
→ Compiler-generierter Standardkonstruktor

Wann existiert **kein** Standardkonstruktor?

- Wenn Programmierer nur Konstruktoren mit Parametern angegeben hat.
→ Standardkonstruktor selbst definieren

Standardkonstruktor mit Vorgabeargumenten?

Frage:

Ist ein Konstruktor, bei dem alle Argumente mit Vorgabeargumenten versehen sind, ebenfalls ein Standardkonstruktor?

Antwort:

Ja, weil er ohne Argumente aufgerufen werden kann.

Standardkonstruktor mit Vorgabeargumenten

Frage:

Ist der folgende Konstruktor ein zulässiger Standardkonstruktor?

```
class kreis  
{  
public:  
    double radius;  
    kreis (double radius = 1.0) {}  
};
```

Antwort:

Ja, weil er ohne Argument aufgerufen werden kann.

Standardkonstruktor mit Vorgabeargumenten

Frage:

Funktioniert dieser Standardkonstruktor korrekt? D.h. tut er das Richtige?

```
class kreis  
{  
public:  
    double radius;  
    kreis (double radius = 1.0) {}  
};
```

Antwort:

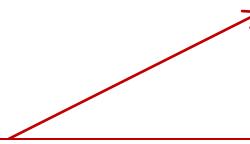
Nein, hier wird nur der Parameter des Konstruktors mit Namen `radius` auf den Wert 1.0 gesetzt. Die Instanzvariable `radius` wird aber nicht verändert.

Standardkonstruktor mit Vorgabeargumenten

Frage:

Was müssten wir verändern, damit er richtig funktioniert?

```
class kreis  
{  
public:  
    double radius;  
  
    kreis (double radius = 1.0) {this->radius = radius;}  
};
```



Erst durch diese Zuweisung wird dem Attribut **radius** der Wert des Parameters **radius** zugewiesen.

Konstruktor: Initialisierung von Objekt-Arrays

Problem:

Wie können wir die Elemente eines Arrays mit Hilfe des Konstruktors initialisieren?

Variante 1: Initialisierungsliste

In einer Initialisierungsliste wird für jedes Array-Element ein eigener Konstruktoraufruf ausgeführt.

Initialisierung von Objekt-Arrays

Variante 1: Initialisierungsliste

Zur Erinnerung ein Beispiel mit Int-Feld Initialisierung:

Beispiel:

```
int main()
{
    int a[3] = {1, 2, 4};
    for(int i = 0; i < 3; i++) {
        cout << a[i] << " " << endl;
    }

    system("PAUSE");
    return 0;
}
```

Initialisierungsliste setzt
Feldelemente auf:

a[0] = 1
a[1] = 2
a[2] = 4

Member einer Klasse

Beispiel mit einem Array-Objekt:

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

class demo
{
public:
    int wert1;
    int wert2;

    demo(int i, int j) {wert1 = i; wert2 = j;}
    void ausgeben() {cout << "wert1 = " << wert1 << " wert2 = "
                    << wert2 << endl;}
};

int main()
{
    demo obj[3] = {demo(1,2), demo(2,3), demo(3,4)};
    for(int i = 0; i < 3; i++) {
        obj[i].ausgeben();
    }
    return 0;
}
```

Ausgabe:

wert1 = 1 wert2 = 2
wert1 = 2 wert2 = 3
wert1 = 3 wert2 = 4
Drücken Sie eine beliebige Taste . . .



1. Objekt wird mit diesem Konstruktor initial.
2. Objekt mit diesem
3. Objekt mit diesem

Initialisierung von Objekt-Arrays

Problem:

Wie können wir die Elemente eines Arrays mit Hilfe des Konstruktors initialisieren?

Variante 2:

- Verzicht auf individuelle Initialisierung einzelner Objekte im Array
- Definition eines Standardkonstruktors, der vom Compiler automatisch zur Initialisierung aller Objekte im Array verwendet wird.

Member einer Klasse

Beispiel mit einem Array-Objekt:

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

class demo
{
public:
    int wert1;
    int wert2;

    demo() {wert1 = 0; wert2 = 1;}
    demo(int i, int j) {wert1 = i; wert2 = j;}
    void ausgeben() {cout << "wert1 = " << wert1 << " wert2 = " << wert2 << endl;}
};

int main()
{
    demo obj[3];

    for(int i = 0; i < 3; i++) {
        obj[i].ausgeben();
    }
    return 0;
}
```

Ausgabe:

wert1 = 0 wert2 = 1

wert1 = 0 wert2 = 1

wert1 = 0 wert2 = 1

Drücken Sie eine beliebige Taste . . .

Standardkonstruktor

Kein expliziter Konstruktor-Aufruf;
daher Initialisierung aller Objekte
im Array mit dem Standardkon-
struktur.

Spezielle Konstruktoren

Konstruktoren mit speziellen Parameterprofilen haben eigene Namen

- **Kopierkonstruktor** (Referenz auf 1 Parameter vom Typ der Klasse)

- Bsp.: `complex (const complex& orig); // Deklaration`
`complex::complex (const complex& orig); // Definition`
- siehe später

Verschieden von
Typ der Klasse

- **Typumwandlungskonstruktor** (mit 1 Parameter von anderem Typ aufrufbar)

- `klassenname(typ param);`
- `klassenname(typ1 par1, typ2 par2 = 0);`
- Dient zur Umwandlung anderer Datentypen in die gewünschte Klasse
- Bsp.: Umwandlung eines (2-dim.) Punktes in eine komplexe Zahl

Bei mehr als einem Parameter:
Vorgabeargument erforderlich!

`complex (const punkt x); // falls sinnvoll...`

Der Destruktor

Der **Destruktor** ist eine (Instanz)Methode einer Klasse, die durch den vom Compiler generierten Code automatisch aufgerufen wird, wenn ein neues Objekt dieser Klasse **aufgelöst** wird.

Aufgabe: **Objektauflösung und Speicherfreigabe**

Destruktoren werden automatisch aufgerufen

- Bei Verlassen des Gültigkeitsbereiches (für Objekte auf dem Stack)

Beispiel: Objekt wird in Funktion X definiert. Beendigung der Funktion X bedeutet Verlassen des Gültigkeitsbereichs des Objekts.

- Bei Anwendung des **delete**-Operators (für Objekte auf dem Heap)

Der Destruktor: Eigenschaften

- trägt denselben Namen wie die Klasse, mit vorangestellter Tilde
Beispiel: Klasse **kreis** mit Destruktor **~kreis()**
- hat keinen Rückgabetyp (es wird auch nicht **void** angegeben)
Beispiel: **~kreis() {}**
- hat keine Parameter und kann daher nicht überladen werden
- Falls kein eigener Destruktor für Klasse definiert wird: Compiler richtet automatisch einen Standarddestruktur mit leerem Anweisungsteil ein
- Bei dynamisch erzeugtem Array von Objekten → **delete[]** verwenden

```
char* pc = new char[5];  
  
delete[] pc;
```

Der Destruktor

Wann sollte ein eigener Destruktor definiert werden?

1. Falls das Objekt zusätzliche Ressourcen allokiert hat

Beispiel: dynamischer Speicher

externe Ressourcen (Dateien, Datenbankverbindungen, ...)

2. Falls beim Auflösen des Objektes „Abschlußarbeiten“ vorgenommen werden sollen

Beispiel: Dekrementieren eines Instanzzählers

3. Im Falle von Vererbung zur **virtual**-Deklaration (siehe später)

(der Standarddestruktur ist nicht **virtual** !)

Member einer Klasse

Der Destruktor: Beispiel

```
class zeit
{
    enum { MAX_LEN = 32 };

    int std;
    int min;
    int sec;

    char* pString;

public:
    zeit( ... );
    zeit( const char *p );
    ~zeit();
};
```

```
zeit::zeit( int h, int m, int s )
{
    std = h; min = m; sec = s;
    pString = new char[MAX_LEN];
    sprintf(pString, "%d:%d:%d", h,m,s);
}

zeit::zeit( const char *p )
{
    sscanf( p, "%d:%d:%d",
            &std, &min, &sec );
    pString = new char[MAX_LEN];
    strcpy( pString, p );
}

zeit::~zeit()
{
    delete[] pString;
}
```

- Destruktor der Klasse „zeit“ gibt den im Konstruktor dynamisch erzeugten Speicherplatz für den String wieder frei
- `delete[]`, da `pString` ein Array ist

Der Destruktor: Beispiele für Objekt-Arrays

```
int main( int argc, char* argv[] )  
{  
    zeit event[MAX];  
  
    for ( int i = 0; i < MAX; i++ )  
    {  
        ...  
        event[i].setTime(...);  
        ...  
    }  
    return 0; // Destruktor von event[MAX] bis event[0]  
};
```

Auflösen von Arrays:

Destruktoraufruf für *alle* Array-Elemente
(vom letzten bis zum ersten)

Dynamisch erzeugtes Array:

```
int main( int argc, char* argv[] )  
{  
    ...  
    zeit* pEvent = new zeit[MAX];  
    ...  
    delete[] pEvent; // Destruktor von pEvent[MAX] bis pEvent[0]  
    pEvent = NULL;  
};
```

Dynamisch allokierte Arrays:

delete[] → Destruktoraufruf für *alle*
Array-Elemente (vom letzten bis ersten)

Der Destruktor

Aufrufreihenfolge:

- Die Reihenfolge des Aufrufs der Destruktoren ist umgekehrt wie die der Konstruktoren.
D.h. der Destruktor des zuletzt erzeugten Objekts wird als erster ausgeführt.
- Hinweis: Der Konstruktor eines globalen Objekts wird vor der ersten Anweisung in main() aufgerufen.
→ Destruktoren der globalen Objekte werden als letztes aufgerufen

Beispiel: nächste Seite

Beispiel für Konstruktor-/Destruktor-Aufrufreihenfolge

```
int main( int argc, char* argv[] )
{
    zeit* pt5 = new zeit("1:1:1"); // Konstruktor von pt5
    zeit t1("1:2:3");           // Konstruktor von t1
    zeit t2("4:5:6");           // Konstruktor von t2
    ...
    if ( argc == 1 )
    {
        zeit t3("7:8:9");      // Konstruktor von t3
        zeit t4("10:11:12");   // Konstruktor von t4
        ...
    }                           // Destruktor von t4, dann von t3
    ...
    delete pt5;                // Destruktor von t5
    ...
    return 0;                  // Destruktor von t2, dann von t1
}
```

Vom Compiler zur Verfügung gestellte Konstruktoren

- Defaultkonstruktor (ggf., siehe vorn)

```
kreis();
```

- Defaultkopierkonstruktor (später)

```
kreis(const kreis& k);
```

- Defaultdestruktor

```
~kreis();
```

- Defaultzuweisungsoperator (später)

```
kreis k1(r), k2;
```

```
k2 = k1;
```

Klassenvariablen

- Zur Erinnerung: Instanzvariable, Instanzmethode:
 - jedes Objekt einer Klasse besitzt eigene Werte der Instanzvariablen und nutzt die Instanzmethoden der Klasse
 - zwei Objekte einer Klasse haben Variablen gleichen Namens, die verschiedene Werte haben und unterschiedliche Speicherbereiche belegen
- jetzt: Daten über die Menge aller Objekte einer Klasse speichern:
Klassenvariablen / -methoden (statische Datenelemente / Methoden)
- **Klassenvariablen:** beschreiben Eigenschaften, die zur *Klasse* gehören
 - d.h. die allen Objekten und keinem bestimmten Objekt zugeordnet sind
- sind wie globale Variablen, auf die jedes Klassenobjekt zugreifen kann; gehören zum Gültigkeitsbereich der Klasse
- Klassenvariablen werden in der Klasse deklariert und außerhalb definiert

Klassenvariablen

Definition mit dem Schlüsselwort **static**:

public:

static double PI;

- Klassenvariablen werden nur 1 Mal im Speicher als Teil ihrer Klasse angelegt, unabhängig davon, wie viele Objekte dieser Klasse erzeugt werden.
- Alle instanzierten Objekte können darauf zugreifen
- Änderungen sind dann aber auch für alle Objekte sichtbar
- Verwendung von Klassenvariablen ist nicht an das Vorhandensein von Objekten gebunden
- Klassenvariablen sind die einzigen Elemente einer Klasse, auf die die Klassenmethoden der Klasse zugreifen können (später)

Klassenvariablen

Klassenvariablen müssen außerhalb der Klasse (nochmal) definiert werden.

Erst dann wird Speicher für die Variable bereitgestellt.

Beispiel:

```
class kreis
{
public:
    double radius;
    static double PI;

    kreis() {}
    ~kreis() {}

};

double kreis::PI = 3.14159; // eigentliche Variablendefinition

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

*außerhalb
von main !*



Klassenvariablen

Sinnvoller wäre hier aber die Verwendung einer **konstanten Klassenvariablen**:

```
public:  
    static const double PI;
```

```
class kreis  
{  
public:  
    double radius;  
    static const double PI;  
  
    kreis() {}  
    ~kreis() {}  
};  
  
const double kreis::PI = 3.14159; // eigentliche Variablendefinition  
  
int main() { ... }
```

Klassenvariablen

Sinnvoller wäre hier aber die Verwendung einer **konstanten Klassenvariablen**

public:

static const double PI;

Frage:

Warum kann diese konstante Klassenvariable nicht im Konstruktor initialisiert werden?

Antwort:

Weil der Konstruktor nur ausgeführt wird, wenn auch eine Instanz (ein Objekt) erzeugt wird. Das ist möglicherweise nie der Fall. Trotzdem kann die Klassenvariable PI verwendet werden und muss daher initialisiert sein.

Klassenvariablen

Falls Klassenvariable vom Typ `const int` oder `const enum`, darf sie direkt in der Klassendefinition initialisiert werden:

Beispiel:

```
class kreis
{
public:
    double radius;
    static const int PI = 3;

    kreis() {}
    ~kreis() {}

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

Direkte Initialisierung



Klassenvariablen

Aufruf von Klassenvariablen innerhalb der Klasse über den Namen:

```
class kreis
{
public:
    double radius;
    static const double PI;

    double umfang()
    {
        return 2 * radius * PI;
    }

    double flaeche()
    {
        return radius * radius * PI;
    }
};
```

Klassenvariablen

Aufruf von Klassenvariablen außerhalb der Klasse über

- den vollqualifizierten Namen

```
int main() {  
  
    cout << " PI: " << kreis::PI << endl;      Klasse: Scope-Operator "::"  
    system("PAUSE");  
    return 0;  
}
```

Klassenvariablen

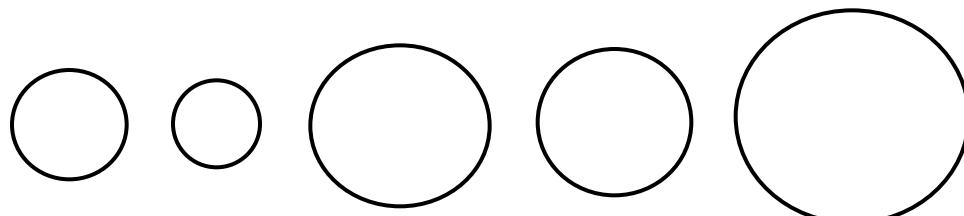
Aufruf von Klassenvariablen außerhalb der Klasse über

- den vollqualifizierten Namen oder ← deutlicher (da Klassenvariable!)
- über ein instanziertes Objekt

```
int main() {  
    kreis c;  
    kreis* pc = new kreis;  
  
    cout << " PI: " << kreis::PI << endl;  
    cout << " PI: " << c.PI << endl;           Objekt: Punktoperator ".."  
    cout << " PI: " << pc->PI << endl;       Zeiger auf Objekt: Pfeiloperator "->"  
    system("PAUSE");  
    return 0;  
}
```

Klassenvariablen

Beispiel: Klasse Kreis mit 5 Instanzen



Mit Hilfe der Instanzvariablen **radius** kann jedem Kreis eine eindeutige Identität gegeben werden.

Fragen:

- Wie viele Kopien der Instanzvariablen **radius** gibt es?
- Wie viele Kopien der Klassenvariablen **PI** gibt es?

Member einer Klasse

Klassenvariablen: Beispiel

```
class ABC          abc.h
{
private:
    int a;
    static int b;
    int c;

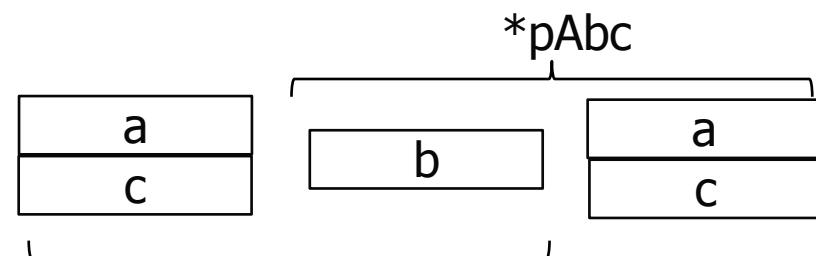
public:
    void setA(int v){a = v;}
    void setB(int v){b = v;}
    void setC(int v){c = v;}
}
```

```
int main( ... )           main.cpp
{
    ABC abc;
    ABC* pAbc = new ABC;

    ...
    abc.setA(5);      // abc.a = 5
    abc.setB(3);      // ABC::b = 3;
    abc.setC(2);      // abc.c = 2;
    pAbc->setC(7);  // pAbc->c = 7;
    pAbc->setB(6);  // ABC::b = 6;
    pAbc->setA(4);  // pAbc->a = 4;
    ...
}
```

int ABC::b; abc.cpp

Definition: Speicherplatz bereitstellen



Statische Variable b nur einmal im Speicher!

abc

Member einer Klasse

Klassenvariablen: Beispiel

```
class ABC          abc.h
{
private:
    int a;
    static int b;
    int c;

public:
    void setA(int v) {a = v;}
    void setB(int v) {b = v;}
    void setC(int v) {c = v;}
}
```

Instanzmethode

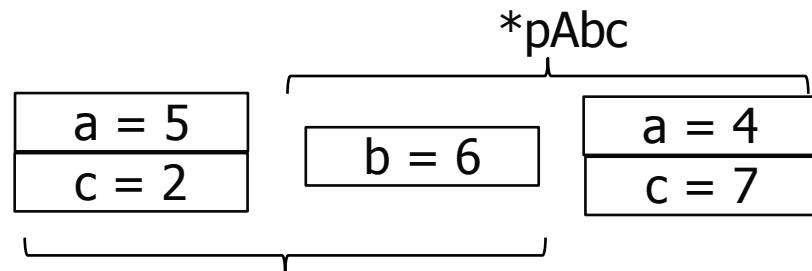
```
int main( ... )          main.cpp
{
    ABC abc;
    ABC* pAbc = new ABC;
    ...
    abc.setA(5);           // abc.a = 5
    abc.setB(3);           // ABC::b = 3;
    abc.setC(2);           // abc.c = 2;
    pAbc->setC(7);       // pAbc->c = 7;
    pAbc->setB(6);       // ABC::b = 6;
    pAbc->setA(4);       // pAbc->a = 4;
    ...
}
```

b ist 3

b ist 6

```
int ABC::b;          abc.cpp
```

b ist uninitialisiert



Statische Variable b nur einmal im Speicher!

Member einer Klasse

Klassenvariablen: Weiteres Beispiel

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

class kreis
{
    double radius;
    static const double PI;

public:
    kreis(double radius = 1.0);

    kreis::kreis(double radius) : radius(radius) {}

    const double kreis::PI = 3.1415927;

    int main(int argc, char* argv[])
    {
        ...
        // Ausgabe von PI?
        return 0;
    }
}
```

Deklaration der Klassenvariablen
(private!)

(Beispiel für Konstruktor
außerhalb Klassendefinition
mit Konstruktorliste)

Definition der Klassenvariablen
(Bereitstellung von Speicherplatz)
und Initialisierung
(außerhalb von `main!`)

PI ist definiert und existiert
innerhalb von `main`

später

Member einer Klasse

```
using namespace std;

class demo {
public:
    static int instanzen;

    demo() {instanzen++;} // Konstruktor
    ~demo() {instanzen--;} // Destruktor
};

int main(int argc, char *argv[])
{
    demo o1, o2, o3;

    cout << "Anzahl Objekte der Klasse demo: " << demo::instanzen <<
        endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Beispiel für die Verwendung
einer Klassenvariable

- Was tut die Klasse?
- Wo steckt noch ein Fehler?
- Was gibt das Programm aus?

Member einer Klasse

```
using namespace std;

class demo {
public:
    static int instanzen;

    demo() {instanzen++;} // Konstruktor
    ~demo() {instanzen--;} // Destruktor
};

int main(int argc, char *argv[])
{
    demo o1, o2, o3;

    cout << "Anzahl Objekte der Klasse demo: " << demo::instanzen <<
    endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Beispiel für die Verwendung
einer Klassenvariable

Klasse zählt, wie viele Objekte zu einem Zeitpunkt existieren. Da die Information unabhängig von einem konkreten Objekt ist, wird sie in einer Klassenvariable gespeichert.

Member einer Klasse

```
using namespace std;

class demo {
public:
    static int instanzen;

    demo() {instanzen++;} // Konstruktor
    ~demo() {instanzen--;} // Destruktor
};

int demo::instanzen = 0; // Fehler: Fehlende Definition der Klassenvariable

int main(int argc, char *argv[])
{
    demo o1, o2, o3;

    cout << "Anzahl Objekte der Klasse demo: " << demo::instanzen << endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Beispiel für die Verwendung
einer Klassenvariable

Der Fehler lag in der fehlenden Definition der Klassenvariable. Die obige Deklaration stellt noch keinen Speicher bereit. Dies muss außerhalb der Klasse nachgeholt werden.

Member einer Klasse

```
using namespace std;

class demo {
public:
    static int instanzen;

    demo() {instanzen++;} // Konstruktor
    ~demo() {instanzen--;} // Destruktor
};

int demo::instanzen = 0;

int main(int argc, char *argv[])
{
    demo o1, o2, o3; // Red box and arrow pointing here

    cout << "Anzahl Objekte der Klasse demo: " << demo::instanzen << endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Beispiel für die Verwendung
einer Klassenvariable

Es werden 3 Objekte der Klasse erzeugt. Daher gibt das Programm aus:

Anzahl der Objekte der Klasse demo: 3

Member einer Klasse

```
using namespace std;  
  
class demo {  
public:  
    static int instanzen; // Klassenvariable  
  
    demo() {instanzen++;} // Konstruktor  
    ~demo() {instanzen--;} // Destruktor  
};  
  
int demo::instanzen = 0;  
  
int main(int argc, char *argv[])  
{  
    ...  
}
```

Beispiel für die Verwendung
einer Klassenvariable

Beachte: statische Elemente i.a. **private**, nicht **public**

- benötigt (statische) Zugriffsfunktionen
- Klassenmethoden

Klassenmethoden

Klassenmethoden

- sind das Pendant zu Klassenvariablen
- werden mit dem Schlüsselwort **static** deklariert
 - Static steht nur vor der Deklaration *in* der Klasse
(also nicht mehr bei einer evt. Definition der Methode außerhalb der Klasse)
- gehören zur Klasse - nicht zum Objekt
 - existieren unabhängig von Objekten (d.h. möglicherweise auch, ohne dass ein einziges Objekt existiert)
 - verfügen über keinen “**this**”-Zeiger
 - direkter Zugriff daher nur auf Klassenvariablen (statische Datenelem.)
 - Zugriff auf Instanzvariablen: übergebe Objekt der Klasse als Parameter

Klassenmethoden

Beispiel für eine Definition innerhalb der Klasse:

```
class kreis
{
public:
    double radius;
    static const double PI;

    static double bogenZuWinkel (double bogen)
    {
        return bogen * 180.0 / PI;
    }
};
```

Klassenmethode, rechnet Bogenmaß
in Winkelmaß um

Klassenmethoden

Aufruf von Klassenmethoden

- innerhalb der Klasse (z.B. in anderen Klassenmethoden) über den Namen `bogenZuWinkel (2.0)`
- außerhalb der Klasse (z.B. in main) über den vollqualifizierten Namen:

```
cout << "Winkel von PI: " << kreis::bogenZuWinkel( kreis::PI );
```

Klasse: Scope-Operator "::
"

oder über ein Objekt:

```
kreis k;
```

Objekt: Punktoperator ". "

```
cout << "Winkel von PI: " << k.bogenZuWinkel( k.PI ) << endl;
```

```
kreis* pk = new kreis;
```

Zeiger auf Objekt: Pfeiloperator "->"

```
cout << "Winkel von PI: " << pk->bogenZuWinkel( pk->PI ) << endl;
```

deutlicher (da Klassenvariable!)

Member einer Klasse

Klassenmethoden: Beispiel

```
class ABC           abc.h
{
private:
    int a;
    static int b;
    int c;

public:
    void setA(int v){a = v;}
    static void setB(int v)
        {b = v;}
    void setC(int v){c = v;}
}
```

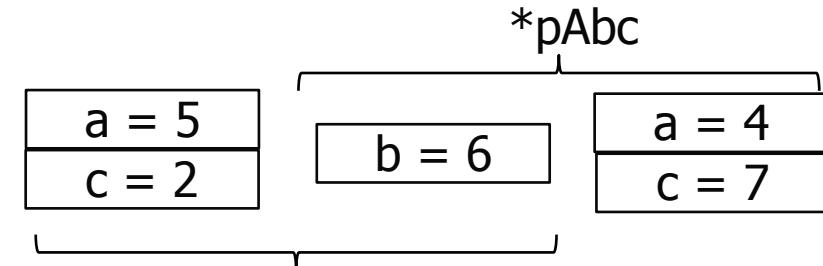
Klassenmethode

int ABC::b; abc.cpp

```
int main( ... )
{
    ABC abc;
    ABC* pAbc = new ABC;
    ...
    abc.setA(5); // abc.a = 5
    ABC::setB(3) // ABC::b = 3;
    abc.setC(2); // abc.c = 2;
    pAbc->setC(7); // pAbc->c = 7;
    ABC::setB(6); // ABC::b = 6;
    pAbc->setA(4); // pAbc->a = 4;
    ...
}
```

b ist 3

b ist 6



Statische Variable b nur einmal im Speicher!

abc

Quelle: Microconsult

393

Klassenmethoden: Weiteres Beispiel

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

class kreis
{
    double radius;
    static const double PI;

public:
    kreis(double radius = 1.0);
    static double getPI() { return PI; }
};

kreis::kreis(double radius) : radius(radius) {}

const double kreis::PI = 3.1415927;

int main(int argc, char* argv[])
{
    ...
    cout << kreis::getPI() << endl;
    return 0;
}
```

Definition der Klassenmethode

Aufruf der Klassenmethode

Klassenmethoden: Zugriff auf Datenelemente

- direkter Zugriff auf alle Klassenvariablen
- kein direkter Zugriff auf Instanzvariablen und Instanzmethoden

Warum nicht?

→ Klassenmethoden gehören nicht zu einer bestimmten Instanz und haben keinen `this`-Pointer

Stattdessen:

→ Objekt der Klasse (bzw. Referenz auf Objekt) muß als Parameter an die Klassenmethode übergeben werden

Klassenmethoden: Zugriff auf Datenelemente

Beispiel: Zugriff auf statische Datenelemente (Klassenvariable)

```
class kreis
{
public:
    double radius;
    static const double PI;

    static double bogenZuWinkel (double bogen)
    {
        return bogen * 180.0 / PI;
    }
};
```

- direkter Zugriff auf Klassenvariable PI

Klassenmethoden: Zugriff auf Datenelemente

Beispiel: Zugriff auf statische Datenelemente (Klassenvariable)

```
class kreis
{
public:
    double radius;
    static const double PI;

    static double bogenZuWinkel (double bogen)
    {
        return bogen * 180.0 / PI;
    }
};
```

- direkter Zugriff auf Klassenvariable PI

Klassenmethoden: Zugriff auf Datenelemente

Beispiel: Zugriff auf nicht-statische Datenelemente (Instanzvariable)

```
class kreis
{
public:
    double radius;
    static const double PI;

    static double bogenZuWinkel (double bogen)
    {
        return bogen * 180.0 / PI;
    }

    static double bogenlaenge( kreis& k, double bogen )
    {
        return bogen * k.radius;
    }
};
```

- direkter Zugriff auf Klassenvariable `PI`
- stattdessen: übergebe Referenz auf Objekt als Parameter; Zugriff auf Instanzvariable über Objekt

Klassenmethoden: Zugriff auf Datenelemente

Beispiel: Zugriff auf nicht-statische Datenelemente (Instanzvariable)

```
class kreis
{
    double radius; // radius privat
public:
    static const double PI;
    double getRadius(){ return radius; }
    static double bogenZuWinkel (double bogen)
    {
        return bogen * 180.0 / PI;
    }
    static double bogenlaenge( kreis& k, double bogen )
    {
        return bogen * k.radius;
    }
};
```

- direkter Zugriff auf Klassenvariable PI

- bei privatem Attribut: Zugriff innerhalb der Klasse gegeben

Klassenmethoden

Frage:

`bogenZuWinkel()` operiert nicht auf `kreis`-Objekten.

Warum haben wir sie dann überhaupt in der Klasse `kreis` definiert?

- Es handelt sich um eine **allgemeine** Hilfsmethode, die beim Arbeiten mit Kreisen nützlich ist.

Klassenmethoden

Frage:

Können Klassenmethoden den `this`-Zeiger verwenden? Begründung!

Antwort:

Nein, weil Klassenmethoden nicht zu einem bestimmten Objekt gehören, sondern zur Klasse. Sie können sogar aufgerufen werden, ohne dass ein einziges Objekt existiert.

Instanzmethode oder Klassenmethode?

Implementierung einer Funktion als Instanz- oder Klassenmethode?

- Beeinflussung des Objektzustands
- Ein-/Ausgabe von Objekt-spezifischen Infos

} → Instanzmethode

Allgemeine Objekt-unabhängige Berechnungen,
die beim Arbeiten mit der Klasse nützlich sind

} → Klassenmethode

Es gibt Zweifelsfälle, bei denen es kein Richtig und Falsch gibt.

Instanzmethode oder Klassenmethode?

Beispiel:

Entwerfen Sie eine Funktion `groesser()`, die zwei `Kreis`-Objekte (`x` und `y`) vergleicht und das Objekt mit dem größeren Radius zurückgibt.

- a. als Klassenfunktion (Vergleich zweier Kreise)
- b. als Instanzfunktion (Vergleich eines anderen Kreises mit sich selbst)

Wie sieht ein Aufruf der jeweiligen Methode aus ?

Hinweis:

Arbeiten Sie mit **Zeigern**, d.h. übergeben Sie den Funktionen lediglich Zeiger auf die Objekte und geben Sie als Ergebnis einen Zeiger auf den größeren der beiden Kreise zurück.

Instanzmethode oder Klassenmethode?

Beispiel:

Entwerfen Sie eine Funktion `groesser()`, die zwei `Kreis`-Objekte (`x` und `y`) vergleicht und das Objekt mit dem größeren Radius zurückgibt.

Klassenmethode

Definition:

`static` steht nur vor der Deklaration *in* der Klasse
(also nicht mehr bei einer evt. Definition der Methode außerhalb der Klasse)

```
static kreis* groesserer_kreis(kreis *a, kreis *b)
{
    if (a->radius > b->radius)
        return a;
    else
        return b;
}
```

Member einer Klasse

Beispiel Klassenmethode:

Verwendung:

```
int main(int argc, char *argv[])
{
    kreis *k1, *k2;
    kreis *groesserer;

    k1 = new kreis;
    k2 = new kreis;

    k1->set_radius(2.0);
    k2->set_radius(4.0);

    groesserer = kreis::groesserer_kreis(k1, k2);

    cout << "Radius groesserer Kreis = " << groesserer->get_radius() ;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Frage:

Wäre folgender Aufruf auch OK?

```
groesserer = k1->groesserer_kreis(k1, k2);
```

Antwort:

Ja! Klassenmethoden können auch über ein Objekt aufgerufen werden.

Instanzmethode oder Klassenmethode?

Beispiel:

Entwerfen Sie eine Funktion `groesser()`, die zwei `Kreis`-Objekte (`x` und `y`) vergleicht und das Objekt mit dem größeren Radius zurückgibt.

Instanzmethode: "Ist das übergebene Objekt größer als ich?"

Definition:

```
kreis* groesser(kreis *that)
{
    if (that->radius > this->radius)
        return that;
    else
        return this;
}
```

Member einer Klasse

Beispiel Klassenmethode:

Verwendung:

```
int main(int argc, char *argv[])
{
    kreis *k1, *k2;
    kreis *groesserer;

    k1 = new kreis;
    k2 = new kreis;

    k1->set_radius(2.0);
    k2->set_radius(4.0);

    groesserer = k1->groesser(k2); // oder k2->groesser(k1);

    cout << "Radius groesserer Kreis = " << groesserer->get_radius();
    return EXIT_SUCCESS;
}
```

Instanzvariablen oder Klassenvariable

Implementierung einer Variablen als Instanz- oder Klassenvariable?

Individuelle Eigenschaften des Objekts

→ Instanzvariable

Eigenschaften der Klasse, die zu allen
Objekten oder zur Klasse als Gesamtheit
gehören.

} → Klassenvariable

Instanzvariablen oder Klassenvariable

Beispielaufgabe:

Eine Klasse soll mit 2 Variablen, **zaehler** und **seriennummer**, ausgestattet sein, um jedem Objekt eine eindeutige Seriennummer geben zu können.

Siehe nächste Seite

Beispiel für die Verwendung von Klassen- und Instanzvariablen

```
class uhr
{
public:

    int zaehler;
    long seriенnummer;

    uhr()
    {
        // ??
    }

};

// Ggf. Variablen-Definition ??
```

Aufgabe: Vervollständigen Sie diese Klasse.

- Klassenvariable(n)?
- Konstruktor
- Ggf. Definition der Klassenvariablen

Member einer Klasse

Beispiel für die Verwendung von Klassen- und Instanzvariablen

```
class uhr
{
    public:
        static int zaehler;
        long seriennummer;

        uhr()
        {
            zaehler++;
            seriennummer = zaehler;
        }

};

int uhr::zaehler = 0; // Definition
```

Der Zähler ist nicht an ein einzelnes Objekt gebunden, sondern an die Klasse als Ganzes. Bei jedem Aufruf des Konstruktors wird der Zähler inkrementiert. Der aktuelle Stand wird als Seriennummer der Uhr verwendet.

Beispiel für die Verwendung von Klassen- und Instanzvariablen

```
class uhr
{
    public:

        static int zaehler;
        long seriенnummer;

        uhr()
        {
            zaehler++;
            seriенnummer = zaehler;
        }

};

int uhr::zaehler = 0; // Definition
```

Klassenvariable zaehler:

- gibt es nur 1 Mal
- zählt die Anzahl der Uhren
- muss außerhalb der Klasse definiert werden.

Instanzvariable seriенnummer:

- jedes Objekt hat eigene Var.
- dient der Identifikation

Konstruktor uhr():

- weist dem Objekt seine Nr. zu
- inkrementiert den Zähler

Beispiel für die Verwendung von Klassen- und Instanzvariablen

```
int main(int argc, char *argv[])
{
    uhr *c1 = new uhr();
    uhr *c2 = new uhr();

    cout << "Seriennummer Uhr 1: " << c1->seriennummer << endl;
    cout << "Seriennummer Uhr 2: " << c2->seriennummer << endl;
    cout << "Anzahl produzierter Uhren: " << uhr::zaehler << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Ausgabe:

Seriennummer Uhr 1: 1

Seriennummer Uhr 2: 2

Anzahl produzierter Uhren: 2

Drücken Sie eine beliebige Taste . . .

Definition der Memberfunktionen innerhalb & außerhalb der Klasse

Methoden (d.h. Instanz- und Klassenmethoden) können innerhalb und außerhalb der Klasse definiert werden

A. Definition innerhalb der Klasse

Kleine Funktionen sollten innerhalb der Klasse bereits definiert werden

- erhöhte Übersichtlichkeit
- Interpretation als `inline`-Funktion

B. Definition außerhalb der Klasse

Größere Funktionen sollten in der Klasse lediglich *deklariert* werden. Die eigentliche *Definition* (d.h. Angabe der Funktionsanweisungen im Funktionskörper) erfolgt dann außerhalb.

Code-Organisation: Aufteilung in Header- u. Implementierungsdatei

Headerdatei (*.hpp, *.h):

- Jede Klasse sollte in ihrer eigenen Datei definiert werden (Headerdatei)
 - Die Header-Datei enthält die reine **Klassendeklaration**, d.h. i.a. nur die **Funktionsprototypen** (Name, Rückgabewert, Parameterliste)
 - ggf. Definition von **inline**-Funktionen

Implementierungsdatei (*.cpp)

- Ergänzung fehlender **Definitionen** der Klasse
 - Funktionsdefinition: Angabe der Anweisungen des Funktionsrumpfes
 - Falls erforderlich: Definition von Klassenvariablen (s.o.)

Code-Organisation: Aufteilung in Header- u. Implementierungsdatei

Bemerkungen:

- Üblicherweise wird die Header-Datei eingebunden
(über include am Anfang der Implementierungsdatei)
- **inline**-Funktionen müssen i.a. in der **Header-Datei** definiert werden
- **Templates** müssen ebenfalls in Header-Datei definiert werden
 - Alternativ: Quelltextdatei mit Definitionen der Templates einbinden
- Bei Klassenmethoden wird das Schlüsselwort **static** nur in der Deklaration, nicht aber in der Implementierung angegeben.

Implementierungs- und Headerdatei

Header-Datei (*.hpp, *.h)

Beispiel: Headerdatei Mitarbeiter.hpp

```
class mitarbeiter
{
public:
    string name;
    int gehalt;

    mitarbeiter();
    mitarbeiter(string n, int s);

    void befoerdern(int betrag);
};
```

Deklaration des
überladenen Konstruktors

Deklaration der Instanz-
funktion **befoerdern()**

Implementierungs- und Headerdatei

Header-Datei (*.hpp, *.h)

Beispiel: **Headerdatei** Mitarbeiter.hpp

```
class mitarbeiter
{
public:
    string name;
    int gehalt;

    mitarbeiter();
    mitarbeiter(string n, int s);
    void befoerdern(int betrag);

};
```

Die Klasse enthält keine einzige
Funktionsanweisung!

Member einer Klasse

Implementierungsdatei

Beispiel: **Implementierungsdatei** Mitarbeiter.cpp

```
#include "Mitarbeiter.hpp"
```

Headerdatei einbinden

```
mitarbeiter::mitarbeiter()
{
    name = "unbekannt";
    gehalt = 1500;
}
```

Definition des
Standardkonstruktors

```
mitarbeiter::mitarbeiter(string n, int s)
{
    name = n;
    gehalt = s;
}
```

Definition des
überladenen Konstruktors

```
void mitarbeiter::befoerdern(int betrag)
{
    gehalt = gehalt + betrag;
}
```

Definition der Instanz-
funktion **befoerdern()**

Member einer Klasse

Anderes Beispiel: Klasse Kreis

1. Headerdatei kreis.hpp

```
class kreis
{
    private:
        double radius;
    public:
        static double linewidth;
    kreis () {radius = 1.0;}
    void setRadius (double r) {radius=r;}
    double umfang ();
    double flaeche ();
};
```

Member einer Klasse

Anderes Beispiel: Klasse Kreis

2. Implementierungsdatei kreis.cpp

```
#include "kreis.hpp"      Einfügen der Klasse aus der Headerdatei
```

```
double kreis::linewidth = 1.0;    Definition der (statischen) Klassenvariablen
```

```
double kreis::umfang()  
{  
    return 2.0 * radius * PI;  
}
```

Definition der Instanz-
funktion **umfang ()**

```
double kreis::flaeche()  
{  
    return radius * radius * PI;  
}
```

Definition der Instanz-
funktion **flaeche ()**

Member einer Klasse

Anderes Beispiel: Klasse Kreis

3. Hauptfunktion main.cpp

```
#include <iostream>
#include "kreis.hpp"      Einfügen der Klasse aus der Headerdatei

using namespace std;

int main(int argc, char *argv[])
{
    kreis k;

    k.setRadius(2.0);      Verwendung der Instanzmethode setRadius()
    cout << "Kreisumfang = " << k.umfang() << endl;
    cout << "Linienbreite = " << kreis::linewidth << endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Member einer Klasse

Anderes Beispiel: Klasse Kreis: Wie wird Methode "umfang()" inline?

Variante a) Funktionsdefinition in Klassendefinition (bei "kurzer" Methode)

Schlüsselwort **inline** nicht erforderlich (wird automatisch gesetzt)

1. Headerdatei kreis.hpp

```
class kreis
{
    private:
        double radius;

    public:
        ...
        double umfang() { return 2.0 * radius * PI; }
};
```

inline nur Empfehlung
an den Compiler!

Funktionsdefinition *innerhalb* der Klassendefinition

2. Implementierungsdatei kreis.cpp

Methode **kreis::umfang()** nicht vorhanden

Member einer Klasse

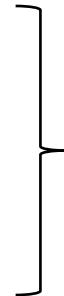
Anderes Beispiel: Klasse Kreis: Wie wird Methode "umfang()" inline?

Variante b) Funktionsdefinition außerhalb Klassendefinition (sonst)

Schlüsselwort **inline** erforderlich; Funktionsdef muß im Header stehen!

1. Headerdatei kreis.hpp

```
class kreis
{
    ...
    inline double umfang();
};
```



Klassendefinition

inline nur Empfehlung
an den Compiler!

```
inline double kreis::umfang()
{
    return 2.0 * radius * PI;
}
```

(Schlüsselwort **inline** i.a.
nur vor Funktionsdefinition)

Funktionsdefinition *außerhalb*
der Klassendefinition,
aber im Header-File

2. Implementierungsdatei kreis.cpp

Methode **kreis::umfang()** nicht vorhanden

Headerdatei: Problem des mehrfachen Einbindens

Falls Headerdatei von *mehreren* anderen Dateien eingebunden wird:

- Es kann Probleme beim Übersetzen geben
 - (z.B. mehrfache Definition, falls der Header Definitionen enthält)

Abhilfe: Präprozessor-Direktive `#ifndef ... #define`

```
// Datei header.h
#ifndef header_h
#define header_h

... // rest of header

#endif // header_h
```

Headerdatei: Problem des mehrfachen Einbindens

Beispiel:

```
// Datei kreis.hpp
#ifndef kreis_hpp
#define kreis_hpp
```

```
class kreis
{
private:
    double radius;
public:
    ...
    double umfang() { return 2.0 * radius * PI; }
};

#endif // kreis_hpp
```

1. Lesen von `kreis.hpp`:

- `#ifndef kreis_hpp` → TRUE, da `kreis_hpp` noch nicht definiert
- `#define kreis_hpp` definiert `kreis_hpp`
- Alles bis `#endif` wird gelesen

2. Lesen von `kreis.hpp`:

- `#ifndef kreis_hpp` → FALSE, da `kreis_hpp` bereits definiert
- Alles bis `#endif` wird ignoriert

Gültigkeitsbereich und Sichtbarkeit von Variablen

- Namen sind (nach vorheriger Deklaration) nur innerhalb des Blocks gültig, in dem sie deklariert wurden.
→ Lokal bezüglich des Blocks
- Namen von Variablen sind auch gültig für innerhalb des Blocks neu angelegte innere Blöcke.
- Die Sichtbarkeit wird aber eingeschränkt durch Deklarationen gleichen Namens in inneren Blöcken.

Sichtbarkeit und Kapselung

Gültigkeitsbereich und Sichtbarkeit von Variablen

Beispiel:

```
int main(int argc, char *argv[])
{
    int a = 1;

    { // Blockbeginn
        int b = 2;
        cout << a << b << endl;
        int a = 10;
        cout << a << b << endl;
    } // Blockende

    cout << a << endl;

    cout << b;  Fehler: b ist außerhalb obigen Blockes nicht definiert!

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Ausgabe (nach Fehlerbeseitigung):

```
12
102
1
```

Drücken Sie eine beliebige Taste

Ausgabe?

aus Breymann C++, © Hanser Verlag München

428

Sichtbarkeit und Kapselung

Gültigkeitsbereich und Sichtbarkeit von Variablen

Beispiel:

```
int main(int argc, char *argv[])
{
    int a = 1;

    { // Blockbeginn
        int b = 2;
        cout << a << b << endl; // 1 2
        int a = 10; // erstes a wird unsichtbar
        cout << a << b << endl; // 10 2
    } // Blockende

    cout << a << endl; // 1 (erstes a wieder sichtbar)

    cout << b; // Fehler!

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Programm mit Kommentaren

Sichtbarkeit und Kapselung

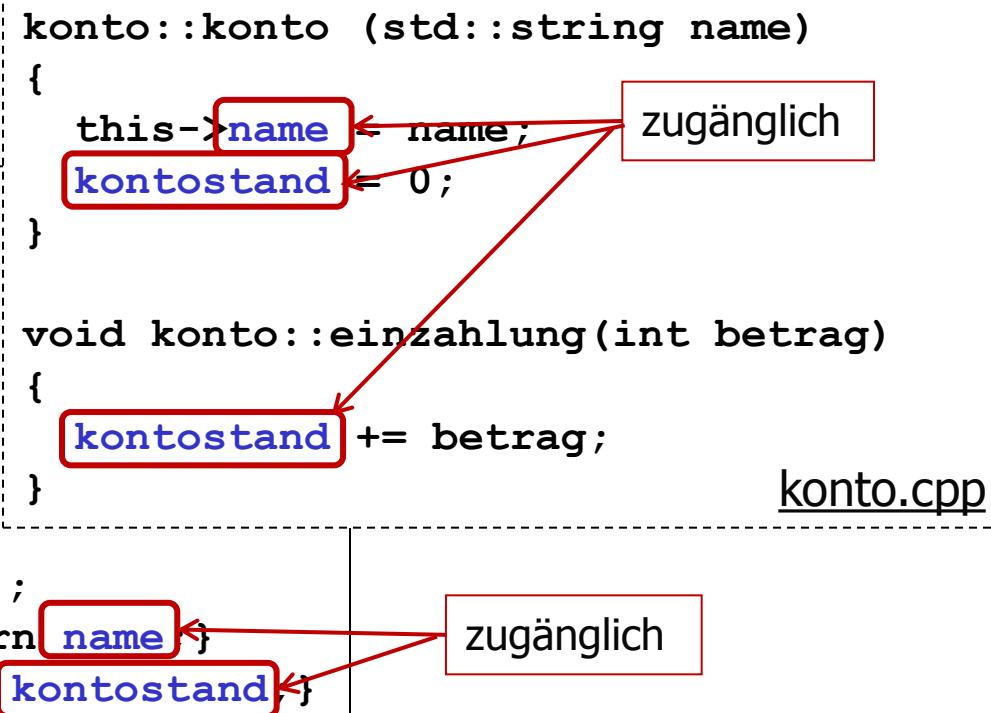
Die Klasse als Gültigkeitsbereich

- Alle Elemente einer Klasse haben die eigene Klasse zum Gültigkeitsbereich
 - Alle Klassenelemente sind im Code der Klasse gültig und verfügbar (ausgenommen eingeschlossene Typdefinitionen)
 - Verwendung z.B. in Memberfunktionen, Konstruktoren etc.
 - Zugriff in der Klasse direkt über den Namen

Beispiel:

```
class konto          konto.h
{
private:
    std::string name;
    int kontostand;

public:
    konto (std::string name);
    void einzahlung (int betrag);
    std::string getName() {return name;}
    int getKontostand() {return kontostand;}
};
```



Sichtbarkeit und Kapselung

Die Klasse als Gültigkeitsbereich

Der Zugriff von *außerhalb* der Klasse wird über **Zugriffsspezifizierer** geregelt:

private:

- Direkt zugänglich nur **innerhalb der eigenen Klasse**
- Zugriff von außerhalb nur über (public) Methoden
- Standard für alle Elemente, die ohne Modifizierer deklariert werden.
- Zweck:
 - Schutz vor unberechtigter Veränderung / unberechtigtem Lesen
 - Geheimhaltung von Implementierungsdetails

protected: (wichtig bei Vererbung → später)

- Zugriff **innerhalb der eigenen sowie innerhalb aller abgeleiteten Klassen**
- Zugriff von außerhalb nur über (public) Methoden

public:

- Freier Zugriff auf die entsprechende Variable oder Funktion von überall

Sichtbarkeit und Kapselung

Die Klasse als Gültigkeitsbereich: Beispiel

```
class konto
{
    private:
        std::string name;
        int kontostand;

    public:
        konto (std::string nam)
        void einzahlung (int b)
        std::string getName()
        int getKontostand() {r
};
```

```
konto::konto (std::string name)
{
    this->name = name;
    kontostand = 0;
}

void konto::einzahlung(int betrag)
{
    kontostand += betrag;
}

}

```

konto.cpp

The diagram illustrates variable accessibility in the `konto.cpp` file. It shows two code snippets: the constructor and the `einzahlung` method. In the constructor, `this->name` and `kontostand` are highlighted with red boxes. Arrows point from these boxes to a red-bordered box containing the German word "zugänglich". In the `einzahlung` method, `kontostand` is highlighted with a red box and has an arrow pointing to the "zugänglich" box. The entire code block is enclosed in a dashed box.

```
int main(int argc, char *argv[])
{
    konto k("Max Mustermann"),
    cout << k.name << endl;          // Fehler
    cout << k.kontostand << endl; // Fehler
    cout << k.getName() << ": " << k.getKontostand() << endl; // o.k.

}
```

nicht zugänglich (da private) main.cpp

zugänglich (über public-Methode eines Objektes)

Kapselung

- Die privaten (und teilweise die protected) Klassenelemente (Attribute, Methoden) werden vor anderen Teilen des Programms verborgen: „**Kapselung**“
- Zugriff auf Objektdaten nur möglich über seine public Zugriffsmethoden
→ **Schnittstelle (Interface)**
- Das Interface eines Objekts sollte einfach und zweckmäßig gestaltet sein. Die inneren Abläufe des Objekts sollten privat sein.
- Kapselung lässt ein Objekt wie eine **Blackbox** aussehen: Das Innere einer Box ist vor Blicken geschützt. Von außen gibt es einige Steuerelemente, die für den Anwender die einzige Möglichkeit darstellen, die Box zu verwenden.

Beispiel: Fernsehgerät

- Benutzer hat keinen Zugang zu den meisten inneren Abläufen
- Benutzer kommuniziert mit dem Gerät über wohl definierte Steuerelemente

Kapselung von Attributen

Frage:

Wozu ist diese "Privatisierung" nötig, wenn wir über den Umweg Zugriffsmethode dann doch den Zugriff ermöglichen?

Antwort:

Durch die Zugriffsmethoden behält das Objekt die Kontrolle über seine privaten Daten und kann jeden Zugriff auf diese kontrollieren.

Sichtbarkeit und Kapselung

Beispiel für versuchten Zugriff auf gekapseltes Attribut:

```
class konto {  
    private:  
        string name;  
        int kontonummer;  
        int kontostand;  
        static int kontenzaehler;  
  
    public:  
        konto (string name);  
        void einzahlung (int betrag);  
        int getKontostand() {return kontostand;}  
        // ...  
};  
  
int main (void) {  
    konto k("max mustermann");  
  
    k.kontostand *= 2;  
  
    cout << k.kontostand << endl;  
  
    system("PAUSE");  
    return 0;  
}
```

Sichtbarkeit und Kapselung

Beispiel für versuchten Zugriff auf gekapseltes Attribut:

Compiler-Meldung:

konto_main.cpp: In function `int main():`

konto_main.cpp:13: error: `int konto::kontostand' is private

konto_main.cpp:85: error: within this context

make.exe: *** [konto_main.o] Error 1

```
int main (void) {  
    konto k("max mustermann");  
  
    k.kontostand *= 2; // Line 13  
  
    cout << k.kontostand << endl; // Line 85  
  
    system("PAUSE");  
    return 0;  
}
```

Geheimhaltung von Funktionen

- Eine *private Funktion* kann nur von anderen Funktionen der gleichen Klasse verwendet werden.
- Für Programmteile außerhalb der entsprechenden Klasse ist eine private Methode nicht sichtbar.

Übung:

Eine Bank möchte als Sicherheitsmaßnahme aufzeichnen, wie häufig auf ein Konto zugegriffen (hier: eingezahlt) wird. Erweitern Sie die Klasse Konto um eine Zählvariable **zaehler** (Klassen- oder Instanzvariable? Privat oder öffentlich/public?) und eine Methode **inkrementzaehler** (Klassen- oder Instanzmethode? Privat oder öffentlich/public?), welche bei jeder Einzahlung **zaehler** inkrementiert.

Sichtbarkeit und Kapselung

Übung:

Eine Bank möchte als Sicherheitsmaßnahme aufzeichnen, wie häufig auf ein Konto zugegriffen (hier: eingezahlt) wird. Erweitern Sie die Klasse Konto um eine Zählvariable **zaehler** (Klassen- oder Instanzvariable? Privat oder öffentlich/public?) und eine Methode **inkrementZaehler** (Klassen- oder Instanzmethode? Privat oder öffentlich/public?), welche bei jeder Einzahlung **zaehler** inkrementiert.

```
class konto {  
    private:  
        string name;  
        int kontonummer;  
        int kontostand;  
        static int kontenzaehler;  
  
    public:  
        konto (string name);  
        void einzahlung (int betrag);  
        int getKontostand() {return kontostand;}  
};
```

Sichtbarkeit und Kapselung

Frage:

Variable `zaehler`:

- Klassen- oder Instanzvariable?
- Privat oder öffentlich/public?

Antwort:

- Instanzvariable, weil für jedes Konto einzeln gezählt werden soll.
- Privat, weil aus Sicherheitsgründen von extern nicht darauf zugegriffen werden darf.

Sichtbarkeit und Kapselung

Frage:

Funktion `inkrementZaehler()`:

- Klassen- oder Instanzfunktion?
- Privat oder öffentlich/public?

Antwort:

- Instanzfunktion, weil Objektattribut (`zaehler`) verändert werden soll.
- Privat, weil es sich um eine interne Zählung handelt, die von außen nicht beeinflusst werden darf.

Sichtbarkeit und Kapselung

```
class konto {  
    private:  
  
        string name;  
        int kontonummer;  
        int kontostand;  
        int zugriffsZaehler;  
        static int kontenZaehler; // externe Def. und Initialisierung  
  
        void inkrementZaehler() {zugriffsZaehler += 1;}  
  
    public:  
  
        konto (string name)  
        {  
            this->name = name;  
            kontonummer = kontenZaehler++;  
            kontostand = 0;  
            zugriffsZaehler = 0;  
        }  
  
        void einzahlung (int betrag)  
        {  
            kontostand += betrag;  
            inkrementZaehler();  
        }  
  
        int getKontostand() { return kontostand; }  
};
```

Lösungsvorschlag

Sichtbarkeit und Kapselung

```
class konto {  
    private:  
        string name;  
        int kontonummer;  
        int kontostand;  
        int zugriffsZaehler;  
        static int kontenZaehler; // externe Def. und Initialisierung  
        void inkrementZaehler() {zugriffsZaehler += 1;}  
  
    public:  
        // ...  
};  
  
int konto::kontenZaehler = 0;  
  
int main (int argc, char* argv[])  
{  
    konto k("max mustermann");  
    k.einzahlung(200);  
    k.inkrementZaehler();  
  
    cout << k.getKontostand() << endl;  
}  
return 0;
```

Versuch eines externen Zugriffs auf die private Funktion:

Compiler-Meldung:

```
konto_main.cpp: In function `int main()':  
konto_main.cpp:17: error: `void konto::inkrementZaehler()' is private  
konto_main.cpp:99: error: within this context  
make.exe: *** [konto_main.o] Error 1
```

Sichtbarkeit und Kapselung

```
class konto {  
    private:  
        string name;  
        int kontonummer;  
        int kontostand;  
        int zugriffsZaehler;  
        static int kontenZaehler; // externe Def.  
  
    void inkrementZaehler() { zugriffsZaehler++; }  
  
    public:  
        konto (string name)  
        {  
            this->name = name;  
            kontonummer = kontenZaehler++;  
            kontostand = 0;  
            zugriffsZaehler = 0;  
        }  
  
        void einzahlung (int betrag)  
        {  
            kontostand += betrag;  
            inkrementZaehler();  
        }  
  
        int getKontostand() { return kontostand; }  
};
```

Frage:

Was genau zählt die Variable zugriffsZaehler?

- Die Aufrufhäufigkeit für ein einzelnes Objekt?
- Die Aufrufhäufigkeit für die gesamte Klasse?

Initialisierung

Antwort:

Die Aufrufhäufigkeit für einzelnes Objekt, weil jedes Objekt eine eigene Variable zugriffsZaehler hat.

Sichtbarkeit und Kapselung

Beispiel zu den Zugriffsspezifizierern

```
class A
{
    public:
        int pubA;
    protected:
        int proA;
    private:
        int privA;
};
```

```
class B
{
    public:
        int pubB;
    protected:
        int proB;
    private:
        int privB;
};
```

Sichtbarkeit und Kapselung

Beispiel zu den Zugriffsspezifizierern

```
class C : A
{
    public:
        int pubC;
    protected:
        int proC;
    private:
        int privC;
        B     objB;
```

Klasse C wird von Klasse A abgeleitet

"Eingebettetes Objekt" der Klasse B.

```
class A
{
    public:
        int pubA;
    protected:
        int proA;
    private:
        int privA;
};
```

```
class B
{
    public:
        int pubB;
    protected:
        int proB;
    private:
        int privB;
};
```

// Zugriff auf Elemente der eigenen Klasse
privC = 1;

Erlaubt?

Ja, weil es sich um ein Element dieser Klasse handelt.

Sichtbarkeit und Kapselung

Beispiel zu den Zugriffsspezifizierern

```
class C : A
{
    public:
        int pubC;
    protected:
        int proC;
    private:
        int privC;
        B     objB;
```

Klasse C wird von Klasse A abgeleitet

"Eingebettetes Objekt" der Klasse B.

```
class A
{
    public:
        int pubA;
    protected:
        int proA;
    private:
        int privA;
};
```

```
class B
{
    public:
        int pubB;
    protected:
        int proB;
    private:
        int privB;
};
```

// Zugriff auf Elemente der eigenen Klasse

```
privC = 1;
```

```
proC = 1;
```

Erlaubt?

Ja, weil es sich um ein Element dieser Klasse handelt.

Sichtbarkeit und Kapselung

Beispiel zu den Zugriffsspezifizierern

```
class C : A
{
    public:
        int pubC;
    protected:
        int proC;
    private:
        int privC;
        B     objB;

public:
    void demoFunc()
    {
        // Zugriff auf Elemente der eigenen Klasse
        privC = 1;
        proC = 1;
        pubC = 1;
    }
}
```

Klasse C wird von
Klasse A abgeleitet

"Eingebettetes Objekt"
der Klasse B.

Erlaubt?
Ja, weil es sich um ein Element dieser Klasse handelt.

```
class A
{
    public:
        int pubA;
    protected:
        int proA;
    private:
        int privA;
};
```

```
class B
{
    public:
        int pubB;
    protected:
        int proB;
    private:
        int privB;
};
```

Sichtbarkeit und Kapselung

Beispiel zu den Zugriffsspezifizierern

```
class C : A
{
    public:
        int pubC;
    protected:
        int proC;
    private:
        int privC;
        B     objB;

public:
    void demoFunc()
    {
        // Zugriff auf Elemente der eigenen Klasse
        privC = 1;
        proC = 1;
        pubC = 1;

        // Zugriff auf geerbte Elemente der Klasse A
        privA = 1;
    }
}
```

Klasse C wird von
Klasse A abgeleitet

"Eingebettetes Objekt"
der Klasse B.

Erlaubt?

Nein, weil es sich um ein privates Element
einer anderen Klasse handelt.

```
class A
{
    public:
        int pubA;
    protected:
        int proA;
    private:
        int privA;
};
```

```
class B
{
    public:
        int pubB;
    protected:
        int proB;
    private:
        int privB;
};
```

Sichtbarkeit und Kapselung

Beispiel zu den Zugriffsspezifizierern

```
class C : A
{
    public:
        int pubC;
    protected:
        int proC;
    private:
        int privC;
        B     objB;
```

Klasse C wird von Klasse A abgeleitet

```
public:
    void demoFunc()
    {
        // Zugriff auf Elemente der eigenen Klasse
        privC = 1;
        proC = 1;
        pubC = 1;

        // Zugriff auf geerbte Elemente der Klasse A
        privA = 1;
        proA = 1;
```

Erlaubt?

Ja, weil es sich um ein geerbtes protected-Element handelt.

```
class A
{
    public:
        int pubA;
    protected:
        int proA;
    private:
        int privA;
};
```

```
class B
{
    public:
        int pubB;
    protected:
        int proB;
    private:
        int privB;
};
```

Sichtbarkeit und Kapselung

Beispiel zu den Zugriffsspezifizierern

```
class C : A
{
    public:
        int pubC;
    protected:
        int proC;
    private:
        int privC;
        B     objB;

public:
    void demoFunc()
    {
        // Zugriff auf Elemente der eigenen Klasse
        privC = 1;
        proC = 1;
        pubC = 1;

        // Zugriff auf geerbte Elemente der Klasse A
        privA = 1;
        proA = 1;
        pubA = 1;
    }
}
```

Klasse C wird von
Klasse A abgeleitet

"Eingebettetes Objekt"
der Klasse B.

```
class A
{
    public:
        int pubA;
    protected:
        int proA;
    private:
        int privA;
};
```

```
class B
{
    public:
        int pubB;
    protected:
        int proB;
    private:
        int privB;
};
```

Erlaubt?

Ja, weil es sich um ein public-Element handelt.

```
// ... Fortsetzung naechste Seite
```

aus: Louis, Objektorientiert in C++ 450

Sichtbarkeit und Kapselung

Beispiel zu den Zugriffsspezifizierern

```
class C : A
{
    public:
        int pubC;
    protected:
        int proC;
    private:
        int privC;
        B    objB;

public:
    void demoFunc()
    {
        // ...
        // Fortsetzung von letzter Seite

        // Zugriff auf Elemente eines Objekts der Klasse B
        objB.privB = 1;
        objB.proB = 1;
        objB.pubB = 1;
    }
};
```

```
class A
{
    public:
        int pubA;
    protected:
        int proA;
    private:
        int privA;
};
```

```
class B
{
    public:
        int pubB;
    protected:
        int proB;
    private:
        int privB;
};
```

Erlaubt?

Nein, weil es sich um ein privates Element einer anderen Klasse handelt.

aus: Louis, Objektorientiert in C++
451

Sichtbarkeit und Kapselung

Beispiel zu den Zugriffsspezifizierern

```
class C : A
{
    public:
        int pubC;
    protected:
        int proC;
    private:
        int privC;
        B    objB;
```

```
public:
    void demoFunc()
    {
        // ...
        // Fortsetzung von letzter Seite

        // Zugriff auf Elemente eines Objekts der Klasse B
        objB.privB = 1;
        objB.proB = 1; ← Erlaubt?
        objB.pubB = 1;
    }
};
```

```
class A
{
    public:
        int pubA;
    protected:
        int proA;
    private:
        int privA;
};
```

```
class B
{
    public:
        int pubB;
    protected:
        int proB;
    private:
        int privB;
};
```

aus: Louis, Objektorientiert in C++
452

Sichtbarkeit und Kapselung

Beispiel zu den Zugriffsspezifizierern

```
class C : A
{
    public:
        int pubC;
    protected:
        int proC;
    private:
        int privC;
        B    objB;

public:
    void demoFunc()
    {
        // ...
        // Fortsetzung von letzter Seite

        // Zugriff auf Elemente eines Objekts der Klasse B
        objB.privB = 1;
        objB.proB = 1;
        objB.pubB = 1;
    }
};
```

```
class A
{
    public:
        int pubA;
    protected:
        int proA;
    private:
        int privA;
};
```

```
class B
{
    public:
        int pubB;
    protected:
        int proB;
    private:
        int privB;
};
```

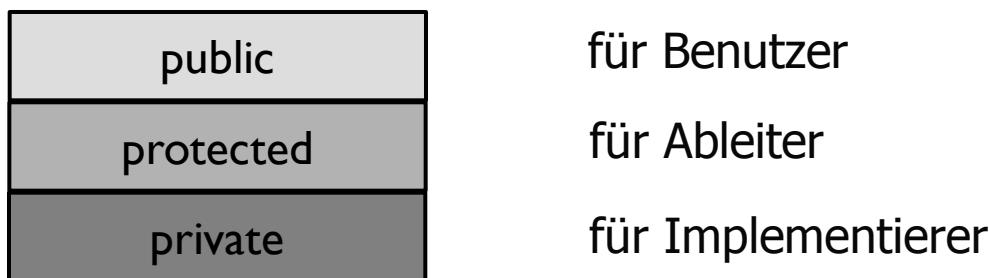
Erlaubt?

Ja, weil es sich um ein public-Element handelt.

Verwendung der Zugriffsspezifizierer

Einige Grundregeln:

- Elemente, die der internen Implementierung der Klasse dienen: **private**
- Elemente, die für die Programmierung mit Objekten nötig sind: **public**
- Membervariablen:
 - Vorzugsweise **private**
 - Zugriff über **public** Memberfunktionen
- Bei Vererbung: Falls abgeleiteten Klassen der direkte Zugriff auf Klassen-elemente gewährt werden soll: **protected** (sparsam verwenden!)



friends

Dieses Schlüsselwort ermöglicht **Umgehung der Zugriffsberechtigungen**.

- Freundfunktionen einer Klasse sind keine Methoden der Klasse, haben aber die gleichen Zugriffsrechte wie diese
 - **direkter Zugriff auf Klassenvariablen**, auch **private / protected**

Vorgehen:

1. In der Klasse, **die den Zugriff gewähren soll**, wird Funktion als **friend** deklariert (Klasse erklärt Funktion zur Freundfunktion, nicht umgekehrt)
2. Freundfunktion wird nicht wie Instanzmethode von Objekten aufgerufen, sondern man muß der Freundfunktion eine Referenz oder ein Objekt der Klasse übergeben
3. Anwendung: z.B. bei Überladung von Operatoren (später)

Sichtbarkeit und Kapselung

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

class demo {
    int wert;
    friend int func(demo &objekt);
};

int func(demo &objekt) {
    objekt.wert = 111;
    return objekt.wert;
}

int main() {
    demo d;
    cout << func(d);
    return 0;
}
```

friend-Deklaration der Funktion

Übergabe einer Objekt-Referenz an die Funktion.

friend-Funktion hat Zugriff auf privates Element

Aufruf der Funktion

Sichtbarkeit und Kapselung

```
#include <iostream>

using namespace std;

class demo {

    int wert;

    friend int func(demo &objekt);

};

int func(demo &objekt) {

    objekt.wert = 111;

    return objekt.wert;

}

int main() {

    demo d;

    cout << func(d);

    return 0;
}
```

Ausgabe?

111

Drücken Sie eine beliebige Taste . . .

friends

Bemerkungen:

- friend-Deklarationen werden nicht vererbt
- Wird eine ganze Klasse als friend markiert, erhalten alle enthaltenen Funktionen diese Deklaration.
→ Beispiel: nächste Seite

Sichtbarkeit und Kapselung

friends

```
class demo {  
    int wert;  
    friend class demo2;  
};
```

```
class demo2 {  
public:  
    int func2(demo &objekt) {  
        objekt.wert = 123;  
        return objekt.wert;  
    }  
};
```

```
int main() {  
    demo d;  
    demo2 d2;  
    cout << d2.func2(d) << endl;  
    return 0;  
}
```

Klasse **demo2** ist "friend" der Klasse **demo**. Daher haben alle Funktionen in **demo2** die **friend**-Deklaration und dürfen Änderungen am privaten Attribut **wert** eines **demo**-Objektes (hier **d**) durchführen.

Klassen-Design

- Es gibt keine festen Regeln für gutes Klassendesign
- Es gibt aber allgem. Richtlinien, die auf bewährten Techniken beruhen

Abweichungen von diesen Richtlinien sind je nach persönlicher Vorliebe und Problemstellung möglich, sollten aber gut begründbar sein.

Allgemein: Klassen sollten

- einfach
- intuitiv
- sicher

verwendbar sein

Klassen-Design

1. Klassen sollten einfach verwendbar sein

"Information hiding": Egal, wie viel Know-How in der Implementierung steckt, der Benutzer der Klasse sollte davon nichts merken.

Beispiel: Unterhaltungselektronik

- public-Elemente sind die Bedienelemente der Elektronik
- private-Elemente sind die interne Implementierung

2. Klassen sollten intuitiv verwendbar sein

Ziel: Benutzer der Klasse sollte diese verwenden können, ohne seitenlange Beschreibungen lesen zu müssen.

- Am Namen der Klasse und den public-Elementen sollte weitgehend erkennbar sein, was sie tun.
- (Dokumentation ist natürlich trotzdem notwendig)

Klassen-Design

3. Klassen sollten sicher verwendbar sein

Idealerweise kann Benutzer eines Objekts dessen öffentliche Elemente in beliebiger Reihenfolge und mit beliebigen Argumenten aufrufen, ohne es in einen ungültigen Zustand zu versetzen.

Ungültiger Zustand: Unerlaubte Werte der Attribute

Beispiel:

Variable enthält Zählerstand eines 3-stelligen Zählers

→ unerlaubter Wert wäre ein Wert ≥ 1000

Klassen-Design

Festlegung der **Zugriffsspezifizierer** (private, protected, public):

Annahme:

- Alle Membervariablen und –funktionen sind bereits bekannt
- Zugriffsrechte müssen noch verteilt werden

Vorgehen:

1. Alle Elemente, die für die Arbeit mit den Objekten benötigt werden → public
alle anderen Elemente → private
2. Für alle public-Elemente entscheiden: stellt direkter Zugriff Sicherheitsrisiko dar?
Falls ja: Zugriffsfunktion (set-Fkt.) einrichten, die kontrollierten Zugriff ermöglicht.
3. Für alle private-Elemente entscheiden: in protected abschwächen, um Programmierern abgeleiteter Klassen mehr Gestaltungsfreiraum zu geben?

Klassen-Design

Konstruktor

Empfehlungen zum Konstruktor-Design:

- Jede Klasse sollte mindestens einen selbstdefinierten Konstruktor haben
- Der Konstruktor sollte wenigstens allen Membervariablen definierte Anfangswerte zuweisen.
- Die Initialisierung der Membervariablen sollte aus Effizienzgründen in der Konstruktorliste durchgeführt werden (und nicht im Funktionskörper)
- In allen (überladenen) Konstruktoren der Klasse sollten alle Membervariablen initialisiert werden.
- Jede Klasse sollte einen Standardkonstruktor (ohne Parameter) besitzen.
Grund: erleichterte Einrichtung von Arrays & Membervariablen vom Typ d. Klasse

Weitere Design-Techniken

const-Instanzmethoden

Alle (Instanz)Methoden, die keine Änderungen an den Membervariablen ihrer Klasse vornehmen, sollten als **const** deklariert werden.

Konstante Instanzmethoden

- dürfen keine Membervariablen verändern
Ausnahme: Membervariablen, die als **mutable** deklariert sind.
- dürfen keine nicht-const Methoden aufrufen
- Der Compiler kann diese Eigenschaft überprüfen und ggf. eine Fehlermeldung ausgeben (falls die Fkt. es doch versucht).
→ Compiler verwenden, um (eigene) Programmierfehler aufzudecken
- Wichtig insbesondere bei der Arbeit mit konstanten Objekten (gleich ...)
- **Bem.:** (statische) Klassenmethoden können nicht const deklariert werden!

Design-Empfehlungen

const-Instanzmethoden

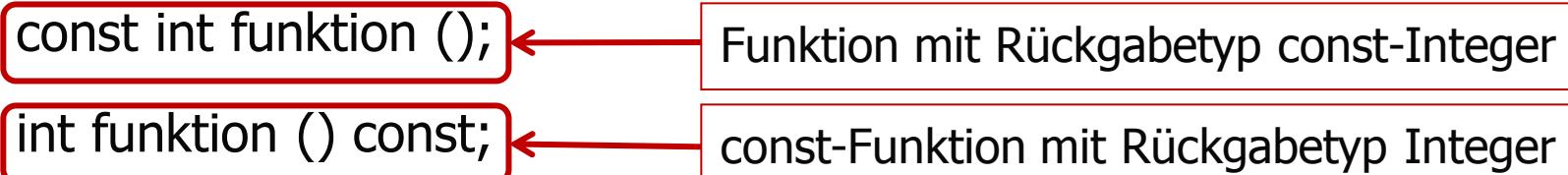
Deklaration einer konstanten Instanzmethode:

Rückgabetyp Funktionsname (PARAMETER) **const**;

Warum steht Schlüsselwort **const** hinten?

→ Unterscheidbarkeit zu Funktionen mit konstantem Rückgabetyp

Beispiel:



Bem.: Schlüsselwort **const** muß auch bei der Definition der Methode stehen

```
class konto {  
    ...  
    string getName() const;  
}
```

Deklaration

```
string konto::getName() const{  
    return name;  
}
```

Definition

Design-Empfehlungen

```
class konto {  
    private:  
        string name;  
        int kontonummer;  
        int kontostand;  
        int zaehler;  
        static int kontenzaehler;  
  
    public:  
        konto (string name) { ... }  
  
        string getName() const  
        {  
            return name;  
        }  
  
        int getNummer() const  
        {  
            return kontonummer;  
        }  
  
        // ...  
}
```

Beispiel für const-Funktionen

OK, hier werden die Attribute nicht verändert → daher Methoden **const**

Design-Empfehlungen

```
class konto {  
    private:  
        string name;  
        int kontonummer;  
        int kontostand;  
        int zaehler;  
        static int kontenzaehler;  
  
    public:  
        konto (string name) { ... }  
  
        string getName() const  
        {  
            return name;  
        }  
  
        int getNummer() const  
        {  
            kontonummer = 2;  
            return kontonummer;  
        }  
  
        // ...  
}
```

Beispiel für const-Funktionen

Compilerfehlermeldung:
assignment of data-member
'konto::kontonummer' in read-only structure

const-Instanzmethoden

Warum ist es wichtig, dass Instanzmethoden, die keine Attribute verändern, als const-Funktionen deklariert werden?

1. Der Compiler kann dadurch helfen, Programmierfehler (ungewollte Veränderung von Attributen) aufzudecken.
2. const-Deklaration bescheinigt der Funktion, dass sie keine Attribute verändert. Diese *Bescheinigung* ist aus folgendem Grund wichtig:
Bei der Arbeit mit **konstanten Objekten** garantiert der Compiler, dass diese nicht verändert werden.

Wie stellt er das sicher?

const-Instanzmethoden

Warum ist es wichtig, dass Instanzmethoden, die keine Attribute verändern, als const-Funktionen deklariert werden?

Bei der Arbeit mit **konstanten Objekten** garantiert der Compiler, dass diese nicht verändert werden.

Wie stellt er das sicher?

1. Direkte Zuweisungen an Membervariablen werden nicht zugelassen.
2. Indirekte Änderungen über Instanzmethoden werden dadurch ausgeschlossen, dass nur konstante Instanzmethoden aufgerufen werden dürfen.

const-Instanzmethoden

Warum ist es wichtig, dass Instanzmethoden, die keine Attribute verändern, als const-Funktionen deklariert werden?

Indirekte Änderungen über Instanzmethoden werden dadurch ausgeschlossen, dass nur konstante Instanzmethoden aufgerufen werden dürfen.



Konstante Instanzmethoden, die nicht als const deklariert wurden, können bei der Arbeit mit einem konstanten Objekt *nicht* verwendet werden (obwohl sie eigentlich unbedenklich sind).

→ unnötige Einschränkung beim (in C++ häufigen) Einsatz konstanter Objekte

Design-Empfehlungen

```
class konto {  
    private:  
        string name;  
        int kontonummer;  
        int kontostand;  
        int zaehler;  
        static int kontenzaehler;  
  
    public:  
        konto (string name) { ... }  
        void einzahlung (int betrag) { ... }  
        int getNummer() const  
        {  
            return kontonummer;  
        }  
        // ...  
    }  
  
int main (void) {  
    const konto k("max mustermann");  
  
    k.einzahlung(200);  
    // ...  
}
```

Beispiel für const-Funktionen

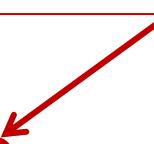
Nicht erlaubt, weil es sich um ein konstantes Objekt handelt.

Design-Empfehlungen

```
class konto {  
    private:  
        string name;  
        int kontonummer;  
        int kontostand;  
        int zaehler;  
        static int kontenzaehler;  
  
    public:  
        konto (string name) { ... }  
  
        int getNummer() const  
        {  
            return kontonummer;  
        }  
        // ...  
    }  
  
int main (void) {  
    const konto k("max mustermann");  
  
    cout << "Kontonummer: " << k.getNummer() << endl;  
    // ...  
}
```

Beispiel für Arbeit mit konstantem Objekt

OK, weil der Aufruf einer const-Funktion am konstanten Objekt sicher keine Attribute verändert..

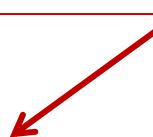


Design-Empfehlungen

```
class konto {  
    private:  
        string name;  
        int kontonummer;  
        int kontostand;  
        int zaehler;  
        static int kontenzaehler;  
  
    public:  
        konto (string name) { ... }  
  
        int getNummer()  
        {  
            return kontonummer;  
        }  
        // ...  
    }  
  
int main (void) {  
    const konto k("max mustermann");  
  
    cout << "Kontonummer: " << k.getNummer() << endl;  
    // ...  
}
```

Beispiel für Arbeit mit konstantem Objekt

Hier nicht erlaubt, weil Funktion nicht als const deklariert wurde (auch wenn sie eigentlich keine Werte verändert).



Get-/Set-Memberfunktionen

Motivation:

Die Möglichkeit einer direkten Änderung von Attributen kann ein Sicherheitsproblem für die Klasse darstellen.

- Wenn Benutzer eines Objekts also die Möglichkeit haben soll, dessen Attribute zu verändern, so sollte das Objekt diesen Zugriff kontrollieren.
- Diese Kontrolle wird möglich, wenn:
 1. Sämtliche Attribute des Objekts als **private** deklariert sind
 2. Notwendiger Zugriff nur mit Hilfe von set-Funktionen möglich ist, welche eine Plausibilitätskontrolle der Änderung vornehmen und diese ggf. auch verweigern.

Get-/Set-Memberfunktionen

Beispiel:

```
class bruch

{
public:
    int zaehler;
    int nenner;

    bruch() : zaehler(1), nenner(1) {}

    double ausrechnen() const
    {
        return (double) zaehler / nenner;
    }
};
```

Kommentar zu dieser Klasse?

Problem: Freier Zugriff auf das Attribut **nenner**. Falls Benutzer des Objekts sich mit Mathematik nicht so gut auskennt, kann er **nenner** gleich 0 setzen. Somit würde bei Ausführung der Funktion **ausrechnen()** eine unzulässige Division durch 0 ausgeführt werden..

Get-/Set-Memberfunktionen

Beispiel:

```
class bruch
{
public:
    int zaehler;
    int nenner;

    bruch() : zaehler(1), nenner(1) {}

    double ausrechnen() const
    {
        return (double) zaehler / nenner;
    }
};
```

Aufgabe: Lösen Sie das Problem u.a. durch Einführung einer entsprechenden set-Memberfunktion.
Die Werte beider Attribute sollen weiterhin vom Benutzer abrufbar sein.

Design-Empfehlungen

```
class bruch
{
    int nenner; // privat

public:
    int zaehler;

    bruch() : nenner(1), zaehler(1) {}

    int get_nenner() const { return nenner; }

    void set_nenner(int n) {
        if (n != 0)
            nenner = n;
    }

    double ausrechnen() const {
        return (double) zaehler / nenner;
    }
};
```

Klassentemplates

Ziel:

Entwurf einer Klasse, die mit unterschiedlichen Datentypen arbeitet

Beispiel: Stack (Stapelspeicher) zur Speicherung *beliebiger* Objekte

- Int-Objekte
 - String-Objekte
 - MeineKlasse-Objekte
 - ...
-
- Mit Templates können Klassen für einen beliebigen, später (bei der Benutzung) festzulegenden Datentyp geschrieben werden.
 - Für den noch unbestimmten Datentyp wird ein Platzhalter eingeführt.

Klassentemplates

Zur Erinnerung: Funktionentemplates

template<typename TypBezeichner>
Funktionsdefinition

TypBezeichner :

beliebiger Name, der in Funktionsdefinition als Datentyp verwendet wird.
Häufig: T (für Template)

Klassentemplates

template<class TypBezeichner>
Klassendefinition

```
template < typename T >
T max( T a, T b )
{ ... }
```

Hier dürfte auch
< typename T >
stehen (wie oben)

```
template < class T >
class Stack
{ ... }
```

Klassentemplates

- Klassentemplate: typenunabhängige Vorlage für Klasse

Allgemeine Form eines Klassentemplates:

```
template<class TypBezeichner>
```

Klassendefinition

TypBezeichner :

beliebiger Name, der in Klassendefinition als Datentyp verwendet wird.

Häufig: einfach T (für Template)

- Compiler erzeugt durch Typbindung echte Klasse („Templateklasse“):

Klassentemplate (Schablone):

```
template < class T >
class Stack
{ ... }
```



Neue Typangabe

Konkrete Templateklasse:

```
Stack<int> i;
Stack<float> f;
Stack<char> c;
```

Klassentemplates

- „Template-Name + Template-Argument“ liefert neue Typangabe, die anderen Typangaben völlig gleichgestellt ist
Bsp.: `Stack<int>`, `Stack<float>`, `Stack<char>`
- Typbezeichner kann in Methoden verwendet werden
- Beachte: Bei der Definition der Methode *außerhalb* der Klasse muss der Template-Kopf (`template<class ...>`) wiederholt und die volle Typangabe verwendet werden (Klassenname + Typbezeichner)
(Typbezeichner in spitzen Klammern an Klassenname anhängen)
Bsp.: `template <class T>`
`void Stack<T>::pop()`
- Instanziierung:

```
Stack<int> i;  
Stack<float> f;  
Stack<char> c;
```

Klassentemplates

Beispiel: Klasse Stack

```
template <class T>
class SimpleStack
{
    unsigned int anzahl;
    T array[MAX_SIZE] ;//Elemente

public:
    const T& top() const;
    void pop();
    void push(const T &x);
};
```

SimpleStack.h

```
int main()
{
    SimpleStack<int> einIntStack;
    cout << "oberstes Element: " << einIntStack.top() << endl;
    SimpleStack<double> einDoubleStack;
    cout << "oberstes Element: " << einDoubleStack.top() << endl;
    ...
}
```

```
template <class T>
const T& SimpleStack<T>::top() const
{
    return array[anzahl-1];
}

template <class T>
void SimpleStack<T>::pop()
{
    --anzahl;
}

template <class T>
void SimpleStack<T>::push(const T &x)
{
    array[anzahl++] = x;
```

Funktionsdefinitionen der
Templateklasse im Header!

Fortsetzung SimpleStack.h

Klassentemplates

Beispiel: Klasse Vektor mit double-Elementen

```
class Vektor
{
private:
    unsigned int size; ← GröÙe des Feldes
    double* v; ← Zeiger auf Feld

public:
    Vektor() : size(0), v(0) {} ← Konstruktor über Initialisierungsliste
    Vektor(unsigned int _size); ← Konstruktor mit GröÙenvorgabe
    Vektor(const Vektor& _vek); ← Kopierkonstruktor
    ~Vektor() { if (v) delete[] v; } ← Destruktor

    void resize(unsigned int _size);
    unsigned int getSize() { return size; }
    const double& at(unsigned int _i) const; ← Elementrückgabe für
    double& at(unsigned int _i); ← Elementrückgabe für
};
```

nicht-konstantes Objekt

konstantes Objekt

Aus: <http://www.cpp-entwicklung.de>

484

Klassentemplates

Beispiel: Klasse Vektor mit double-Elementen

```
Vektor::Vektor(unsigned int _size) : size(_size) {  
    v = new double[_size];  
}
```

Konstruktor mit Größenvorgabe

Feld dynamisch anlegen

```
Vektor::Vektor(const Vektor& _vek) : size(_vek.size) {  
    v = new double[_size];  
    for(unsigned int i=0; i<_size; i++)  
        v[i] = _vek.v[i];  
}
```

Kopierkonstruktor

```
void Vektor::resize(unsigned int _size) {  
    if (v) delete[] v;  
    size = _size;  
    v = new double[_size];  
}
```

Rückgabe: Referenz
auf Konstante

Funktion verändert
Objekt nicht

```
const double& Vektor::at(unsigned int _i) const {  
    if (_i<size && v!=0) return v[_i];  
    else return (-1);  
}
```

Elementrückgabe für
konstantes Objekt

```
double& Vektor::at(unsigned int _i) {  
    if (_i<size && v!=0) return v[_i];  
    else return (-1);  
}
```

Elementrückgabe für
nicht-konstantes Objekt

Aus: <http://www.cpp-entwicklung.de>

185

Beispiel: Klasse Vektor mit double-Elementen

Hauptfunktion:

```
int main(int argc, char* argv[]) {
    unsigned int i=0;
    Vektor v(5);

    for(i=0; i<5; i++)
        v.at(i) = static_cast<double>(i+3);

    for(i=0; i<5; i++)
        cout << v.at(i) << " ";
    cout << endl;

    system("PAUSE");
    return 0;
}
```



Vektor von 5 double-Elementen



Initialisierung der Feldelemente;
Beachte: *Schreibender* Zugriff auf
Elemente des Vektors; erfordert
nicht-konstante Variante von `at`



Ausgabe

Ausgabe?

3 4 5 6 7

Drücken Sie eine beliebige Taste . . .

Beispiel: Klasse Vektor als Template

```
template <class T>           ← Template und Typbezeichner einführen
class Vektor
{
private:
    unsigned int size;
    T* v;           ← Datentyp durch Typbezeichner ersetzen

public:
    Vektor() : size(0), v(0) {}
    Vektor(unsigned int _size);
    Vektor(const Vektor& _vek);
    ~Vektor() { if (v) delete[] v; }

    void resize(unsigned int _size);
    unsigned int getSize() { return size; }
    const T& at(unsigned int _i) const;
    T& at(unsigned int _i);
};
```

Klassentemplates

Beispiel: Klasse Vektor als Template

```
template <class T>
Vektor<T>::Vektor(unsigned int _size) : size(_size) {
    v = new T[_size];
}

template <class T>
Vektor<T>::Vektor(const Vektor<T>& _vek) : size(_vek.size) {
    v = new T[_size];
    for(unsigned int i=0; i<size; i++)
        v[i] = _vek.v[i];
}

template <class T>
void Vektor<T>::resize(unsigned int _size) {
    if (v)
        delete[] v;
    size = _size;
    v = new T[_size];
}

template <class T>
const T& Vektor<T>::at(unsigned int _i) const { ... }

template <class T>
T& Vektor<T>::at(unsigned int _i) { ... }
```

Templatekopf erforderlich;
Schlüsselwort **class / typename**

Typangabe über <> mit
Templateparameter T

Datentyp durch Typbezeichner ersetzen

Beispiel: Klasse Vektor als Template

Hauptfunktion:

```
int main(int argc, char* argv[]) {
    unsigned int i=0;
    Vektor<int> v(5);           ← Vektor von 5 int-Elementen
    for(i=0; i<5; i++)
        v.at(i) = i+3;          ← Initialisierung der Feldelemente
    for(i=0; i<5; i++)
        cout << v.at(i) << " ";
    cout << endl;
}
```

Ausgabe?

3 4 5 6 7

Drücken Sie eine beliebige Taste . . .

Einschub: Klasse Bruch erweitert

```
class bruch
{
    int nenner; // privat
public:
    int zaehler;
    bruch() : zaehler(1), nenner(1) {}
    int get_nenner() const { return nenner; }
    void set_nenner(int n) {
        if (n != 0)
            nenner = n;
    }
    double ausrechnen() const {
        return (double) zaehler / nenner;
    }

    void setze( int z, int n )      {
        zaehler = z;
        set_nenner(n);
    }
};
```

Neue Methode
zum Setzen der Attribute



Beispiel: Klasse Vektor als Template

Hauptfunktion: Vektor mit selbstdefinierten Datentypen

```
int main(int argc, char* argv[]) {
    // Vektor von Bruch
    Vektor<bruch> b(7);           ← Vektor von 7 bruch-Elementen

    for(i=0; i<7; i++)
        b.at(i).setze(i, i+1);   ← Initialisierung der Feldelemente über Methode setze

    for(i=0; i<7; i++)
        cout << b.at(i).ausrechnen() << " ";
    cout << endl;                ← Ausgabe über Methode ausrechnen
```

Ausgabe?

0 0.5 0.666667 0.75 0.8 0.833333 0.857143

Drücken Sie eine beliebige Taste . . .

Klassentemplates

Beispiel: Klasse Stack erweitert

Klassendefinition

```
template <class T>
class SimpleStack {
private:
    static const unsigned int MAX_SIZE = 20; // maximale Größe
    unsigned int anzahl;                  // Anzahl Elemente
    T array[MAX_SIZE];                  // Behälter für Elemente

public:
    SimpleStack() : anzahl(0) {}

    bool empty() const { return anzahl == 0; }           Ist der Stack leer?
    bool full() const { return anzahl == MAX_SIZE; }     Ist der Stack voll?
    unsigned int size() const { return anzahl; }          Größe des Stacks
    void clear() { anzahl = 0; }                          Stack leeren

    const T& top() const;                                Oberstes Element liefern
    void pop();                                         Oberstes Element vom Stack entfernen

    // Vorbedingung für top und pop: Stack ist nicht leer
    void push(const T &x);                            x auf den Stack legen
    // Vorbedingung für push: Stack ist nicht voll

};
```

aus: Breymann, C++
492

Klassentemplates

Beispiel: Klasse

```
template <class T>
const T& SimpleStack<T>::top() const {
```

```
assert(!empty());
return array[anzahl-1];
}
```

```
template <class T>
void SimpleStack<T>::pop() {
    assert(!empty());
    --anzahl;
}
```

```
template <class T>
void SimpleStack<T>::push(const T &x) {
    assert(!full());
    array[anzahl++] = x;
}
```

Reiner Lesezugriff (Schreiben ist nur über Funktion push() möglich.)

Methodenimplementation

Methode verändert Attribute nicht

Klassenname erweitert um TypBezeichner (s.o.)

Verifikation der logischen Annahme, d.h. Fehlermeldung, falls Annahme nicht zutrifft. Fehlermeldung enthält Annahme, Datei und assert-Zeilenummer.

Funktion verändert das übergebene Argument nicht (speichert lediglich ins aktuelle Objekt).

Klassentemplates

```
int main() {  
    SimpleStack<int> einIntStack;  
  
    // Stack füllen  
    int i = 100;  
    while(!einIntStack.full())  
        einIntStack.push(i++);  
  
    cout << "Anzahl : " << einIntStack.size() << endl;  
  
    // oberstes Element anzeigen  
    cout << "oberstes Element: " << einIntStack.top() << endl;  
  
    cout << "alle Elemente entnehmen und anzeigen: " << endl;  
    while(!einIntStack.empty()) {  
        i = einIntStack.top();  
        einIntStack.pop();  
        cout << i << '\t';  
    }  
  
    cout << endl;  
    system("PAUSE");  
    return 0;  
}
```

Anwendung mit Integer-Elementen

Ein Stack-Objekt für Integer-Objekte wird erzeugt.

Stack wird mit Elementen gefüllt, bis er voll ist.

Solange Stack nicht leer ist, oberstes Element anzeigen und entfernen.

Klassentemplates

```
int main() {  
    SimpleStack<double> einDoubleStack;  
  
    // Stack mit (beliebigen) Werten füllen  
    double d = 1.00234;  
  
    while(!einDoubleStack.full()) {  
        d = 1.1 * d;  
        einDoubleStack.push(d);  
        cout << einDoubleStack.top() << '\t';  
    }  
  
    // einDoubleStack.push(1099.986); // Fehler, da Stack voll  
    cout << "\n4 Elemente des Double-Stacks entnehmen:" << endl;  
    for (int i = 0; i < 4; ++i) {  
        cout << einDoubleStack.top() << '\t';  
        einDoubleStack.pop();  
    }  
  
    cout << endl;  
    cout << "Restliche Anzahl : "  
        << einDoubleStack.size() << endl;  
    system("PAUSE");  
    return 0;  
}
```

Anwendung mit Double-Elementen

Ein Stack für Double-Objekte wird erzeugt.

Stack wird mit Elementen gefüllt, bis er voll ist. Eingefügte Elemente werden angezeigt.

Entnahme und Ausgabe von 4 Elementen

SimpleStack funktioniert wegen << nicht mit Typ bruch
→ Operatorüberladung (Operator << für bruch definieren)

Operatorüberladung: Einführung

C++ unterstützt zahlreiche Operatoren für die fundamentalen Datentypen

Beispiel: Erlaubte Operationen auf Integer-Variablen sind

- Arithmetische Operationen, wie `+`, `-`, `*`, `/`
- Vergleichsoperationen, wie `<`, `>`, `>=`, `<=`
- Bitmanipulationen, wie `&` oder `|`

Problem: Die meisten "Konzepte", für die die Verwendung von Operatoren nützlich wäre, sind aber keine fundamentalen C++-Datentypen.

Beispiel:

Komplexe Zahlen, Matrizen, Zeichenketten, ...

Operatorüberladung: Einführung

Problem: Die meisten Konzepte, für die die Verwendung von Operatoren nützlich wäre, sind aber keine fundamentalen C++-Datentypen.

- Diese Konzepte müssen durch geeignete Klassen dargestellt werden
- Die Operatoren müssen für diese Klassen definiert/angepasst werden

Was bedeutet das?

Definition eines Operators für eine Klasse bedeutet anzugeben, was der Operator bei Anwendung auf ein (zwei) Objekt(e) dieser Klasse genau tut.

Beispiel: Komplexe Zahlen

Addition zweier komplexer Zahlen → Addition von Real- & Imaginärteilen

Unäre / binäre Operatoren

Man unterscheidet zwischen

- unären (einstelligen)
- binären (zweistelligen)

Operatoren.

Unärer Operator

- ein Operand
- Beispiele: $x--$, $++a$, $-x$

Minuszeichen



Binärer Operator

- zwei Operanden
- Beispiele: $a + b$, $x < y$, $true \&& false$

Operatorüberladung für Klassenobjekte

Möglichkeiten zur Operatorüberladung

Wo und wie wird definiert, was Operator @ bezogen auf Klasse X tun soll?

Bemerkung: Irgendein Operator,
irgendeine Klasse.

Möglichkeit 1: Operatorüberladung als Instanzfunktion

- Objekt ruft Instanzfunktion auf und arbeitet mit this-Zeiger
- wichtig: Der erste Operand kann nur ein Objekt der Klasse sein!

```
class X
{...
 public:
    <return type> operator@ (...); // z.B. X operator@(X);
};                                // z.B. void operator@();
```

I.a. void / Objekt oder Referenz auf Klasse X

Möglichkeit 2: Operatorüberladung als globale Funktion

- Operanden müssen Argumente der globalen Funktion sein

```
class X
{...};
<return type> operator@ (...);      // z.B. X operator@(X,X);
                                         // z.B. void operator@(X);
```

I.a. void / Objekt oder Referenz auf Klasse X

Operatorüberladung für Klassenobjekte

Operatorüberladung: a) binäre Operatoren, Instanzfunktion

Wo und wie wird definiert, was Operator @ bezogen auf Klasse x tun soll?

Bemerkung: Irgendein Operator,
irgendeine Klasse.

Binärer Operator:

- Annahme: aa und bb seien Objekte der Klasse x
- Ziel: Definition der Anweisung $\text{aa} \text{ @ } \text{bb}$

Möglichkeit 1: Operatorüberladung als Instanzfunktion

$\text{aa} \text{ @ } \text{bb}$ wird vom Compiler als $\text{aa}.\text{operator@}(\text{bb})$ interpretiert

→ Instanzfunktion $\text{operator@}()$ mit einem Parameter und geeigneten Rückgabedatentyp (z.B. x falls Ergebnis von $\text{aa} \text{ @ } \text{bb}$ vom Typ x ist) so definieren, dass sie d. gewünschte Eigenschaft des Operators hat.

$\text{x operator@}(x); \quad // \text{ Prototyp der Operatorfunktion}$

Operatorüberladung für Klassenobjekte

Operatorüberladung: a) binäre Operatoren, Instanzfunktion

Beispiel: zweistelliger Operator + , Klasse „complex“

```
class complex {  
    double re, im;  
  
public:  
    complex () : re(0.0), im(0.0) {};  
    complex (double r, double i) : re(r), im(i) {};  
  
    complex operator+ (complex c);  
    double getReal() {return re;}  
    double getIm() {return im;}  
    void setReal(double r) {re = r;}  
    void setIm(double i) {im = i;}  
};
```

Annahme: **a** und **b** seien vom Typ complex
→ Interpretation von **a + b** als **a.operator+(b)**
→ Um die gewünschte Bedeutung zu bekommen,
muss die Funktion noch geeignet definiert werden.

Operatorüberladung für Klassenobjekte

Beispiel: zweistelliger Operator + , Klasse „complex“

```
// Klassendefinition siehe oben
complex complex::operator+(complex c) {
    complex tmp;
    tmp.re = re + c.re;
    tmp.im = im + c.im;
    return tmp;
}
```

Ausgabe?

3.3 3.3

```
int main(int argc, char *argv[])
{
    complex a(1.2, 3.0), b(2.1, 0.3);
    complex c;

    c = a + b;
    cout << c.getReal()
    << " " << c.getIm()
    << endl;
    return EXIT_SUCCESS;
}
```

Bedeutung von **a + b**: **a.operator+(b)**

- Operatorfunktion wird an **a** aufgerufen
- **b** wird an Funktion übergeben
- Rückgabewert wird **c** zugewiesen

Operatorüberladung: a) binäre Operatoren, Instanzfunktion

Bemerkungen:

- Operatoren, die typischerweise als Instanzfunktion überladen werden:
`=, +=, -=, *=, /=`
 - Bem.: `+=, -=, *=, /=` sind *eigenständige* Operatoren und werden *nicht* automatisch gebildet durch Anwendung des `+` und `=` Operators etc., falls diese existieren!
 - Bei diesen Operatoren wird in der Regel Zugriff (lesend und schreibend) auf geschützte Membervariablen benötigt.
 - Schreibenden Zugriff sollten nur Instanzfunktionen bekommen

Operatorüberladung für Klassenobjekte

Operatorüberladung: a) binäre Operatoren, globale Funktion

Wie wird definiert, was Operator @ bezogen auf Klasse x tun soll?

Binärer Operator:

- Annahme: aa und bb seien Objekte der Klasse x
- Ziel: Definition der Anweisung aa @ bb

Möglichkeit 2: Operatorüberladung als globale Funktion

aa @ bb wird vom Compiler als operator@ (aa,bb) interpretiert

→ **Nicht-Memberfunktion** operator@ () mit zwei Parametern und geeignetem Rückgabedatentyp so definieren, dass die gewünschte Eigenschaft des Operators erreicht wird.

D.h. sie gehört nicht zu einer Klasse

x operator@ (x,x) ; // Prototyp der Operatorfunktion

Operatorüberladung für Klassenobjekte

Operatorüberladung: a) binäre Operatoren, globale Funktion

Nicht-Memberfunktion `operator@()` mit zwei Parametern und geeignetem Rückgabetyp so definieren, dass sie die gewünschte Eigenschaft des Operators am Objekt realisiert.



Bemerkung: Dies ist i.a. nur möglich, wenn das Objekt Zugriff auf die zu modifizierenden Eigenschaften ermöglicht.

Warum?

Weil es sich nicht um eine Memberfunktion handelt. Falls die Attribute nicht public sind, müssen geeignete Zugriffsfunktionen somit bereit gestellt werden.

Alternative: Klasse deklariert globale Funktion als „friend“

Operatorüberladung für Klassenobjekte

Operatorüberladung: a) binäre Operatoren, globale Funktion

Beispiel: zweistelliger Operator + , Klasse „complex“

```
class complex {  
    double re, im;  
  
public:  
    complex () : re(0.0), im(0.0) {};  
    complex (double r, double i) : re(r), im(i) {};  
  
    double getReal() {return re;}  
    double getIm() {return im;}  
    void setReal(double r) {re = r;}  
    void setIm(double i) {im = i;}  
};
```

Operatorfunktion `operator@ (x,x)` ist
kein Mitglied der Klasse

Operatorüberladung für Klassenobjekte

Beispiel: zweistelliger Operator + , Klasse „complex“

```
// Klassendefinition siehe oben

complex operator+ (complex a, complex b) {
    complex tmp;
    tmp.setReal(a.getReal() + b.getReal());
    tmp.setIm(a.getIm() + b.getIm());
    return tmp;
}

int main(int argc, char *argv[])
{
    complex x(1.2, 3.0), y(2.1, 0.3);
    complex c;

    c = x + y;
    cout << c.getReal()
    << " " << c.getIm()
    << endl;

    return EXIT_SUCCESS;
}
```

Kein Zugriff auf
private Elemente
→ Zugriffsmethoden!

Interpretation von **x+y** als **operator+(x,y)**
→ Operatorfunktion wird aufgerufen
→ **x** und **y** werden an Funktion übergeben
→ Rückgabewert wird **c** zugewiesen

Operatorüberladung für Klassenobjekte

Beispiel: zweistelliger Operator + , Klasse „complex“

```
// Klassendefinition siehe oben

complex operator+ (complex a, complex b) {
    complex tmp;
    tmp.setReal(a.getReal() + b.getReal());
    tmp.setIm(a.getIm() + b.getIm());
    return tmp;
}

int main(int argc, char *argv[])
{
    complex x(1.2, 3.0), y(2.1, 0.3);
    complex c;

    c = x + y;
    cout << c.getReal()
    << " " << c.getIm()
    << endl;
    return EXIT_SUCCESS;
}
```

Ausgabe?

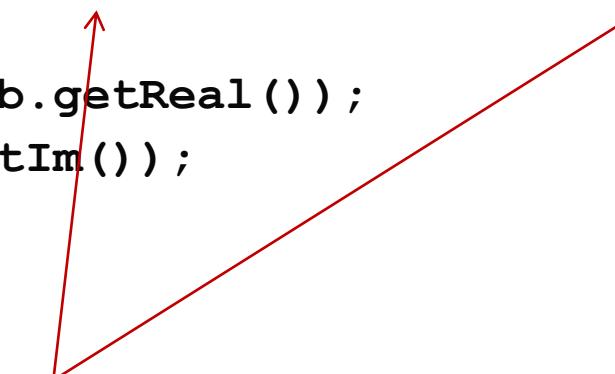
3.3 3.3

Operatorüberladung für Klassenobjekte

Beispiel: zweistelliger Operator + , Klasse „complex“

```
// Klassendefinition siehe oben

complex operator+ (const complex &a, const complex &b) {
    complex tmp;
    tmp.setReal(a.getReal() + b.getReal());
    tmp.setIm(a.getIm() + b.getIm());
    return tmp;
}
```



Bemerkung: Alternativ kann die Funktion auch mit (konstanten) Referenzen als Übergabe-Argumente programmiert werden.

Im Falle *konstanter* Referenzen müssen hierbei die get-Methoden konstant sein:

```
double getReal() const { return re; }
double getIm() const { return im; }
```

Operatorüberladung für Klassenobjekte

Beispiel: zweistelliger Operator + , Klasse „complex“

Deklaration als **friend**: Zugriff auf private Elemente

```
class complex
{
    ...
public:
    ...
friend complex operator+(const complex& a, const complex& b );
};
```

```
complex operator+ (const complex &a, const complex &b) {
    complex tmp;
    tmp.re = a.re + b.re;
    tmp.im = a.im + b.im;
    return tmp;
}
```

Operatorüberladung: a) binäre Operatoren, globale Funktion

Bemerkungen:

- Operatoren, die typischerweise als globale Funktion überladen werden:
arithmetische (+, -, *, /) und relationale Operatoren (<, >, <=, >=)
- Bei diesen Operatoren soll in der Regel folgendes erfüllt sein:
 - Ein Operand ist Objekt der Klasse, für die Operator überladen wird
 - Anderer Operand ist ebenfalls Objekt dieser Klasse **oder beliebig**
 - **Reihenfolge** beider Operanden sollte **beliebig** sein

Beispiel: Klasse Vektor, Operator +

Annahme: Objekte v und w der Klasse Vektor seien gegeben

Folgende Anwendungen des Operators sollen möglich sein:

- $v + w$ oder $w + v$ (Addition zweier Vektoren)
- $v + 2$ oder $5 + v$ (Addition eines Vektors v mit einem Skalar)

Operatorüberladung: a) binäre Operatoren, Kommutativität

2. Beispiel: zweistelliger Operator + , Klasse „Vektor“

Annahme: Objekte v und w der Klasse Vektor seien gegeben

Folgende Anwendungen des Operators sollen möglich sein:

- $v + w$ oder $w + v$ (Addition zweier Vektoren)
- $v + 5$ oder $5 + v$ (Addition eines Vektors v mit einem Skalar)

Möglichkeit 1: Operatorüberladung als Instanzfunktion

`aa + bb` wird vom Compiler als `aa.operator+(bb)` interpretiert

Damit ließe sich die Addition zweier Vektoren erreichen

Problem: Addition mit einem Skalar, wenn Skalar erster Operand ist.

Bsp. $5 + v \rightarrow 5.operator+(v) ??$

\rightarrow funktioniert nicht, da 5 kein Objekt der Klasse Vektor ist.

Lösung: Operatorüberladung als globale Funktion

Operatorüberladung: a) binäre Operatoren, Kommutativität

Möglichkeit 2: Operatorüberladung als globale Funktion

`aa + bb` wird vom Compiler als `operator+(aa,bb)` interpretiert

Damit lässt sich die Addition eines Vektors mit einem Skalar bei beliebiger Reihenfolge realisieren.

Wie?

→ Überladung der Funktion

```
operator+(vector a, int b)
```

```
operator+(int b, vector a)
```

Operatorüberladung: b) unäre Operatoren (Präfix), Instanzfunktion

Unäre Operatoren:

- Annahme: `aa` sei ein Objekt der Klasse `x`
- Ziel: Definition von `@aa` (Präfixoperator) bzw. `aa@` (Postfixoperator)

Präfixoperator: Möglichkeit 1: Operatorüberladung als Instanzfunktion

`@aa` wird vom Compiler als `aa.operator@()` interpretiert

→ Instanzfunktion `operator@()` ohne Parameter so definieren, dass sie die gewünschte Eigenschaft des Operators am Objekt realisiert.

`void operator@(); // Prototyp der Operatorfunktion`

Operatorüberladung für Klassenobjekte

Operatorüberladung: b) unäre Operatoren (Präfix), Instanzfunktion

Beispiel: einstelliger Präfixoperator ++ als Instanzfunktion

```
class complex {  
  
    double re, im;  
  
public:  
    complex () : re(0.0), im(0.0) {};  
    complex (double r, double i) : re(r), im(i) {};  
  
    void operator++();  
    double getReal() {return re;}  
    double getIm() {return im;}  
    void setReal(double r) {re = r;}  
    void setIm(double i) {im = i;}  
};
```

Operatorüberladung für Klassenobjekte

Beispiel: einstelliger Präfixoperator ++ als Instanzfunktion

```
// Klassendefinition siehe oben
```

```
void complex::operator++() {
    re += 1;
    im += 1;
}
```

Interpretation von `++c` als `c.operator++()`

- Operatorfunktion wird an `c` aufgerufen
- Real- und Im.-Teil werden inkrementiert
- Kein Rückgabewert, `c` wird direkt verändert

```
int main(int argc, char *argv[])
{
    complex c;

    cout << c.getReal() << " " << c.getIm() << endl;
    ++c;
    cout << c.getReal() << " " << c.getIm() << endl;

    return EXIT_SUCCESS;
}
```

Operatorüberladung: b) unäre Operatoren (Präfix), globale Funktion

Unäre Operatoren:

- Annahme: `aa` sei ein Objekt der Klasse `X`
- Ziel: Definition von `@aa` (Präfixoperator) bzw. `aa@` (Postfixoperator)

Präfixoperator: Möglichkeit 2: Operatorüberladung als globale Funktion

`@aa` wird vom Compiler als `operator@(aa)` interpretiert

→ *Nicht-Memberfunktion* `operator@()` mit einem Parameter so definieren, dass sie die gewünschte Eigenschaft des Operators am Objekt realisiert.

```
void operator@(X); // Prototyp der Operatorfunktion
```

Operatorüberladung für Klassenobjekte

Operatorüberladung: b) unäre Operatoren (Präfix), globale Funktion

Beispiel: einstelliger Präfixoperator ++ als Nicht-Memberfunktion

```
class complex {  
  
    double re, im;  
  
public:  
    complex () : re(0.0), im(0.0) {};  
    complex (double r, double i) : re(r), im(i) {};  
  
    double getReal() {return re;}  
    double getIm() {return im;}  
    void setReal(double r) {re = r;}  
    void setIm(double i) {im = i;}  
};
```

Operatorüberladung für Klassenobjekte

Beispiel: einstelliger Präfixoperator ++ als Nicht-Memberfunktion

```
// Klassendefinition siehe oben
```

```
void operator++ (complex &a) {  
    a.setReal(a.getReal() + 1);  
    a.setIm(a.getIm() + 1);  
}
```

```
int main(int argc, char *argv[]){  
    complex c;  
  
    cout << c.getReal()  
    << " " << c.getIm()  
    << endl;  
    ++c;  
    cout << c.getReal()  
    << " " << c.getIm() << endl;  
    return EXIT_SUCCESS;  
}
```

Alternative:
friend (siehe vorn)

Interpretation von ++c als **operator++(c)**

- Operatorfunktion wird aufgerufen
- Objekt c wird übergeben
- Real- und Im.-Teil werden mit Hilfe der set- und get-Funktionen inkrementiert
- Kein Rückgabewert, c wird direkt verändert

Operatorüberladung: b) unäre Operatoren (Postfix)

Jetzt noch fehlend: Postfix-Operator

Problem:

Funktion `operator@()` wird schon für den Präfixoperator gebraucht.

Idee: Zusätzlichen Dummy-Funktionsparameter einführen.

Bedeutung:

- Funktion **ohne** Dummy-Parameter definiert Präfixoperator
- Funktion **mit** Dummy-Parameter definiert Postfixoperator

Operatorüberladung:b) unäre Operatoren (Postfix),Instanzfunktion

Postfixoperator: Möglichkeit 1: Operatorüberladung als Instanzfunktion

`aa@` wird vom Compiler als `aa.operator@(int)` interpretiert

→ Instanzfunktion `operator@()` mit einem Dummy-Parameter (`int`) so definieren, dass sie die gewünschte Eigenschaft des Operators am Objekt realisiert.

```
void operator@(int d); // Prototyp der Operatorfkt.
```

Operatorüberladung für Klassenobjekte

Operatorüberladung:b) unäre Operatoren (Postfix),Instanzfunktion

Beispiel: einstelliger Präfixoperator ++ als Memberfunktion

```
class complex {  
  
    double re, im;  
  
public:  
    complex () : re(0.0), im(0.0) {};  
    complex (double r, double i) : re(r), im(i) {};  
  
    complex& operator++();           // Praefix-Operator  
    complex operator++(int d);      // Postfix-Operator  
    double getReal() {return re;}  
    double getIm() {return im;}  
    void setReal(double r) {re = r;}  
    void setIm(double i) {im = i;}  
};
```

Operatorüberladung für Klassenobjekte

```
complex& complex::operator++() {           // Präfix-Operator
    re += 1;
    im += 1;
    return *this;
}

complex complex::operator++(int d) {      // Postfix-Operator
    complex old = *this; // alten Zustand merken fuer Rueckgabe
    this->re += 1;       // Objekt wird veraendert
    this->im += 1;
    return old;          // alter Zustand wird zurueck gegeben
}

int main(int argc, char *argv[]) {
    complex a, b, c;
    cout << a.getReal() << " " << a.getIm() << endl;
    cout << b.getReal() << " " << b.getIm() << endl;
    c = a++; // a wird veraendert, alter Wert wird c zugewiesen
    cout << c.getReal() << " " << c.getIm() << endl;
    c = ++b; // b wird veraendert, neuer Wert wird c zugewiesen
    cout << c.getReal() << " " << c.getIm() << endl;
    return EXIT_SUCCESS;
}
```

Operatorüberladung für Klassenobjekte

```
complex& complex::operator++(int d) { // Postfix-Operator
    complex old = *this; // alten Zustand merken fuer Rueckgabe
    this->re += 1;       // Objekt wird veraendert
    this->im += 1;
    return old;          // alter Zustand wird zurueck gegeben
}

int main(int argc, char *argv[])
{
    complex a, c;
    c = a++;
    cout << c.getReal() << " " << c.getIm() << endl;
    // ...
}
```

Frage:

Funktioniert die hier angegebene Variante (Rückgabe einer Referenz) auch?

Antwort:

Nein, weil eine Referenz auf ein complex-Objekt zurück gegeben wird, welches nach Beendigung der Funktion `operator++()` nicht mehr existiert.

Operatorüberladung für Klassenobjekte

```
complex& complex::operator++(int d) { // Postfix-Operator
    complex old = *this; // alten Zustand merken fuer Rueckgabe
    this->re += 1;       // Objekt wird veraendert
    this->im += 1;
    return old;          // alter Zustand wird zurueck gegeben
}
```

```
int main(int argc, char *argv[])
{
    complex a, c;
    c = a++;
```

Ausgabe:

0 0

1 1

2.12211e-313 5.36136e-306

```
cout << c.getReal() << " " << c.getIm() << endl;
++c;
```

```
cout << c.getReal() << " " << c.getIm() << endl;
```

```
for (int i=0; i<10000; i++)
{
```

```
    complex *x = new complex();
```

```
}
```

```
cout << c.getReal() << " " << c.getIm() << endl;
```

```
// ...
```

Speicherplatz, auf den c zeigt,
wurde hier zufälligerweise
noch nicht verändert.

Nach Änderungen im Speicher
wurde der Platz, auf den c zeigt,
nun zufällig überschrieben.

Operatorüberladung: b) unäre Operatoren (Postfix), globale Fkt.

Postfixoperator: Möglichkeit 2: Operatorüberladung als globale Funktion

`aa@` wird vom Compiler als `operator@(aa,int)` interpretiert

→ Nicht-Memberfunktion `operator@()` mit zwei Parametern (einer davon ist der Dummy-Parameter) so definieren, dass sie die gewünschte Eigenschaft des Operators am Objekt realisiert.

`void operator@(X, int); // Prototyp der Operatorfkt.`

Operatorüberladung für Klassenobjekte

Operatorüberladung: b) unäre Operatoren (Postfix), globale Fkt.

Beispiel: einstelliger Postfixoperator ++ als Nicht-Memberfunktion

```
class complex {  
  
    double re, im;  
  
public:  
    complex () : re(0.0), im(0.0) {};  
    complex (double r, double i) : re(r), im(i) {};  
  
    double getReal() {return re;}  
    double getIm() {return im;}  
    void setReal(double r) {re = r;}  
    void setIm(double i) {im = i;}  
};
```

Operatorüberladung für Klassenobjekte

```
complex operator++ (complex &a, int) {  
    complex &old = a;  
    a.setReal(a.getReal() + 1);  
    a.setIm(a.getIm() + 1);  
    return old;  
}  
  
int main(int argc, char *argv[]) {  
    complex a, b, c;  
    cout << a.getReal() << " " << a.getIm() << endl;  
    cout << b.getReal() << " " << b.getIm() << endl;  
    c = a++; // a wird verändert, alter Wert wird c zugewiesen  
    cout << c.getReal() << " " << c.getIm() << endl;  
    c = ++b; // b wird verändert, neuer Wert wird c zugewiesen  
    cout << c.getReal() << " " << c.getIm() << endl;  
    return EXIT_SUCCESS;  
}
```

Ausgabe:

0	0
0	0
1	1
1	1

korrekt wäre:

0	0
---	---

Korrekt?

→ Nein!
→ Wo ist Fehler?
old ist nur *Referenz* auf das übergebene Objekt a, welches verändert wird. Damit ist der alte Zustand nicht gespeichert worden.

Operatorüberladung für Klassenobjekte

kein &

```
complex operator++ (complex &a, int) {  
    complex old = a;  
    a.setReal(a.getReal() + 1);  
    a.setIm(a.getIm() + 1);  
    return old;  
}
```

```
int main(int argc, char *argv[]) {  
    complex a, b, c;  
    cout << a.getReal() << " " << a.getIm() << endl;  
    cout << b.getReal() << " " << b.getIm() << endl;  
    c = a++; // a wird verändert, alter Wert wird c zugewiesen  
    cout << c.getReal() << " " << c.getIm() << endl;  
    c = ++b; // b wird verändert, neuer Wert wird c zugewiesen  
    cout << c.getReal() << " " << c.getIm() << endl;  
    return EXIT_SUCCESS;  
}
```

Korrekte Variante

Ausgabe:

0	0
0	0
0	0
1	1

Möglichkeiten zur Operatorüberladung: Zusammenfassung

Wo und wie wird definiert, was Operator @ bezogen auf Klasse x tun soll?

Bemerkung: Irgendein Operator,
irgendeine Klasse.

Möglichkeit 1: Operatorüberladung als Instanzfunktion

- Operator wird als **Instanzfunktion einer Klasse (Methode)** definiert
- Vorteil: direkter Zugriff auf private / protected Membervariablen
- Nachteil: erster Operand muß Klassenobjekt sein
- Bsp.: Zuweisungen $=, +=, -=, *=, /=$; Operatoren, die Attribute ändern

Möglichkeit 2: Operatorüberladung als globale Funktion

- Operator wird als **globale Funktion** definiert
- Vorteil: erster Operand kann beliebiges Objekt sein
- Nachteil: Zugriff auf private Attribute nur über öffentliche Methoden
 - Oder: als „friend“ deklarieren → direkter Zugriff auf private Attribute
 - Aber: Falls Klassenvariablen verändert werden sollen → Instanzfunktion!
- Bsp.: arithmetische und relationale Operatoren (Kommutativität)

Operatorüberladung

Liste der wichtigsten Operatoren, die überladen werden können:

+	-	*	/	%	^	&		~	!
=	<	>	+ =	- =	* =	/ =	<<	>>	==
!=	<=	>=	&&		++	--	[]		

Bemerkungen:

- Bedeutung des Operators sollte unbedingt erhalten bleiben (intuitiv sein)
 - z.B. sollte der +-Operator eine wie auch immer geartete Addition vornehmen
 - Bei Bedeutungsverschiebungen: Alternative: Methode „add“
- Es können keine neuen Operatoren definiert werden

Operatorüberladung: Weitere Regeln

- Mindestens ein Operand muß Klasse / Referenz auf Klasse sein
 - D.h. Funktionsweise der Operatoren für elementare Datentypen unveränderbar
- Operatorauslegung (Anzahl Operanden, Priorität etc.) unveränderbar
- Nicht überladen werden können:
:: , . , .* , ?: (bedingter Ausdruck), sizeof , typeid, C++-Cast-Operatoren, #, ##, defined
- Nur als Instanzfunktion überladbar: = () [] ->
- Für Operator *verkettung* : Rückgabewert: Referenz auf linkes Argument

Operatorüberladung für Klassenobjekte

Operatorüberladung bei Klassentemplates: Besonderheit bei friends

- Besonderheit bei Freund-Deklarationen in Klassentemplates; Bsp (Teil 1):

```
class bruch {  
    int nenner;  
  
public:  
    int zaehler;  
    bruch(int n = 1, int z = 1) : nenner(n), zaehler(z) {}  
    int get_nenner() const { return nenner; }  
    void set_nenner(int n) { if (n != 0) nenner = n; }  
    friend bruch operator*(const bruch &a, const bruch &b);  
};  
  
bruch operator*(const bruch &a, const bruch &b)  
{  
    return bruch( a.zaehler * b.zaehler, a.nenner * b.nenner );  
}
```

(noch) kein Template

Operator* als globale friend-Funktion

Operatorüberladung für Klassenobjekte

Operatorüberladung bei Klassentemplates: Besonderheit bei friends

- Besonderheit bei Freund-Deklarationen in Klassentemplates; Bsp (Teil 2):

```
template <class T> class bruch {  
    T nenner;  
  
public:  
    T zaehler;  
    bruch(T n = 1, T z = 1) : nenner(n), zaehler(z) {}  
    T get_nenner() const { return nenner; }  
    void set_nenner(T n) { if (n != 0) nenner = n; }  
    friend bruch<T> operator*(const bruch<T> &a, const bruch<T> &b);  
};  
  
template <class T>  
bruch<T> operator*(const bruch<T> &a, const bruch<T> &b)  
{  
    return bruch<T>( a.zaehler * b.zaehler, a.nenner * b.nenner );  
}
```

jetzt Template

auch Template

Operatorüberladung für Klassenobjekte

Operatorüberladung bei Klassentemplates: Besonderheit bei friends

- Besonderheit bei Freund-Deklarationen in Klassentemplates; Bsp (Teil 2):

```
int main( int argc, char* argv[] )  
{  
    bruch<int> b1(1, 2), b2(3, 4);  
    bruch<int> b = b1 * b2;  
  
    return 0;  
}
```

Warnung:

warning: friend declaration `bruch<T> operator*(const bruch<T>&, const
bruch<T>&)' declares a **non-template function**

warning: (if this is not what you intended, make sure the function template
has already been declared and add <> after the function name here)

Linker-Fehler:

undefined reference to `operator*(bruch<int> const&, bruch<int> const&)'

Operatorüberladung für Klassenobjekte

Operatorüberladung bei Klassentemplates: Besonderheit bei friends

- Besonderheit bei Freund-Deklarationen in Klassentemplates; Bsp (Teil 2):

```
template <class T> class bruch
{
    T nenner;
public:
    T zaehler;
    bruch(T n = 1, T z = 1) : nenner(n), zaehler(z) {}
    T get_nenner() const { return nenner; }
    void set_nenner(T n) { if (n != 0) nenner = n; }
    friend bruch<T> operator*(const bruch<T> &a, const bruch<T> &b);
};

template <class T>
bruch<T> operator*(const bruch<T> &a, const bruch<T> &b)
{
    return bruch<T>( a.zaehler * b.zaehler, a.nenner * b.nenner );
}
```

Compiler hält **operator*** hier für non-Template-Funktion (siehe Warnung)

Hier wird eine Template-Funktion **operator*** definiert

Operatorüberladung für Klassenobjekte

Operatorüberladung bei Klassentemplates: Besonderheit bei friends

- Besonderheit bei Freund-Deklarationen in Klassentemplates; Bsp (Teil 2):

```
int main( int argc, char* argv[] )  
{  
    bruch<int> b1(1, 2), b2(3, 4);  
    bruch<int> b = b1 * b2;  
  
    return 0;  
}
```

Hier wird die **non-Template-Funktion operator*** aufgerufen (**friend**-Deklaration), die aber nicht definiert ist (nur die Template-Funktion)
→Linker-Fehler

Warnung:

warning: friend declaration `bruch<T> operator*(const bruch<T>&, const bruch<T>&)' declares a **non-template function**

warning: (if this is not what you intended, make sure the function template has already been declared and add <> after the function name here)

Linker-Fehler:

undefined reference to `operator*(bruch<int> const&, bruch<int> const&)'

Operatorüberladung für Klassenobjekte

Operatorüberladung bei Klassentemplates: Besonderheit bei friends

Deklarationen vor der Klassendefinition!

- Besonderheit bei Freund-Deklarationen in Klassentemplates; Lösung 1:

```
template <class T> class bruch; Deklaration des Klassentemplates (für nächste Zeile)
template<class T> bruch<T> operator*(const bruch<T> &a, const bruch<T> &b);

template <class T> class bruch
{
    T nenner;
public:
    T zaehler;
    bruch(T n = 1, T z = 1) : nenner(n), zaehler(z) {}
    T get_nenner() const { return nenner; }
    void set_nenner(T n) { if (n != 0) nenner = n; }
    friend bruch<T> operator* <> (const bruch<T> &a, const bruch<T> &b);
};

Zur Verdeutlichung der Template-Funktion (siehe Warnung)
template <class T>
bruch<T> operator*(const bruch<T> &a, const bruch<T> &b)
{
    return bruch<T>( a.zaehler * b.zaehler, a.nenner * b.nenner );
}
```

Operatorüberladung für Klassenobjekte

Operatorüberladung bei Klassentemplates: Besonderheit bei friends

Definitionen *in* der Klassendefinition!

- Besonderheit bei Freund-Deklarationen in Klassentemplates; Lösung 2:

```
template <class T>
class bruch
{
    T nenner;
public:
    T zaehler;
    bruch(T n = 1, T z = 1) : nenner(n), zaehler(z) {}
    T get_nenner() const { return nenner; }
    void set_nenner(T n) { if (n != 0) nenner = n; }

    friend bruch<T> operator* (const bruch<T> &a, const bruch<T> &b)
    {
        return bruch<T>( a.zaehler * b.zaehler, a.nenner * b.nenner );
    }
};
```

Definition von **operator*** innerhalb der Klassendefinition
→ Keine Konfusion mehr bzgl. Template / non-Template

Siehe auch <http://www.parashift.com/c++-faq/template-friends.html>

Operatorüberladung bei Klassentemplates: Beispiel

Klasse Vektor (siehe vorn): Operatorüberladung als **Methode**

```
template <class T>
class Vektor
{
    unsigned int size;
    T* v;

public:
    Vektor() : size(0), v(0) {}
    Vektor(unsigned int _size);
    Vektor(const Vektor& _vek);
    ~Vektor() { if (v) delete[] v; }
    ...
}
```

Operatorüberladung als Methode: wie gehabt;
Typangabe mit Typbezeichner T: **Vektor<T>**

```
Vektor<T>& operator+=( const Vektor<T>& _vek );
Vektor<T>& operator-=( const Vektor<T>& _vek );
Vektor<T>& operator=( const Vektor<T>& _vek );
bool operator==( const Vektor<T>& _vek );
Vektor<T>& operator++();           // Präfix-Operator
Vektor<T> operator++( int t );   // Postfix-Operator
```

Methoden

}

Operatorüberladung bei Klassentemplates: Beispiel

```
template<class T> class Vektor; // Deklaration der Template-Klasse  
                                (sonst wäre Vektor<T> in nächster Zeile unbekannt)  
  
template<typename T> Vektor<T> operator+( const Vektor<T>& _vek1,  
                                         const Vektor<T>& _vek2);  
  
template <class T>  
class Vektor  
{  
    unsigned int size;  
    T* v;  
  
public:  
    Vektor() : size(0), v(0) {}  
    Vektor(unsigned int _size);  
    Vektor(const Vektor& _vek);  
    ~Vektor() { if (v) delete[] v; }  
    ...  
    Vektor<T>& operator+=( const Vektor<T>& _vek );  
    ...  
};  
friend Vektor<T> operator+(<>) const Vektor<T>& _vek1, const Vektor<T>& _vek2);  
                                Bezug auf Template-Fkt.
```

Deklaration der globalen Funktion als Template-Fkt.

wie vorher

Operatorüberladung bei Klassentemplates: Beispiel

```
template<class T> class Vektor;
template<typename T> Vektor<T> operator+( const Vektor<T>& _vek1,
                                              const Vektor<T>& _vek2 );
template<typename T> Vektor<T> operator-( const Vektor<T>& _vek1,
                                              const Vektor<T>& _vek2 );
template<typename T> std::ostream& operator<<( std::ostream& os,
                                                 const Vektor<T>& _vek );
```

Deklarationen

```
template <class T>
class Vektor {
    unsigned int size;
    T* v;

public:
```

```
    Vektor() : size(0), v(0) {}
    Vektor(unsigned int _size);
    Vektor(const Vektor& _vek);
    ~Vektor() { if (v) delete[] v; }
```

```
...
    Vektor<T>& operator+=( const Vektor<T>& _vek );
    Vektor<T>& operator==( const Vektor<T>& _vek );
    Vektor<T>& operator=( const Vektor<T>& _vek );
    bool operator==( const Vektor<T>& _vek );
    Vektor<T>& operator++();           // Präfix-Operator
    Vektor<T> operator++( int t );   // Postfix-Operator
```

Vollständige Klasse
(Teil 1)

// ... Fortsetzung

Methoden

Aus: <http://www.cpp-entwicklung.de>

Operatorüberladung bei Klassentemplates: Beispiel

```
template<class T> class Vektor;
template<typename T> Vektor<T> operator+( const Vektor<T>& _vek1,
                                              const Vektor<T>& _vek2 );
template<typename T> Vektor<T> operator-( const Vektor<T>& _vek1,
                                              const Vektor<T>& _vek2 );
template<typename T> std::ostream& operator<<( std::ostream& os,
                                                 const Vektor<T>& _vek );

template <class T>
class Vektor {
    unsigned int size;
    T* v;

public:
    Vektor() : size(0), v(0) {}
    Vektor(unsigned int _size);
    Vektor(const Vektor& _vek);
    ~Vektor() { if (v) delete[] v; }
    ...
    Vektor<T>& operator+=( const Vektor<T>& _vek );
};

friend Vektor<T> operator+ <>( const Vektor<T>& _vek1, const Vektor<T>& _vek2 );
friend Vektor<T> operator- <>( const Vektor<T>& _vek1, const Vektor<T>& _vek2 );
friend std::ostream& operator<< <>( std::ostream& os, const Vektor<T>& _vek );
};
```

Deklarationen

Vollständige Klasse
(Teil 2)

Methoden (hier nur `operator+` angegeben)

Globale Funktionen (`friend`-Deklarationen)

Aus: <http://www.cpp-entwicklung.de>

Operatorüberladung bei Klassentemplates: Beispiel

Operatorüberladung: Beispiel (Klasse Vektor)

```
template <typename T>
Vektor<T>& Vektor<T>::operator+=( const Vektor<T>& _vek )
{
    assert( size == _vek.size );
    for(unsigned int i=0; i<size; i++)
        v[i] += _vek.v[i];
    return *this;
}
```

← += als Methode

```
template <typename T>
Vektor<T>& Vektor<T>::operator-=( const Vektor<T>& _vek )
{
    assert( size == _vek.size );
    for(unsigned int i=0; i<size; i++)
        v[i] -= _vek.v[i];
    return *this;
}
```

← -= als Methode

Operatorüberladung bei Klassentemplates: Beispiel

Beispiel: Klasse Vektor als Template

```
template <typename T>
Vektor<T>& Vektor<T>::operator=( const Vektor<T>& _vek ) {
    assert( size == _vek.size );
    for(unsigned int i=0; i<size; i++)
        v[i] = _vek.v[i];
    return *this;
}
```

= als Methode

```
template <typename T>
bool Vektor<T>::operator==( const Vektor<T>& _vek ) {
    if ( size != _vek.size )
        return false;

    bool result = true;
    for(unsigned int i=0; i<size; i++)
    {
        if ( v[i] != _vek.v[i] ) {
            result = false;
            break;
        }
    }
    return result;
}
```

== als Methode

Operatorüberladung bei Klassentemplates: Beispiel

Beispiel: Klasse Vektor als Template

```
template <typename T>
Vektor<T>& Vektor<T>::operator++()
{
    for(unsigned int i=0; i<size; i++)
        v[i] += static_cast<T>(1);
    return *this;
}
```

Präfixoperator ++
als Methode

```
template <typename T>
Vektor<T> Vektor<T>::operator++( int t )
{
    Vektor<T> tmp = *this;
    for(unsigned int i=0; i<size; i++)
        v[i] += static_cast<T>(1);
    return tmp;
}
```

Postfixoperator ++
als Methode

kein &
wegen lokalem Objekt tmp

Operatorüberladung bei Klassentemplates: Beispiel

Beispiel: Klasse Vektor als Template

```
template<typename T>
Vektor<T> operator+( const Vektor<T>& _vek1, const Vektor<T>& _vek2)
{
```

```
    assert( _vek1.size == _vek2.size );
    Vektor<T> tmp = _vek1;
    tmp += _vek2;
    return tmp;
```

Wegen **friend**:
Zugriff auf private Elemente

Hier kein Schlüsselwort **friend**!

```
template<typename T>
Vektor<T> operator-( const Vektor<T>& _vek1, const Vektor<T>& _vek2)
{
```

```
    assert( _vek1.size == _vek2.size );
    Vektor<T> tmp = _vek1;
    tmp -= _vek2;
    return tmp;
```

Operator+ als globale
friend-Funktion

Operator- als globale
friend-Funktion

Zurückführung auf
Methode += bzw. -=

Vorteil der globalen Funktion:

Erstes Argument muß nicht notwendig Objekt der Klasse sein

Operatorüberladung bei Klassentemplates: Beispiel

Beispiel: Klasse Vektor als Template

```
template<typename T>
std::ostream& operator<<( std::ostream& os, const Vektor<T>& _vek ) {
    for(unsigned int i=0; i<_vek.size(); i++)
        os << _vek.v[i] << " ";
    return os;
}
```

Ausgabeoperator << als
globale **friend**-Funktion

Erstes Argument: Referenz auf **ostream**
→ Daher keine Implementierung als Methode

Aufgrund der Operatorüberladung:
→ Jetzt kann +=, -=, =, ==, ++, +, - und <<
auf Objekte der Klasse Vektor angewendet werden

Ggf. noch implementieren: --, !=, >>, etc.

Operatorüberladung bei Klassentemplates: Beispiel

Beispiel: Klasse Vektor als Template

Hauptfunktion:

```
int main(int argc, char* argv[]) {
    unsigned int i=0;
    Vektor<int> v1(5);           ← Vektor von 5 int-Elementen
    for(i=0; i<5; i++)
        v1.at(i) = i+3;          ← Initialisierung der Feldelemente
    Vektor<int> v2(5);           ← Vektor von 5 int-Elementen
    for(i=0; i<5; i++)
        v2.at(i) = i;            ← Initialisierung der Feldelemente
    Vektor<int> v3( v1++ );     ← Kopierkonstruktor und Postfixoperator
    if (! ( v3 == v1 ) )          ← Test auf Gleichheit mit ==
        cout << "v3 und v1 sind ungleich!" << endl;
    Vektor<int> v4(5);           ← Vektor von 5 int-Elementen
    v4 = v1 + v2 + v3;
    v1 = v4 - v3 - v2;          ← (verkettete) Addition und Subtraktion
    v2 += ++v1;                 ← Präfixoperator und Zuweisung += bzw. -=
    v3 -= v2;                   ← Präfixoperator und Zuweisung += bzw. -=
    cout << "v1 = " << v1 << endl << "v2 = " << v2 << endl;
    cout << "v3 = " << v3 << endl << "v4 = " << v4 << endl;   ← Direkte Ausgabe
    return 0;                      über <<
```

Operatorüberladung bei Klassentemplates: Beispiel

Beispiel: Klasse Vektor als Template

Hauptfunktion:

```
int main(int argc, char* argv[]) {  
    unsigned int i=0;  
    Vektor<int> v1(5);  
    for(i=0; i<5; i++)  
        v1.at(i) = i+3;  
    Vektor<int> v2(5);  
    for(i=0; i<5; i++)  
        v2.at(i) = i;  
    Vektor<int> v3( v1++ );  
    if (! ( v3 == v1 ))  
        cout << "v3 und v1 sind ungleich!" << endl;  
    Vektor<int> v4(5);  
  
    v4 = v1 + v2 + v3;  
    v1 = v4 - v3 - v2;  
    v2 += ++v1;  
    v3 -= v2;  
    cout << "v1 = " << v1 << endl << "v2 = " << v2 << endl;  
    cout << "v3 = " << v3 << endl << "v4 = " << v4 << endl;  
  
    return 0;  
}
```

Ausgabe?

```
v3 und v1 sind ungleich!  
v1 = 5 6 7 8 9  
v2 = 5 7 9 11 13  
v3 = -2 -3 -4 -5 -6  
v4 = 7 10 13 16 19
```

Operatorüberladung bei Klassentemplates: Beispiel

Beispiel: Klasse Vektor als Template

Hauptfunktion:

```
int main(int argc, char* argv[]) {
    unsigned int i=0;
    Vektor<int> v1(5);
    for(i=0; i<5; i++)
        v1.at(i) = i+3; // v1 = 3, 4, 5, 6, 7
    Vektor<int> v2(5);
    for(i=0; i<5; i++)
        v2.at(i) = i; // v2 = 0, 1, 2, 3, 4
    Vektor<int> v3( v1++ ); // v3 = 3, 4, 5, 6, 7; v1 = 4, 5, 6, 7, 8
    if (! ( v3 == v1 ))
        cout << "v3 und v1 sind ungleich!" << endl;
    Vektor<int> v4(5);

    v4 = v1 + v2 + v3; // v4 = 7, 10, 13, 16, 19
    v1 = v4 - v3 - v2; // v1 = 4, 5, 6, 7, 8
    v2 += ++v1; // v1 = 5, 6, 7, 8, 9; v2 = 5, 7, 9, 11, 13
    v3 -= v2; // v3 = -2, -3, -4, -5, -6
    cout << "v1 = " << v1 << endl << "v2 = " << v2 << endl;
    cout << "v3 = " << v3 << endl << "v4 = " << v4 << endl;

    return 0;
}
```

Objekte

Bei der Arbeit mit Objekten fallen die gleichen Arbeiten an, die man von der Programmierung mit einfachen Datentypen her kennt:

- Erzeugen
- Vergleichen
- Kopieren
- Ein- / ausgeben
- Auflösen

Diese Grundoperationen werden standardmäßig leider nur dürftig unterstützt

→ müssen für selbst definierte Klassen in der Regel angepasst werden,
damit sich Objekte der eigenen Klasse sinnvoll kopieren, vergleichen, ...

Objekterzeugung

- als Variable einer Klasse (Stack)

```
complex c;                      // Standardkonstruktor
class complex c;                // alternative Schreibweise
complex c(1.0, 2.0);           // überladener Konstruktor:
                                // Argumente an Variablenamen anhängen
complex c();                  // Fehler (könnte verwechselt werden
                                // mit Funktionsdeklaration)
```

- dynamisch mit „new“-Operator (Heap)

```
complex* pc = new complex;        // Standardkonstruktor
class complex *pc = new class complex; // alternat. Schreibw.
complex *pc = new complex();       // bei new: () erlaubt
complex *pc = new complex(1.0, 2.0); // überladener Konstr.
```

Objekterzeugung

- als Variable einer Klasse (Stack)

```
complex c;           // Standardkonstruktor
class complex c;    // alternative Schreibweise
complex c(1.0, 2.0); // überladener Konstruktor:
                     // Argumente an Variablennamen anhängen
complex c();      // Fehler (könnte verwechselt werden
                     // mit Funktionsdeklaration)
```

Bei Definition einer Variablen vom Typ einer Klasse wird

- die Variable (hier **c**) mit Speicherplatz für ein Objekt der Klasse angelegt
- ein Objekt der Klasse erzeugt (Konstruktoraufruf) und
- im Speicher der Variablen abgelegt.

Objekterzeugung

- dynamisch mit „new“-Operator (Heap)

```
complex* pc = new complex;           // Standardkonstruktor  
class complex *pc = new class complex; // alternat. Schreibw.  
complex *pc = new complex();         // bei new: () erlaubt  
complex *pc = new complex(1.0, 2.0); // überladener Konstr.
```

Bei Definition eines dynamisch allozierten Objekts einer Klasse wird

- Speicherplatz für ein Objekt der Klasse angelegt
- ein Objekt der Klasse erzeugt (Konstruktoraufruf)
- im Speicher der Variablen abgelegt
- die von **new** gelieferte Speicheradresse in Variabler (hier **pc**) gespeichert

Einschub: Stack und Heap (Freispeicher)

Ein benanntes Objekt (auf dem Stack) hat eine Lebensdauer, die durch seinen Gültigkeitsbereich festgelegt ist.

Beispiel:

```
void funktionDemo (void) {  
    konto k1; ←  
    // irgend etwas ...  
    return;  
}
```

Gültigkeitsbereich des (mit **k1**) benannten Konto-Objekts ist die Funktion. Außerhalb ist der Name **k1** nicht sichtbar.

→ Objekt wird zu Beginn der Funktionsarbeitung erzeugt (auf Stack) und nach Beendigung der Funktion wieder vernichtet.

Einschub: Stack und Heap (Freispeicher)

Ein benanntes Objekt (auf dem Stack) hat eine Lebensdauer, die durch seinen Gültigkeitsbereich festgelegt ist.

Beispiel:

```
void funktionDemo (void) {  
    konto k1;  
    // irgend etwas ...  
    return;  
}
```

Angenommen, das erzeugte Konto-Objekt soll auch nach Beendigung der Funktion noch weiterexistieren (z.B. könnte die Funktion einen Zeiger auf dieses Objekt an den Aufrufer zurück geben).

Wie kann man das realisieren?

Objektverwaltung

Beispiel:

```
konto* funktionDemo (int ein) {  
    konto k1("max mustermann");  
  
    k1.einzahlung(ein);  
  
    return &k1;  
}  
  
int main (void) {  
    konto *kp;  
  
    kp = funktionDemo(200);  
  
    cout << kp->toString()  
        << endl;  
  
    system("PAUSE");  
    return 0;  
}
```

Was passiert in diesem Programm?

1. Konto-Objekt wird erzeugt und in Variable k1 gespeichert.
2. Objekt-Attribut Kontostand wird verändert.
3. Adresse des Objekts wird zurück gegeben.
4. Objekt wird gelöscht.

Frage:

Worauf zeigt die zurück gegebene Adresse?

Antwort:

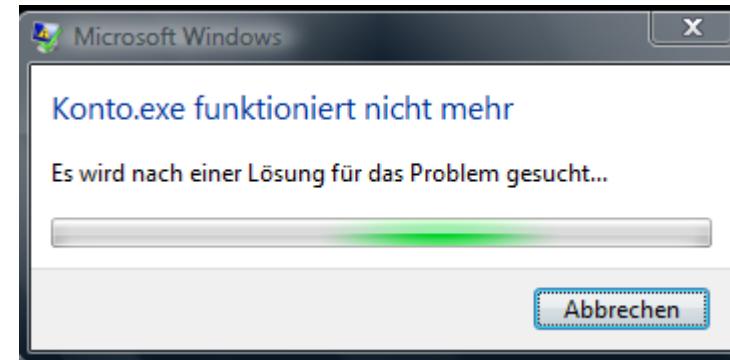
Auf einen von uns nicht mehr kontrollierten Speicherbereich. Also "irgendwo" hin.

Objektverwaltung

Beispiel:

```
konto* funktionDemo (int ein) {  
    konto k1("max mustermann");  
  
    k1.einzahlung(ein);  
  
    return &k1;  
}  
  
int main (void) {  
    konto *kp;  
  
    kp = funktionDemo(200);  
  
    cout << kp->toString()  
        << endl;  
  
    system("PAUSE");  
    return 0;  
}
```

Mögliche Verhalten:



Objektverwaltung

Beispiel:

```
konto* funktionDemo (int ein) {  
    konto k1("max mustermann");  
  
    k1.einzahlung(ein);  
  
    return &k1;  
}  
  
int main (void) {  
    konto *kp;  
  
    kp = funktionDemo(200);  
  
    cout << kp->toString()  
        << endl;  
  
    system("PAUSE");  
    return 0;  
}
```

Oder:

Kontoinhaber: Kontoinhaber: ,
Kontonummer: 4471256,
Kontostand: 2293724,
Zugriffe: 1980568541
Drücken Sie eine beliebige Taste . . .

Einschub: Stack und Heap (Freispeicher)

Lösung:

Objekt mit dem new-Operator erzeugen

→ Objekt wird im sog. *Freispeicher* (Heap, dynam. Speicher) angelegt

Objekte im Freispeicher existieren unabhängig von ihrem Gültigkeitsbereich so lange weiter, bis sie mit delete entfernt werden.

Objektverwaltung

Beispiel:

```
konto* funktionDemo (int ein) {  
    konto *k1 = new konto("max mustermann");  
  
    k1->einzahlung(ein);  
  
    return k1;  
}  
  
int main (void) {  
    konto *kp;  
  
    kp = funktionDemo(200);  
  
    cout << kp->toString() << endl;  
  
    delete kp;  
    // cout << kp->toString() << endl; //FEHLER, Objekt exist.  
    return 0;  
}
```

Was passiert in diesem Programm?

1. Konto-Objekt wird erzeugt und dessen Adresse in Variable k1 gespeichert.
2. Objekt-Attribut Kontostand wird verändert.
3. Adresse des Objekts wird zurück gegeben.
4. Objekt wird aber hier nicht gelöscht.

Erst hier wird das Objekt vom Programmierer ausdrücklich gelöscht.

Objekterzeugung und Lebensdauer

Objekte können in unterschiedlichen Kontexten erzeugt werden, z.B.

- a. bei Definition einer lokalen Variablen (Stack),
- b. als nicht-statisches Element eines anderen Objekts X, welches gemeinsam mit X erzeugt und zerstört wird (Komposition),
- c. als Element eines Feldes, welches dann gemeinsam mit dem Feld erzeugt (und auch zerstört) wird,
- d. als lokales statisches Objekt,
- e. als globales oder als statisches Klassenelement benutztes Objekt,
- f. durch Verwendung des `new`-Operators (Heap).

Von diesem Kontext hängt auch die Lebensdauer des Objektes ab.

a. Definition einer lokalen Variablen

Lebensdauer lokaler Objekte:

Erzeugung sobald Kontrollfluss durch die Variablendefinitionsstelle führt

Zerstörung sobald der Block mit der Variablendefinition verlassen wird

Beispiel:

```
void f (int i) {  
    demo aa;  
    demo bb;  
    if (i>0) {  
        demo cc;  
        // ...  
    }  
    demo dd;  
    // ...  
}
```

Aufrufreihenfolge Konstruktor/Destruktor?

Konstruktor-Aufrufreihenfolge:

- 1.aa
- 2.bb
- 3.cc (falls $i > 0$)
- 4.dd

Destruktor-Aufrufreihenfolge:

- 1.cc (wird zerstört bevor dd erzeugt wird)
- 2.dd
- 3.bb
- 4.aa

b. Objekt als Teil eines anderen Objekts: Die Komposition

- Vergleiche: Kapitel "Grundbegriffe der OOP"
- Objekte dürfen Objekte anderer Klassen enthalten
- Bei Erzeugung eines Objekts der umgebenen Klasse wird das enthaltene Objekt automatisch miterzeugt.

Aufrufreihenfolge der Konstruktoren:

- **Zuerst** werden die Konstruktoren der **enthaltenen Objekte** ausgeführt.
Reihenfolge: Reihenfolge der Deklaration in der Klasse
- **Anschließend** wird der Konstruktor für die **umgebene Klasse** aufgerufen

Aufrufreihenfolge der Destruktoren:

- Zuerst wird der Destruktor für die umgebene Klasse aufgerufen
- Danach werden die Destruktoren der enthaltenen Objekte ausgeführt

Objektverwaltung

```
class umgebend
{
public:
    enthalten obj1;

    umgebend() {
        cout << "Konstruktor der \
Klasse umgebend" << endl;
    }

    ~umgebend() {
        cout << "Destruktor der \
Klasse umgebend" << endl;
    }
};
```

```
class enthalten
{
    double wert;

public:
    enthalten() : wert(1) {
        cout << "Konstruktor der \
Klasse enthalten" << endl;
    }

    ~enthalten() {
        cout << "Destruktor der \
Klasse enthalten" << endl;
    }
};
```

```
int main() {
    umgebend x;

    return 0;
}
```

Ausgabe?

Konstruktor der Klasse enthalten
Konstruktor der Klasse umgebend
Destruktor der Klasse umgebend
Destruktor der Klasse enthalten

c. Objekt als Feldelement

- Objekte können in Feldern (Arrays) organisiert werden
- a) Standardkonstruktor vorhanden: Initialisierung ohne Konstruktor möglich

Beispiel:

```
demo obj[10]; // erzeugt Feld mit 10 Objekten der Klasse obj  
// & initialisiert diese m. Standardkonstruktor
```

- Konstruktoraufruf: nacheinander von obj[0] bis obj[9]
- Destruktoraufruf: nacheinander von obj[9] bis obj[0]

- b) Standardkonstruktor *nicht* vorhanden: Initial. über Initialisierungsliste

Beispiel:

```
demo obj[3] = {demo(1,2), demo(2,3), demo(3,4)};
```

d. Lokales statisches Objekt

Zur Erinnerung:

Statische Variable in einer Funktion

- *Lokale Variablen* in einer Funktion werden bei jedem Funktionsaufruf neu angelegt und ggf. initialisiert.
 - Jeder Funktionsaufruf hat seine *eigene Kopie der lokalen Variable*
- Eine **statische Variable** wird nur **ein einziges Mal** im Speicher angelegt und *für alle Funktionsaufrufe verwendet.*
 - Alle Funktionsaufrufe arbeiten mit derselben Variable
 - Variable wird nur beim ersten Aufruf der Funktion initialisiert
 - Variable behält ihren Wert über mehrere Aufrufe hinweg
 - "Gedächtnis" der Funktion, ohne Verwendung einer globale Variablen

d. Lokales statisches Objekt

Beispiel:

```
void func() {  
    int x = 0;  
    static int y = 0;  
  
    x++;  
    y++;  
    printf("x = %d, y = %d\n", x, y);  
}  
  
int main(int argc, char *argv[]) {  
    for (int i=0; i<5; i++) {  
        func();  
    }  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

Ausgabe?

```
x = 1, y = 1  
x = 1, y = 2  
x = 1, y = 3  
x = 1, y = 4  
x = 1, y = 5
```

d. Lokales statisches Objekt

Anstelle der lokalen statischen *Variablen* jetzt: lokales statisches *Objekt*

Beispiel:

```
void func (int i) {  
  
    static meineKlasse obj;  
  
    // ...
```

Eigenschaften

Ähnlich wie Eigenschaften einer lokalen statischen Variable

- Es existiert für alle Funktionsaufrufe **nur ein Objekt**, welches gemeinsam genutzt wird. D.h. Objekt wird beim Verlassen der Funktion nicht zerstört.
- Konstruktor eines lokalen statischen Objekts wird ***nur ein Mal*** beim ersten Erreichen der Deklarationsstelle ausgeführt.
- Destruktoraufruf: bei Programmende in umgekehrter Konstr.-Reihenfolge

Objektverwaltung

Beispiel:

```
class demo
{
    double wert;

public:
    demo() : wert(1) {
        cout << "Konstruktor, wert = " << wert << endl;
    }

    ~demo() {
        cout << "Destruktor, wert = " << wert << endl;
    }

    void incWert() {wert++;}
    double getWert() {return wert;}
};
```

Objektverwaltung

```
void func (int i) {  
    cout << "Aufruf func()" << endl;  
    static demo d1;  
    cout << "Wert Objekt d1 = ";  
    cout << d1.getWert() << " " << endl;  
  
    if (i>0) {  
        static demo d2;  
        cout << "Wert Objekt d2 = ";  
        cout << d2.getWert() << " " << endl;  
        d2.incWert();  
    }  
    d1.incWert();  
    cout << " " << endl;  
    //d2.incWert(); // Fehler  
}  
  
int main() {  
    for (int i=0; i<4; i++)  
        func(i);  
    system("pause");  
    return 0;  
}
```

Block wird erst bei Aufruf mit $i>0$ durchlaufen
→ Objekt d2 wird erst dann konstruiert.

Beide Objekte werden nur ein Mal initialisiert,
unabhängig davon, wie häufig die Funktion
aufgerufen wird.

Objektverwaltung

Ausgabe ?

```
D:\Dev-Cpp>Projekt3.exe
```

```
Aufruf func()
```

```
Konstruktor, wert = 1
```

```
Wert Objekt d1 = 1
```

```
Aufruf func()
```

```
Wert Objekt d1 = 2
```

```
Konstruktor, wert = 1
```

```
Wert Objekt d2 = 1
```

```
Aufruf func()
```

```
Wert Objekt d1 = 3
```

```
Wert Objekt d2 = 2
```

```
Aufruf func()
```

```
Wert Objekt d1 = 4
```

```
Wert Objekt d2 = 3
```

```
Drücken Sie eine beliebige Taste . . .
```

```
Destruktor, wert = 4
```

```
Destruktor, wert = 5
```

Bemerkungen:

- Die Destruktor-Ausgabe ist nur zu sehen, wenn das Programm von der Kommandozeile aus aufgerufen wird.
- Der Destruktor von Objekt d2 wird als erstes aufgerufen – danach der von d1.

e. Globales oder als statisches Klassenelement benutztes Objekt

- Erzeugung bei Definition
- Zerstörung am Programmende

→ Beispiel auf der nächsten Seite

e. Globales oder als statisches Klassenelement benutztes Objekt

```
class demo
{
    int wert;

public:
    demo( int w = 0 ) : wert(w) { cout << "Konstruktor von demo,
                                    wert = " << wert << endl; }
    ~demo() { cout << "Destruktor von demo, wert = " << wert << endl; }
};
```

```
class demo2
{
    int wert;
    static demo d;      // statische Klassenvariable

public:
    demo2( int w = 0 ) : wert(w) { cout << "Konstruktor von demo2,
                                    wert = " << wert << endl; }
    ~demo2() { cout << "Destruktor von demo2, wert = " << wert << endl; }
};
```

e. Globales oder als statisches Klassenelement benutztes Objekt

```
demo d(1);
```

globales Objekt

```
demo demo2::d(2);
```

als statisches Klassenelement benutztes Objekt

```
int main( int argc, char* argv[] ) {
    cout << "main startet" << endl;
    demo2* pd2;
    if (1)  {
        cout << "start if-Block" << endl;
        demo d1(3);
        pd2 = new demo2(4);
        cout << "end if-Block" << endl;
    }
    cout << "zurueck in main" << endl;
    delete pd2;
    system("PAUSE");
    cout << "mainendet" << endl;
    return 0;
}
```

e. Globales oder als statisches Klassenelement benutztes Objekt

```
demo d(1);

demo demo2::d(2);

int main( int argc, char* argv[] ) {
    cout << "main startet" << endl;
    demo2* pd2;
    if (1)  {
        cout << "start if-Block" << endl;
        demo d1(3);
        pd2 = new demo2(4);
        cout << "end if-Block" << endl;
    }
    cout << "zurueck in main" << endl;
    delete pd2;
    system("PAUSE");
    cout << "mainendet" << endl;
    return 0;
}
```

Ausgabe?

```
Konstruktor von demo, wert = 1
Konstruktor von demo, wert = 2
main startet
start if-Block
Konstruktor von demo, wert = 3
Konstruktor von demo2, wert = 4
end if-Block
Destruktor von demo, wert = 3
zurueck in main
Destruktor von demo2, wert = 4
Drücken Sie eine beliebige Taste . . .
main endet
Destruktor von demo, wert = 2
Destruktor von demo, wert = 1
```

e. Globales oder als statisches Klassenelement benutztes Objekt

- Erzeugung der globalen Objekte bei Definition (vor `main`)
- Erzeugung von `pd2` erst bei Aufruf von `new`
- Vernichtung der globalen Objekte am Programmende in umgekehrter Reihenfolge wie Erzeugung

Ausgabe?

```
Konstruktor von demo, wert = 1
Konstruktor von demo, wert = 2
main startet
start if-Block
Konstruktor von demo, wert = 3
Konstruktor von demo2, wert = 4
end if-Block
Destruktor von demo, wert = 3
zurueck in main
Destruktor von demo2, wert = 4
Drücken Sie eine beliebige Taste . . .
main endet
Destruktor von demo, wert = 2
Destruktor von demo, wert = 1
```

f. Mit „new“ erzeugte dynamische Variable (Heap)

- Bei der Objekterzeugung mit dem `new`-Operator wird ein neues Objekt im **Freispeicher** angelegt.
- Dieses Objekt wird erst wieder zerstört, wenn der `delete`-Operator darauf angewendet wird.

Lebensdauer von Objekten: Zusammenfassung

- lokale Variable (Stack):
Erzeugung sobald Kontrollfluss durch die Variablendefinitionsstelle führt
Zerstörung sobald der Block mit der Variablendefinition verlassen wird
- lokales statisches Objekt:
Erzeugung beim ersten Erreichen der Deklarationsstelle (nur 1 Mal!)
Zerstörung am Programmende
- globales oder als statisches Klassenelement benutztes Objekt:
Erzeugung bei Definition
Zerstörung am Programmende
- mit „new“ dynamisch erzeugte Variable (Heap):
Erzeugung sobald Kontrollfluss an die Stelle des `new`-Operators kommt
Zerstörung erst bei Anwendung des `delete`-Operators

Objektvergleich

- Vergleich bedeutet: Gleichheit, kleiner als, größer als
- Die Bedeutung des Objektvergleichs muss vom Autor der Klasse festgelegt werden.
 - Üblicherweise:
 - Vergleich des Wertes einer Instanzvariable
 - Vergleich einer Kombination aus mehreren Instanzvariablen
- Nicht immer ist ein Vergleich überhaupt sinnvoll

Beispiel: Was bedeutet die Gleichheit zweier Konten?

→ gleicher Kontostand?

→ gleicher Kontoinhaber?

Sind zwei Konten gleich, wenn sie den gleichen Kto.stand aufweisen?

Objektvergleich

Zur Definition der Gleichheit/Ungleichheit müssen die Operatoren `==` und `!=` überladen werden:

- Gleichheitsoperator liefert `true`, wenn die Objekte gleich sind
- Ungleichheitsoperator liefert `false`, wenn die Objekte gleich sind

Eigenschaften des Gleichheitsoperators:

- reflexiv: Vergleich eines Objekts mit sich selbst muss `true` liefern
- symmetrisch: Ergebnis von `a == b` muss gleich `b == a` sein
- transitiv: Wenn `a == b` und `b == c` muss folgen `a == c`

Eigenschaften des Ungleichheitsoperators:

- reflexiv und symmetrisch, aber nicht transitiv

Bemerkung:
Analoges Vorgehen für
`<`, `<=`, `>`, `>=`

Objektverwaltung

```
class punkt
{
    double x;
    double y;

public:
    punkt() : x(0), y(0) {}
    punkt(int x_arg, int y_arg) : x(x_arg), y(y_arg) {}

    bool operator==(const punkt v);
    bool operator!=(const punkt v);

};

bool punkt::operator==(const punkt v)
{
    return (x == v.x) && (y == v.y);
}

bool punkt::operator!=(const punkt v)
{
    return !(*this == v);
    // oder auch: return (x != v.x) || (y != v.y);
}
```

Beispiel: Überladung als Methode

Objektverwaltung

```
class punkt
{
    double x;
    double y;

public:
    punkt() : x(0), y(0) {}
    punkt(int x_arg, int y_arg) : x(x_arg), y(y_arg) {}

};

bool operator== (const punkt v1, const punkt v2)
{
    return (v1.x == v2.x) && (v1.y == v2.y);
}

bool operator!= (const punkt v1, const punkt v2)
{
    return !(v1 == v2);
    // oder auch: return (v1.x != v2.x) || (v1.y != v2.y);
}
```

Beispiel: Überladung als globale Funktion

Frage:

Funktioniert das so?

Antwort:

Nein, Zugriff auf
private Instanzvar.
nicht möglich.

Objektverwaltung

```
class punkt
{
    double x;
    double y;

public:
    punkt() : x(0), y(0) {}
    punkt(int x_arg, int y_arg) : x(x_arg), y(y_arg) {}

    friend bool operator== (const punkt v1, const punkt v2);
    friend bool operator!= (const punkt v1, const punkt v2);

};

bool operator== (const punkt v1, const punkt v2)
{
    return (v1.x == v2.x) && (v1.y == v2.y);
}

bool operator!= (const punkt v1, const punkt v2)
{
    return !(v1 == v2);
    // oder auch: return (v1.x != v2.x) || (v1.y != v2.y);
}
```

Beispiel: Überladung als globale Funktion

Lösung:

Als "friend" deklarieren.

Objektverwaltung

```
int main()
{
punkt v1;
punkt v2(10, 0);
punkt v3(10, 10);
punkt v4(0, 0);
punkt v5(0, 0);

// ==-Operator testen
cout << endl << " ==-Operator testen " << endl;
if (v1 == v2)
;
else
    cout << "Unterschied in x korrekt erkannt" << endl;
if (v2 == v3)
;
else
    cout << "Unterschied in y korrekt erkannt" << endl;
if (v4 == v1)
    cout << "Gleichheit korrekt erkannt" << endl;
if (v2 == v2)
    cout << "Reflexivitaet korrekt erkannt" << endl;
```

Beispiel (Hinweis: `operator==` und `operator!=` sind *entweder* als Methode *oder* als globale Funktion implementiert)

Objektverwaltung

```
// Fortsetzung                                Beispiel (Methode oder globale Funktion)
if (v1 == v4 && v4 == v1)
    cout << "Symmetrie korrekt erkannt" << endl;
if (v1 == v4 && v4 == v5 && v1 == v5)
    cout << "Transitivitaet korrekt erkannt" << endl;

// !=-Operator testen
cout << endl << " !=-Operator testen " << endl;
if (v1 != v2)
    cout << "Unterschied in x korrekt erkannt" << endl;
if (v2 != v3)
    cout << "Unterschied in y korrekt erkannt" << endl;
if (v4 != v1)
    ;
else
    cout << "Gleichheit korrekt erkannt" << endl;
if (v2 != v2)
    ;
else
    cout << "Reflexivitaet korrekt erkannt" << endl;

return 0;
}
```

Ausgabe:

```
== Operator testen
Unterschied in x korrekt erkannt
Unterschied in y korrekt erkannt
Gleichheit korrekt erkannt
Reflexivitaet korrekt erkannt
Symmetrie korrekt erkannt
Transitivitaet korrekt erkannt
```

```
!= Operator testen
Unterschied in x korrekt erkannt
Unterschied in y korrekt erkannt
Gleichheit korrekt erkannt
Reflexivitaet korrekt erkannt
Drücken Sie eine beliebige Taste . . .
```

Kopieren von Objekten

Kopieren eines Objekts `obj1` bedeutet in der Regel:

1. Bereitstellung von genügend viel Speicherplatz für eine Objektkopie `obj2`
2. Kopieren der Werte aller Instanzvariablen des Objekts `obj1` in die zugehörigen Instanzvariablen des Objekts `obj2`

wert1	123
wert2	456

`demo obj1`

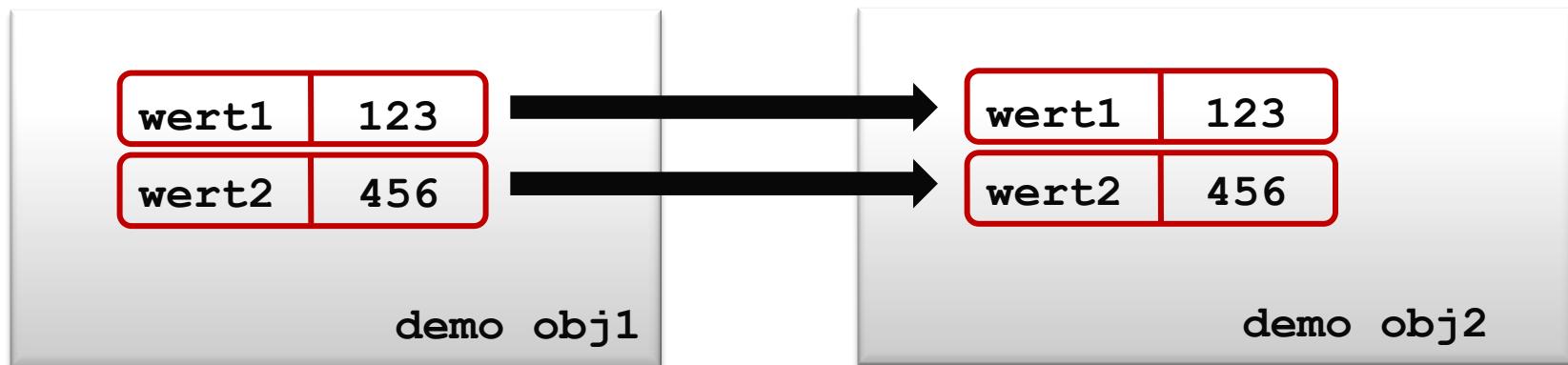
wert1	undefined
wert2	undefined

`demo obj2`

Kopieren von Objekten

Kopieren eines Objekts `obj1` bedeutet in der Regel:

1. Bereitstellung von genügend viel Speicherplatz für eine Objektkopie `obj2`
2. Kopieren der Werte aller Instanzvariablen des Objekts `obj1` in die zugehörigen Instanzvariablen des Objekts `obj2`

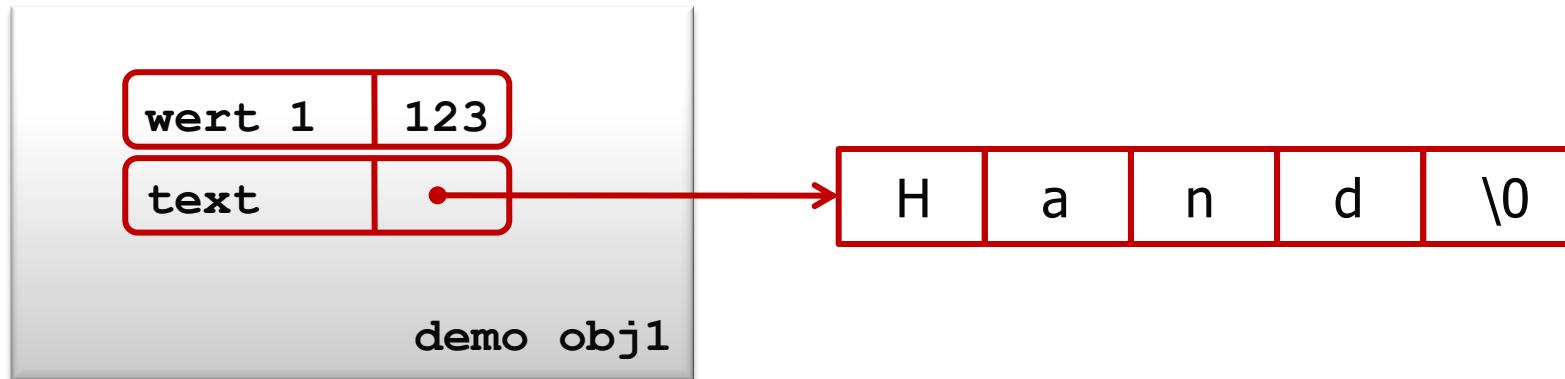


Diese Art des Kopierens bezeichnet man als **flache Kopie**.

Kopieren von Objekten

Unklar dabei bislang: Umgang mit Zeigern als Instanzvariablen?

Frage insbesondere: Soll die *Adresse* kopiert werden oder das *Objekt*, auf das der Zeiger verweist?



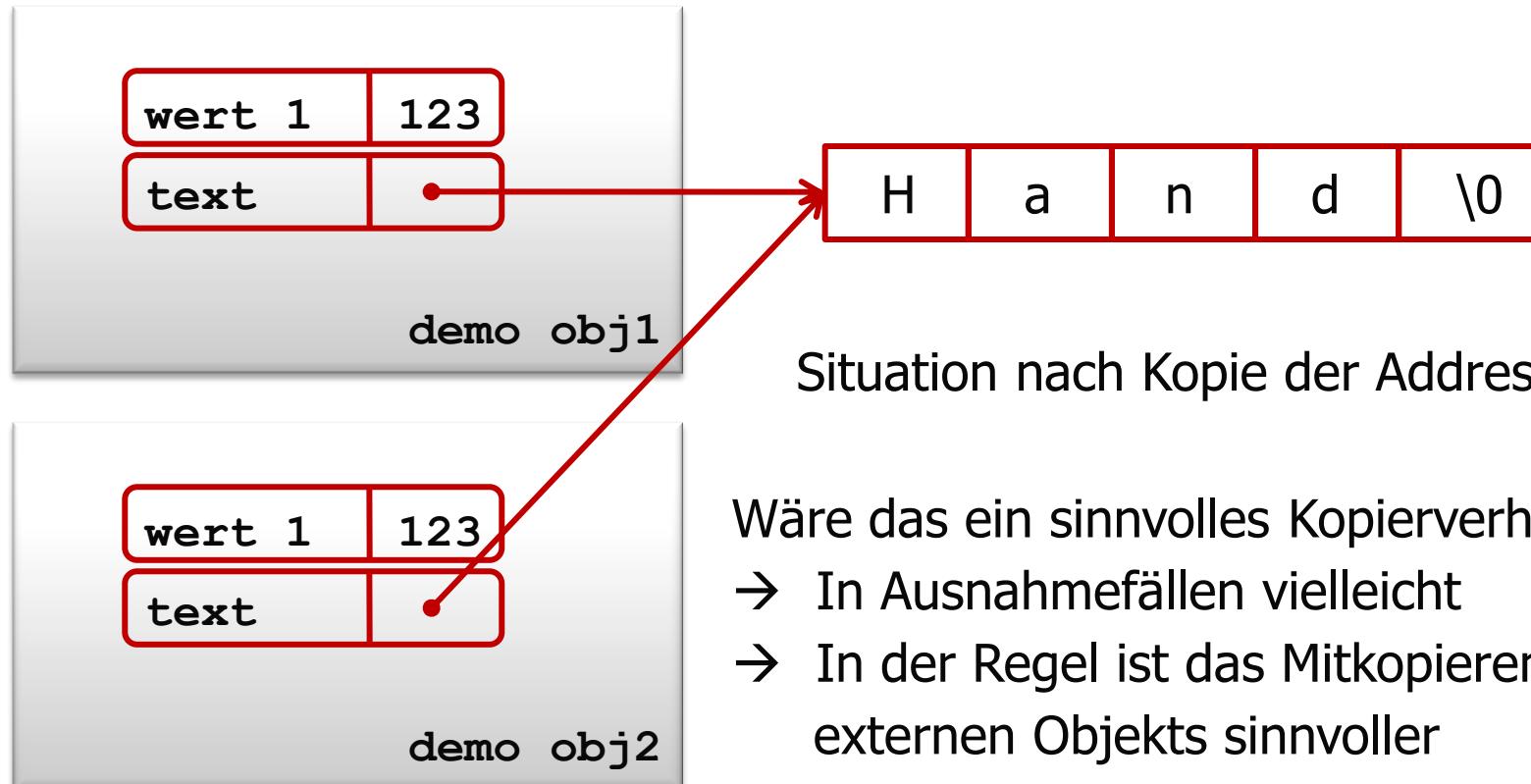
Frage: Welche Art des Kopierens (Adresse oder Objekt) würde beim Anfertigen einer *flachen* Kopie durchgeführt werden?

Antwort: Es würde lediglich die Adresse kopiert werden.

Kopieren von Objekten

Unklar dabei bislang: Umgang mit Zeigern als Instanzvariablen?

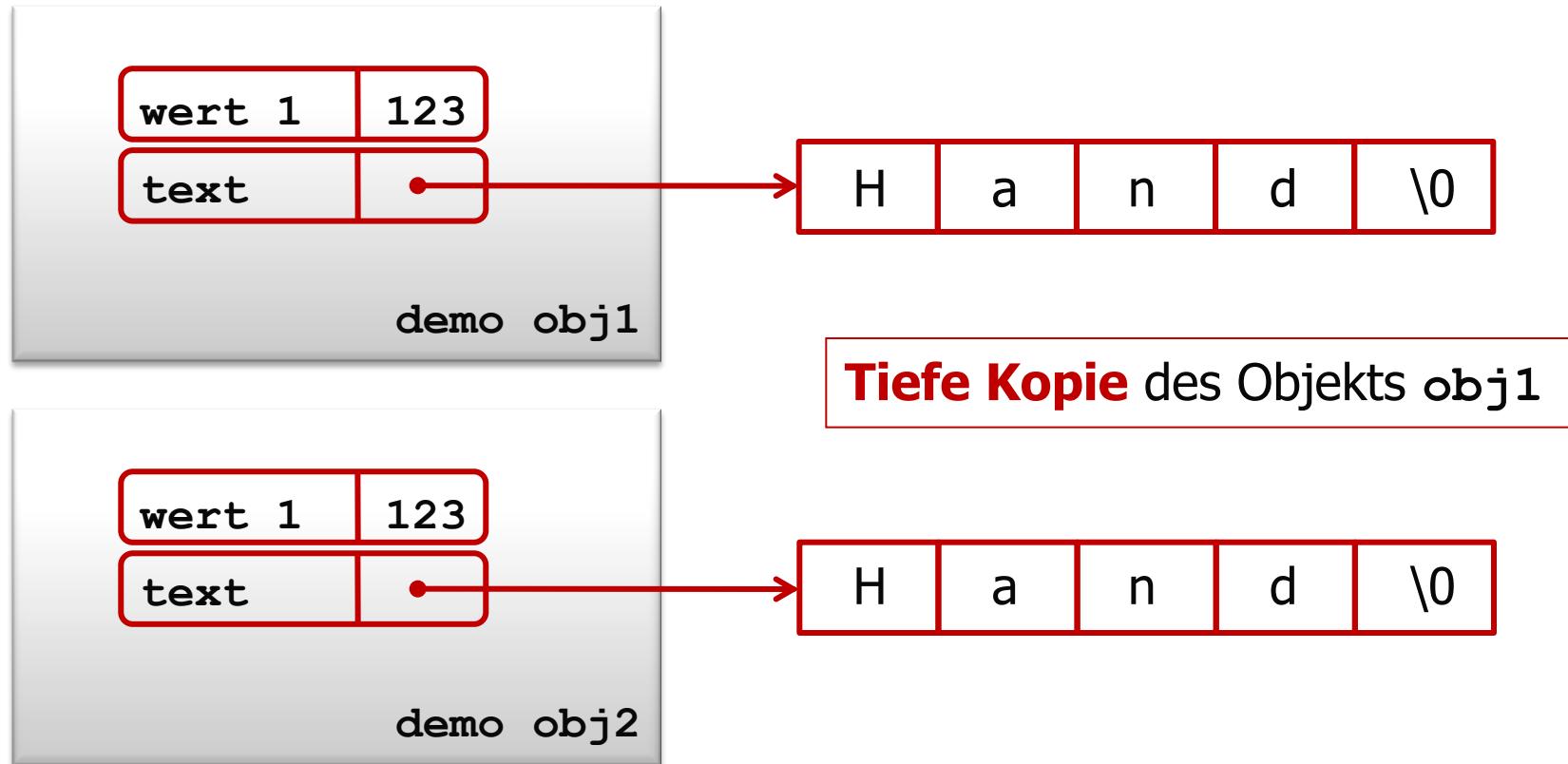
Frage insbesondere: Soll die *Adresse* kopiert werden oder das *Objekt* auf das der Zeiger verweist?



Kopieren von Objekten

Unklar dabei bislang: Umgang mit Zeigern als Instanzvariablen?

Frage insbesondere: Soll die *Adresse* kopiert werden oder das *Objekt* auf das der Zeiger verweist?



Kopieren von Objekten

Beim Kopieren von Objekten ist immer eines der folgenden Elemente beteiligt:

- Zuweisungsoperator =
- Kopierkonstruktor

Beispiel:

```
demo obj1, obj2;  
obj2 = obj1;           // Zuweisung -> Zuweisungsoperator  
  
demo obj3 = obj1;    // Initialisierung (KEINE Zuweisung!)  
                     // -> wird durch Kopierkonstruktor realisiert  
  
demo obj3( obj1 ); // alternative Schreibw. zu demo obj3 = obj1  
                     // -> wird durch Kopierkonstruktor realisiert
```

Objekt wird einer Variablen zugewiesen

Objekt wird initialisiert -> Kopierkonstruktur

Kopieren von Objekten

Kopierkonstruktor

Der Kopierkonstruktor ist ein Konstruktor mit genau einem Argument:

- (konstante) Referenz auf ein Objekt der eigenen Klasse

Beispiel: Prototyp eines Kopierkonstruktors

```
demo (const demo &org);
```

Wirkung:

Kopiert die Werte des Objekts, auf das die Referenz zeigt, in das neu erstellte.

Falls kein eigener Kopierkonstruktor definiert wird → System stellt automat.

einen Standard-Kopierkonstruktor für die Erstellung einer *flachen* Kopie bereit

Kopieren von Objekten

Ausführung des Kopierkonstruktors ...

- a. ... wenn dieser explizit zur Instanzbildung aufgerufen wird

Beispiel:

```
demo obj2(obj1); // Initialisierung  
demo obj2 = obj1; // Initialisierung
```

Kopieren von Objekten

Ausführung des Kopierkonstruktors ...

- b. ... wenn ein Objekt einer Funktion als Wert übergeben wird (call by value)

Beispiel:

```
void func(demo par) {  
    // ...  
}  
  
int main (void) {  
    demo obj1;  
    // ...  
    func(obj1);  
    // ...
```

Beim Aufruf der Funktion wird das Objekt mit Hilfe des Kopierkonstruktors in den Parameter **par** kopiert. Nach dem Ende der Funktion wird das Objekt in **par** wieder gelöscht.

Kopieren von Objekten

Ausführung des Kopierkonstruktors ...

- c. ... wenn eine Funktion ein Objekt als Rückgabewert zurück liefert (return by value)

Beispiel:

```
demo func(demo par) {  
    // ...  
    return par;  
}  
  
int main (void) {  
    demo obj1, obj2;  
    // ...  
    obj2 = func(obj1);
```

Das beim Aufruf in den Parameter **par** kopierte Objekt wird nach Ende der Funktion wieder zerstört. Vorher wird es jedoch durch die Rückgabe-Anweisung noch einmal kopiert und anschließend der Variablen **obj2** zugewiesen.

Objektverwaltung

Vollständiges Beispiel zu c.:

```
class vektor {  
    double x;  
    double y;  
  
public:  
    vektor() : x(0), y(0) {}  
    vektor(int x_arg = 0, int y_arg = 0) : x(x_arg), y(y_arg) {}  
    vektor( const vektor& v ) { x = v.x; y = v.y;  
        cout << "Kopierkonstruktor" << endl; }  
    void setX(double x){this->x = x;}  
    void setY(double y){this->y = y;}  
    void out() {cout << "x = " << x << " y = " << y << endl;}  
};  
  
int main() {  
    vektor v1, v2(10, 0);  
    cout << "vor func" << endl;  
    v1 = func(v2);  
    cout << "nach func" << endl;  
    vektor v3(v1);  
    return 0;  
}
```

Externe Funktion:

```
vektor func(vektor par) {  
    cout << "in func" << endl;  
    par.setX(20.0);  
    cout << "func ende" << endl;  
    return par; }
```

Nur für Ausgabe

Ausgabe ?

```
vor func  
Kopierkonstruktor  
in func  
func ende  
Kopierkonstruktor  
nach func  
Kopierkonstruktor
```

Kopieren von Objekten

Wenn Programmierer der Klasse keinen Zuweisungsoperator und Kopierkonstruktor vorsieht, werden vom System *Standardvarianten* bereit gestellt.

Diese Standardvarianten erzeugen lediglich eine *flache Kopie* des Objekts

→ Wird eine *tiefe Kopie* benötigt, so müssen Konstruktor und Zuweisungsoperator für die betreffende Klasse *überladen* werden.

Was bedeutet das?

- Definition eines Kopierkonstruktors
- Überladung des Zuweisungsoperators
- i.a. auch Definition eines Destruktors



„Dreierregel“
„Regel der Großen Drei“);
siehe später

Definition des Kopierkonstruktors

Definition des Kopierkonstruktors bedeutet Überladung des Konstruktors, so dass eine Version mit folgendem Prototyp existiert:

```
Klassenname (const Klassenname &argument);
```

Dieser Konstruktor muss dafür sorgen, dass das neu erzeugte Objekt eine (tiefe) Kopie des Objekts ist, dessen Referenz übergeben wurde.

Objektverwaltung

Beispiel: Kopierkonstruktor für tiefe Kopie

```
class demo
{
public:
    int    wert1;
    double wert2;
    char   *text;

    // Konstruktor
    ...
    // Kopierkonstruktor
    demo(const demo &org);
    ...
};
```

```
// Kopierkonstruktor
demo::demo(const demo& org)
{
    wert1 = org.wert1;
    wert2 = org.wert2;
    text = new char[strlen(org.text)+1];
    strcpy(text,org.text);
}
```

sorgt für tiefe Kopie

```
int main()
{
    demo obj1 = demo(12, 7, "Hallo");
    demo obj2(obj1); // Mit Kopierkonstruktor eine Kopie erstellen
    ...
}
```

Überladung des Zuweisungsoperators

Zur Erinnerung (aus Kapitel "Überladen von Operatoren"):

- Zuweisungsop. kann nur als Instanzfkt., nicht als globale Fkt. überladen werden!

Operatorüberladung als Instanzfunktion

`aa = bb` wird vom Compiler als `aa.operator=(bb)` interpretiert

→ Instanzfunktion `operator=()` mit einem Parameter und geeignetem Rückgabedatentyp (`x` falls Ergebnis von `aa = bb` vom Typ `x` ist) so definieren, dass sie d. gewünschte Eigenschaft des Operators hat.

`x operator=(x); // Prototyp der Operatorfunktion`

Bemerkung zum Rückgabewert: Zuweisungen dürfen auch in einem anderen Ausdruck (z.B. if-Anweisung) verwendet werden.

Beispiel: `if (a = b)`

Daher muss die Zuweisung einen Rückgabewert haben: den Wert des zugewiesenen Objekts.

Objektverwaltung

Beispiel: Zuweisungsoperator für tiefe Zuweisung

```
class demo
{
public:
    int     wert1;
    double  wert2;
    char   *text;

    // Konstruktor
    ...
    // Zuweisungsoperator
    demo& operator=(const
                      demo &op);
    ...
};

int main()
{
    demo obj1 = demo(12, 7, "Hallo");
    demo obj2;
    obj2 = obj1; // Zuweisungsoperator
    ...
}
```

```
// Zuweisungsoperator
demo& demo::operator=(const demo &op)
{
    if (this == &op) // Schutz vor
        return *this; // Selbstzuweisung

    wert1 = op.wert1;
    wert2 = op.wert2;

    delete[] text; //alten String loeschen
    text = new char[strlen(op.text)+1];
    strcpy(text,op.text);

    return *this;
}
```

sorgt für tiefe Zuweisung

(Ggf. `text` bei Initialisierung und
nach jedem „`delete`“ auf `NULL` setzen,
um hier durch „`delete`“ keinen
nicht-existenten Speicher zu löschen)

Kopierkonstruktor und Zuweisungsoperator

... sind ähnlich, aber nicht identisch:

- **Zuweisungsoperator:** kann von vollständig initialisiertem Objekt ausgehen
 - **Kopierkonstruktor:** muß i.a. erst ein vollständiges Objekt erzeugen
 - u.U. erst Speicher holen, dann Objektinitialisierung
- Kopierkonstruktor kann sich ggf. auf Zuweisungsoperator stützen:

```
// Zuweisungsoperator
demo& demo::operator=(const demo &op)
{
    if (this == &op) // Schutz vor
        return *this; // Selbstzuweisung

    wert1 = op.wert1;
    wert2 = op.wert2;
    // keine Allokation (Speicher vorh.)
    strcpy(text,op.text); ←
    return *this;
}
```

```
// Kopierkonstruktor
demo::demo(const demo& org)
{
    // Allokation
    text = new char[MAX_STRING_LENGTH];
    *this = org; // Zuweisungsoperator
}
```

Hier wird davon ausgegangen, daß für „text“ bereits ein Feld der Länge MAX_STRING_LENGTH alloziert wurde

Objektverwaltung

Mehrfachzuweisungen

```
int main()
{
    demo obj1 = demo(12, 7, "Hallo");
    demo obj2, obj3, obj4;

    obj2 = obj1;           // o.k.
    obj3 = obj4 = obj1;   // ???
    ...
}
```

obj3.operator=(obj4.operator=(obj1));

(Referenz auf) Objekt
der Klasse demo

Anwendung des linken `operator=` nur zulässig, wenn Rückgabewert
des rechten `operator=` (Referenz auf) Objekt der Klasse `demo`;
bei Rückgabewert `void`: keine Mehrfachzuweisung möglich!

- Liefert der Operator eine Referenz auf eigenes Objekt → Operatorverkettung möglich
- Eine Kette von Zuweisungen wird von rechts nach links abgearbeitet

Daumenregel: „Dreierregel“ („Regel der Großen Drei“)

- Kopierkonstruktor
- Destruktor
- Zuweisungsoperator

Wenn eine Klasse eines dieser drei Elemente definiert, sollten i.a. auch die beiden anderen Elemente definiert werden

Grund:

- Die Definition eines der Elemente bedeutet, daß die compilergenerierte Standardvariante hierfür nicht ausreicht
- Dann gilt dies vermutlich auch für die anderen Elemente, d.h. auch diese sollten entsprechend definiert werden

Ein- und Ausgabe von Objekten

- geeignete Überladung der Umleitungsoperatoren << und >>
Bsp: Überladung des Ausgabeoperators “<<” als globale Funktion

```
class Person{  
    char Name[100];  
    int age;  
    bool married;  
  
public:  
    Person(...) {...}  
    friend ostream& operator<<( ostream& s, const Person& x );  
};
```

```
int main(int argc, char* argv[]) {  
    Person x(...), y(...);  
    cout << x << y << endl;  
    return 0;  
}
```

```
Referenz auf Streamobjekt      Referenz auf Streamobjekt  
↓                          ↓  
ostream& operator<< ( ostream& s, const Person& x ) {  
    s << x.Name << endl << x.age << endl;  
    s << (x.married ? "verheiratet" : "ledig") << endl;  
    return s; // Rückgabe einer Referenz auf s wegen Verkettung von <<  
}
```

(const) Referenz auf Klassenobjekt
oder Klassenobjekt als Wert

Bem.: Objekt *zweiter* Operand → Überladung als Instanzfkt. von „Person“ nicht möglich

Auflösen von Objekten

Objekt im Speicher einer Variablen wird aufgelöst, wenn

- der Variablen ein neues Objekt zugewiesen wird oder
 - die Variable ihre Gültigkeit verliert:
 - lokale Variable (Stack): bei Verlassen des umschließenden Blocks
 - statische Variable: am Programmende
 - Instanzvariable: bei Auflösen des übergeordneten Objektes
 - mit `new` dynamisch erzeugte Variable (Heap): bei Aufruf von `delete`
- jedes „new“ hat ein „delete“!
- dynamisch erzeugte **Arrays**: Verwendung von `delete[]`
Bsp.: `Person* pp = new Person[10];
delete[] pp;`
 - Wann sollte ein **eigener Destruktor** definiert werden?
 - Falls **zusätzl. Ressourcen** (z.B. dynam. Speicher) alloziert wurden
 - Falls „Abschlußarbeiten“ vorgenommen werden sollen
 - Im Falle von Vererbung zur **virtual**-Deklaration (siehe später)

Vom Kompiler zur Verfügung gestellte Konstruktoren / Destruktoren

- Defaultkonstruktor
 - falls kein eigener Konstruktor definiert wurde
 - keine Initialisierung der Instanzvariablen!
 - Defaultkopierkonstruktor
 - *flache* Kopie
 - Defaultzuweisungsoperator
 - *flache* Zuweisung
 - Defaultdestruktor
 - keine Freigabe dynamischen Speichers,
externer Ressourcen etc.
- Bruch b;**
Bruch (const Bruch& b);
Bruch c(1,2), d;
d = c;
~Bruch();

Übersicht (1)

Objektorientierte Programmierung (OOP):

- Einführung
- Objekt und Klasse
- Attribute, Methoden
- Geheimnisprinzip, Kapselung
- Vererbung
- Klassifikation
- Beziehungen zwischen Klassen
- Polymorphie

Schnellkurs C++:

- Einführung
- **Bekannte Sprachmittel:**
 - Variablen, Datentypen, Operatoren, Kontrollstrukturen, Arrays, Strukturen
- **Neue Sprachmittel:**
 - Referenzen
 - Funktionen: Vorgabeargumente, Überladung, Templates
 - Namensräume
 - Ein- und Ausgabe
 - Strings
 - Typumwandlung

Objektorientierte Programmierung mit C++:

- Klassen, Vererbung, Polymorphie, Mehrfachvererbung

Übersicht (2)

Objektorientierte Programmierung mit C++:

- **Klassen**
 - Klasse als Datentyp
 - Member: Instanzvariablen / -methoden, Klassenvariablen / -methoden, Konstruktor, Destruktor, this-Zeiger
 - Sichtbarkeit und Kapselung, Zugriffsspezifizierer, friends
 - Designempfehlungen: const, get/set,
 - Klassentemplates
 - Operatorüberladung für Klassenobjekte
 - Objektverwaltung: Erzeugen, Vergleichen, Kopieren, Auflösen, Komposition...
- **Vererbung**
 - Syntax und Einsatz
 - Basisklassen-Unterobjekt
 - Verdecken, Überschreiben, Überladen
 - Zugriffsmodifizierer, Zugriffsrechte

Übersicht (3)

Objektorientierte Programmierung mit C++:

- Polymorphie
 - Frühe und späte Bindung
 - Virtuelle Funktionen, virtueller Destruktor
 - Abstrakte Methoden
 - Abstrakte Klassen
- Mehrfachvererbung
- Fehlerbehandlung (exception handling)

Objektorientierte Programmierung mit C++: Vererbung

Vererbung: Einführung

Betrachte Klasse **BankAccount**:

```
class BankAccount
{
    int balance;
    int accountNr;
    int clientNr;
    ...
public:
    BankAccount( ... ) { ... }
    ...
    void deposit (int amount)
    { balance += amount; }

    bool withdraw(int amount)
    { balance -= amount;
        return true; }
};
```

BankAccount

- balance: int
 - accountNr: int
 - clientNr: int
- + deposit(): void
 - + withdraw(): bool

Vererbung: Einführung

Betrachte Klasse **BankAccount**:

```
class BankAccount
{
    int balance;
    int accountNr;
    int clientNr;
    ...
public:
    BankAccount( ... ) { ... }
    ...
    void deposit (int amount)
    { balance += amount; }

    bool withdraw(int amount)
    { balance -= amount;
        return true; }
};
```

Was passiert, wenn wir die Klasse weiter spezialisieren wollen?

z.B. Klasse **GiroAccount**

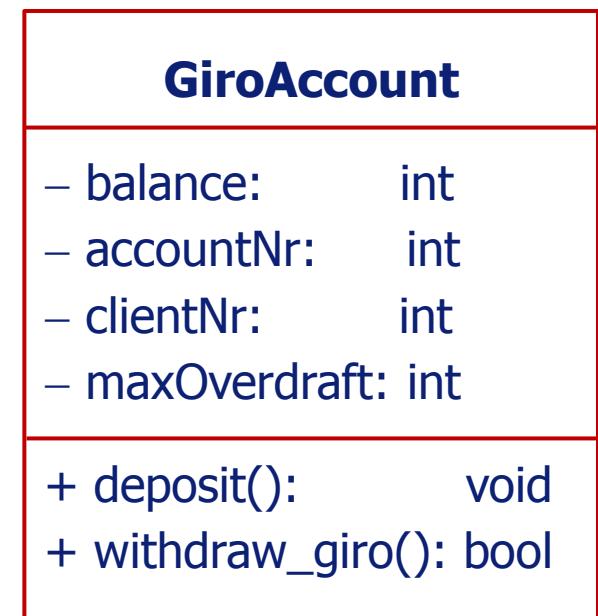
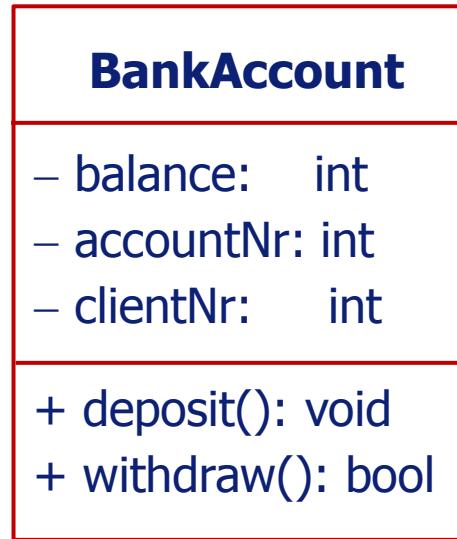
- hat alle Elemente von BankAccount und ggf. weitere Elemente, z.B. **maxOverdraft**
- hat alle Methoden von BankAccount und ggf. weitere Methoden
- einige Methoden von GiroAccount haben identische Funktion wie die von BankAccount
- einige Methoden haben modifizierte Funktionalität

Vererbung: Einführung

Betrachte Klasse **BankAccount**:

```
class BankAccount
{
    int balance;
    int accountNr;
    int clientNr;
    ...
public:
    BankAccount( ... ) { ... }
    ...
    void deposit (int amount)
    { balance += amount; }

    bool withdraw(int amount)
    { balance -= amount;
        return true; }
};
```



Quelle: Microconsult

Möglichkeit 1: Realisierung als Komposition

```
class BankAccount
{
    int balance;
    int accountNr;
    int clientNr;
    ...
public:
    BankAccount( ... ) { ... }
    ...
    void deposit (int amount)
        { balance += amount; }

    bool withdraw(int amount)
        { balance -= amount;
          return true; }
};
```

```
class GiroAccount
{
    BankAccount a;
    int maxOverdraft;
    ...
public:
    GiroAccount( ... ) { ... }

    ...
    void deposit (int amount)
        { a.deposit(amount); }

    bool withdraw(int amount) {
        if (a.getBalance()
            + maxOverdraft >= amount)
            {return a.withdraw(amount); }
        return false; }
};
```

Vererbung

Möglichkeit 1: Realisierung als Komposition

```
class BankAccount
{
    int balance;
    int accountNr;
    int clientNr;
    ...
public:
    BankAccount( ... ) { ... }
    ...
    void deposit (int amount)
        { balance += amount; }

    bool withdraw(int amount)
        { balance -= amount;
        return true; }
};
```

```
class GiroAccount
{
    BankAccount a;
    int maxOverdraft;
    ...
public:
    GiroAccount( ... ) { ... }
    ...
    void deposit (int amount)
        { a.deposit(amount); }

    bool withdraw(int amount)
        { if (a.getBalance() + maxOverdraft >= amount)
            { return a.withdraw(amount); }
        return false; }
};
```

BankAccount ist Teil von GiroAccount

Zusätzliches Element

Funktion übernommen

Zugriffsfkt.

Funktion verändert

Vererbung

Möglichkeit 1: Realisierung als Komposition



Damit **GiroAccount** verwendet werden kann wie **BankAccount**:

- Implementiere Methoden von **BankAccount** auch in **GiroAccount**
 - Einfacher Aufruf der Methode des eingebetteten Objekts (**deposit**)
 - Aufruf & Veränderung der Methode des eingeb. Objekts (**withdraw**)

Falls **BankAccount** um weitere Methoden **erweitert** wird:

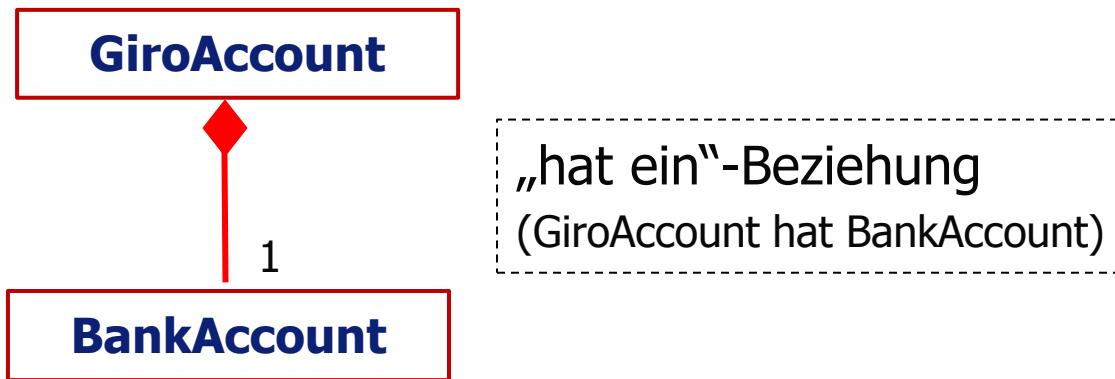
- Diese Methoden können in **GiroAccount** nur genutzt werden, wenn sie auch in **GiroAccount** implementiert werden
- Bei vielen anwendenden Klassen (z.B. **SavingsAccount**, ...) und vielen Methoden kann der Aufwand beträchtlich werden!

Einfacher: Attribute / Methoden von **BankAccount** direkt verfügbar in
GiroAccount ⇒ **Vererbung**



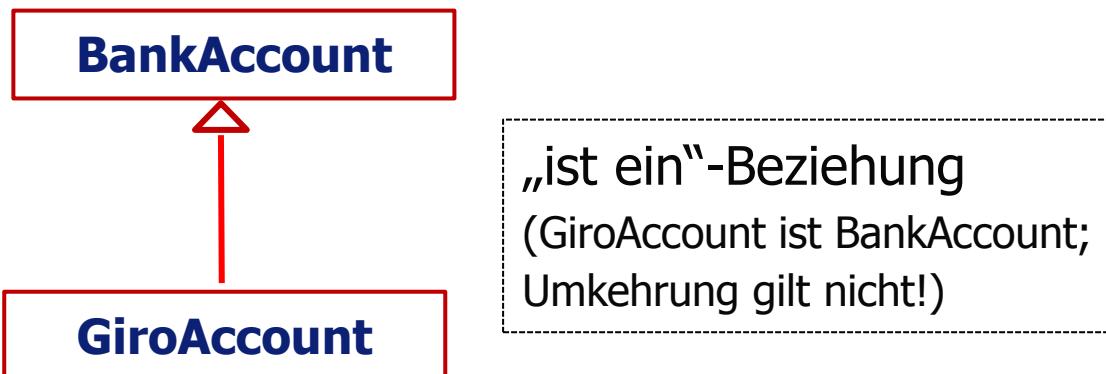
Komposition und Vererbung

Komposition:



Quelle: Microconsult

Vererbung:



Vererbung

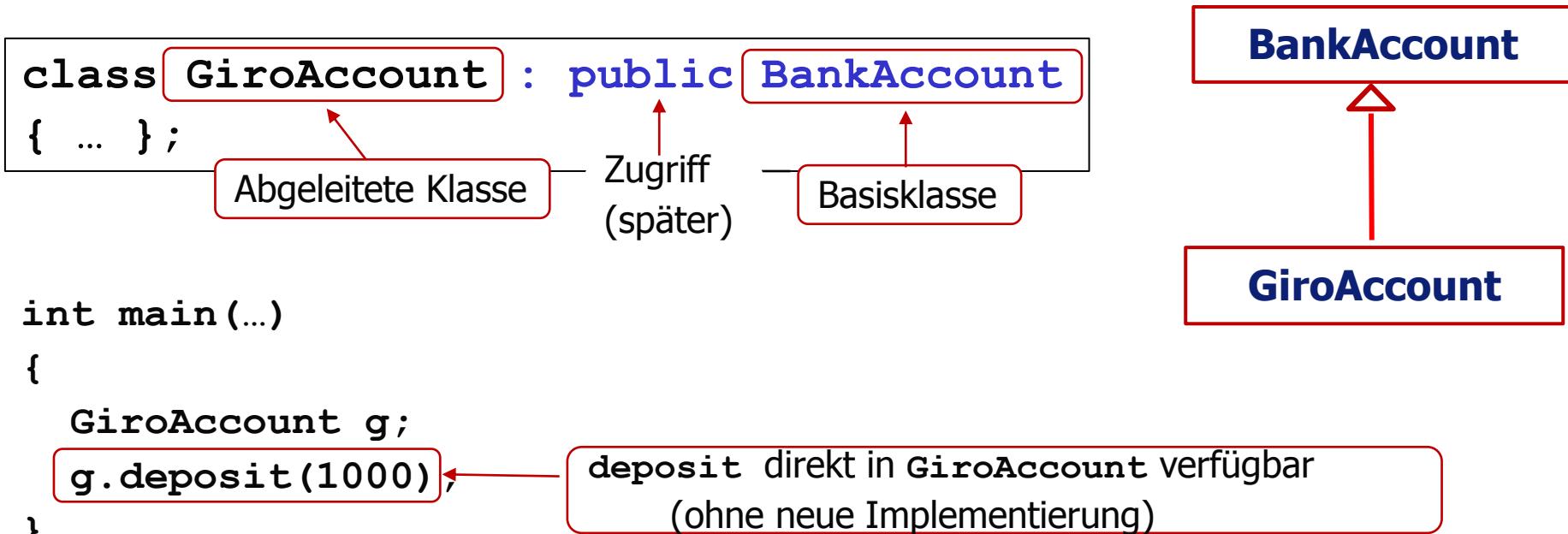
- Vererbung ist ein wesentliches Grundprinzip der OOP
- Vererbung stellt eine „ist eine (Art von)“ –Beziehung zwischen Klassen dar
 - Ein Girokonto ist eine spezielle Art eines Bankkontos
- Technisch gesehen bedeutet Vererbung lediglich, dass eine Klasse die in ihr enthaltenen Elemente an eine andere Klasse weitergibt (vererbt).
- Die Vererbungsbeziehung besteht zwischen



Vererbung

Beispiel GiroAccount und BankAccount:

- Ein Girokonto ist ein Bankkonto → alles, was ein Bankkonto ausmacht, ist auch im Girokonto „von Natur aus“ vorhanden
 - Die Eigenschaften des Bankkontos werden an das Girokonto vererbt
 - „**GiroAccount** erbt von **BankAccount**“
 - „**GiroAccount** wird von **BankAccount** abgeleitet“



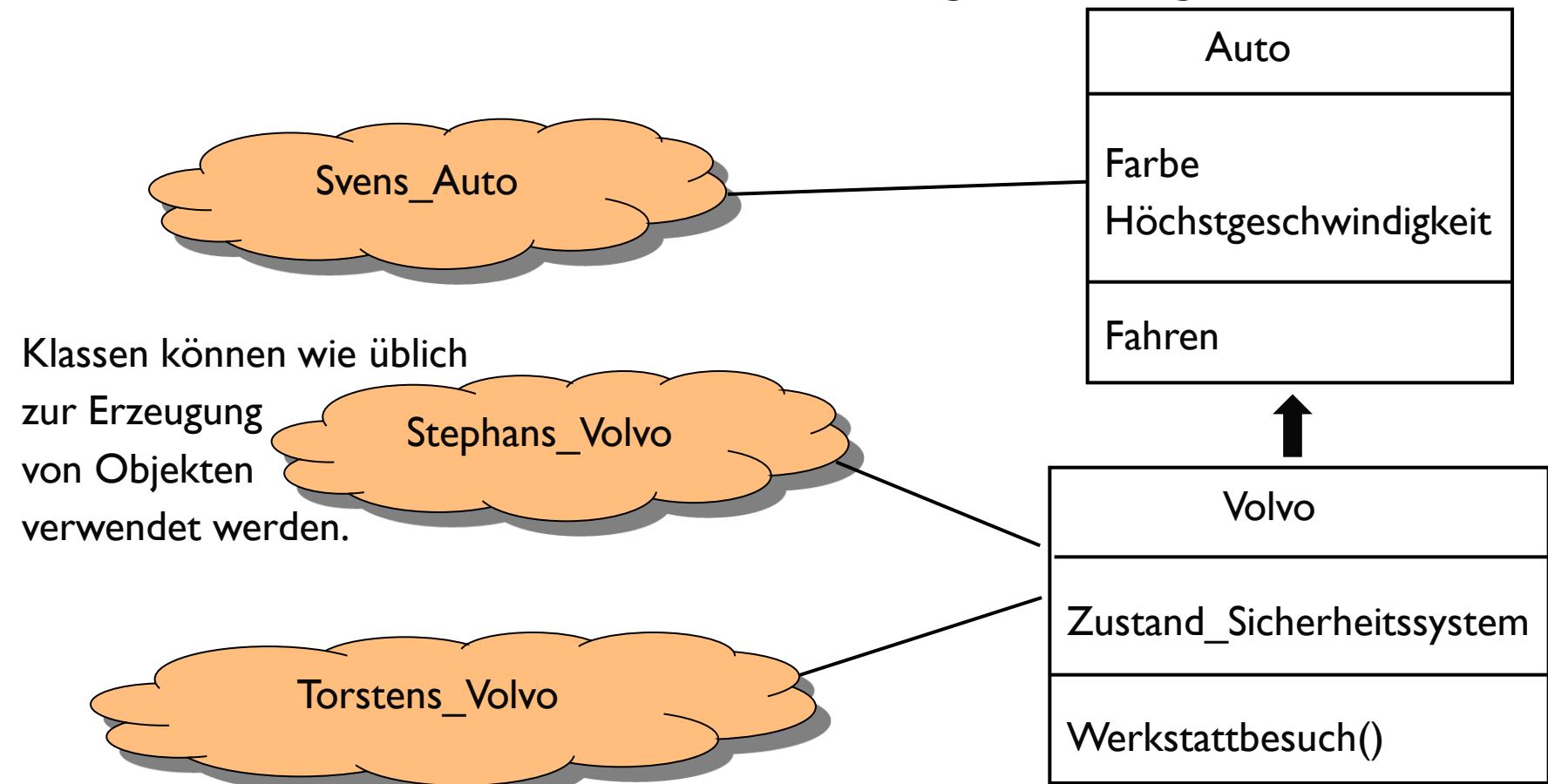
Vererbung

Wichtige Fakten:

- Basisklasse vererbt alle in ihr enthaltenen Elemente bis auf:
 - Kontruktoren, Destruktoren, Zuweisungsoperatoren, friend-Deklarationen
- Alle Attribute und Methoden der Basisklasse sind (abhängig vom Zugriffsschutz) Bestandteil der abgeleiteten Klasse
 - Ohne erneute Deklaration oder Implementierung
 - Zugriff aber abhängig vom Zugriffsschutz
- Auswahl der vererbten Elemente durch Programmierer ist nicht möglich.
→ Nicht zu verhindern, dass bestimmtes Element der Basisklasse vererbt wird
- Auf die Basisklasse hat die Vererbung **keine Auswirkung**
- Von einer Basisklasse können beliebig viele Klassen abgeleitet werden

Weitere Bemerkung:

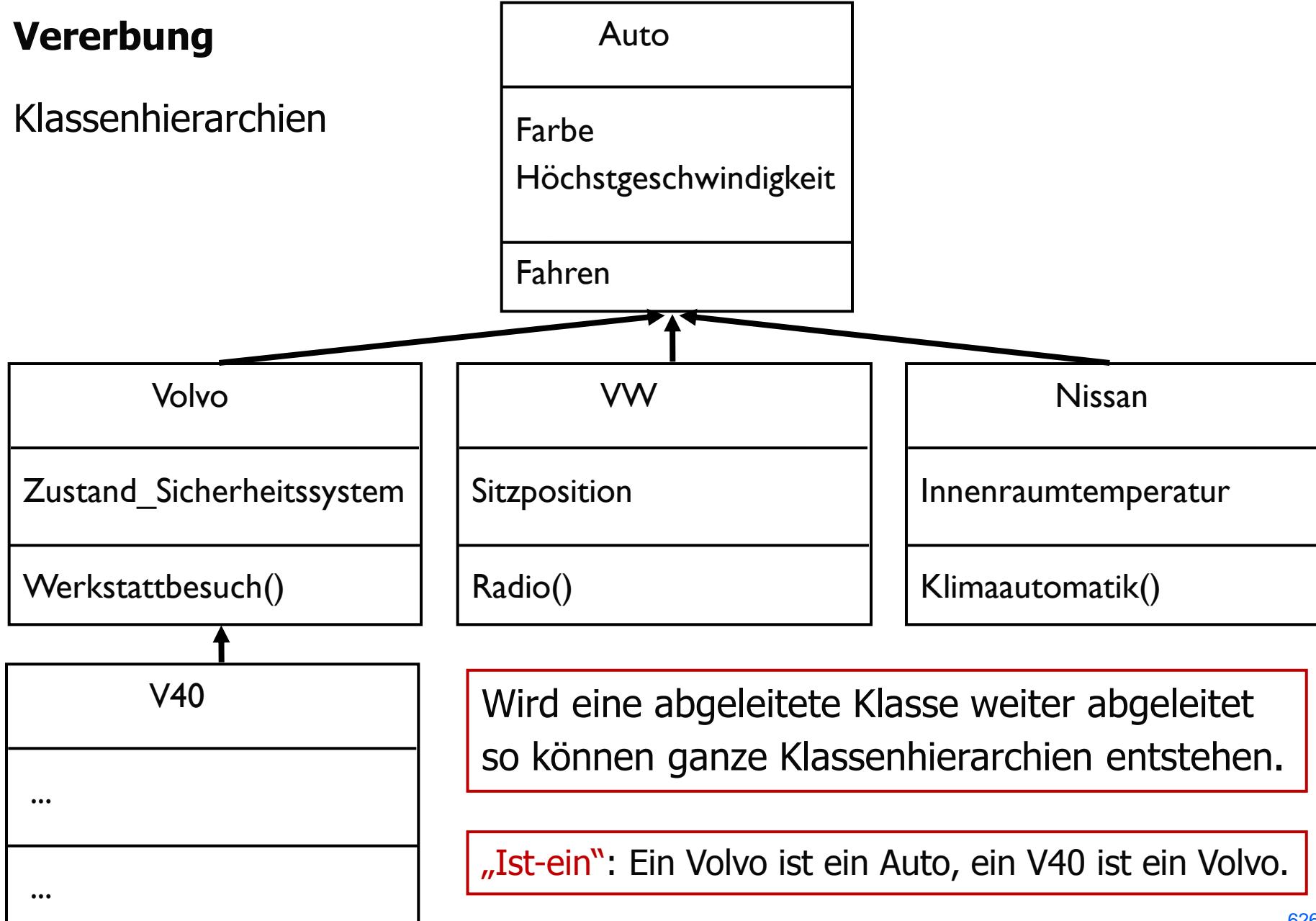
- Basisklasse und abgeleitete Klasse sind ***ganz normale Klassen***
- Bezeichnungen ("Basisklasse", "abgeleitete Klasse") beschreiben lediglich die Funktion der Klasse in der Vererbungsbeziehung



Vererbung

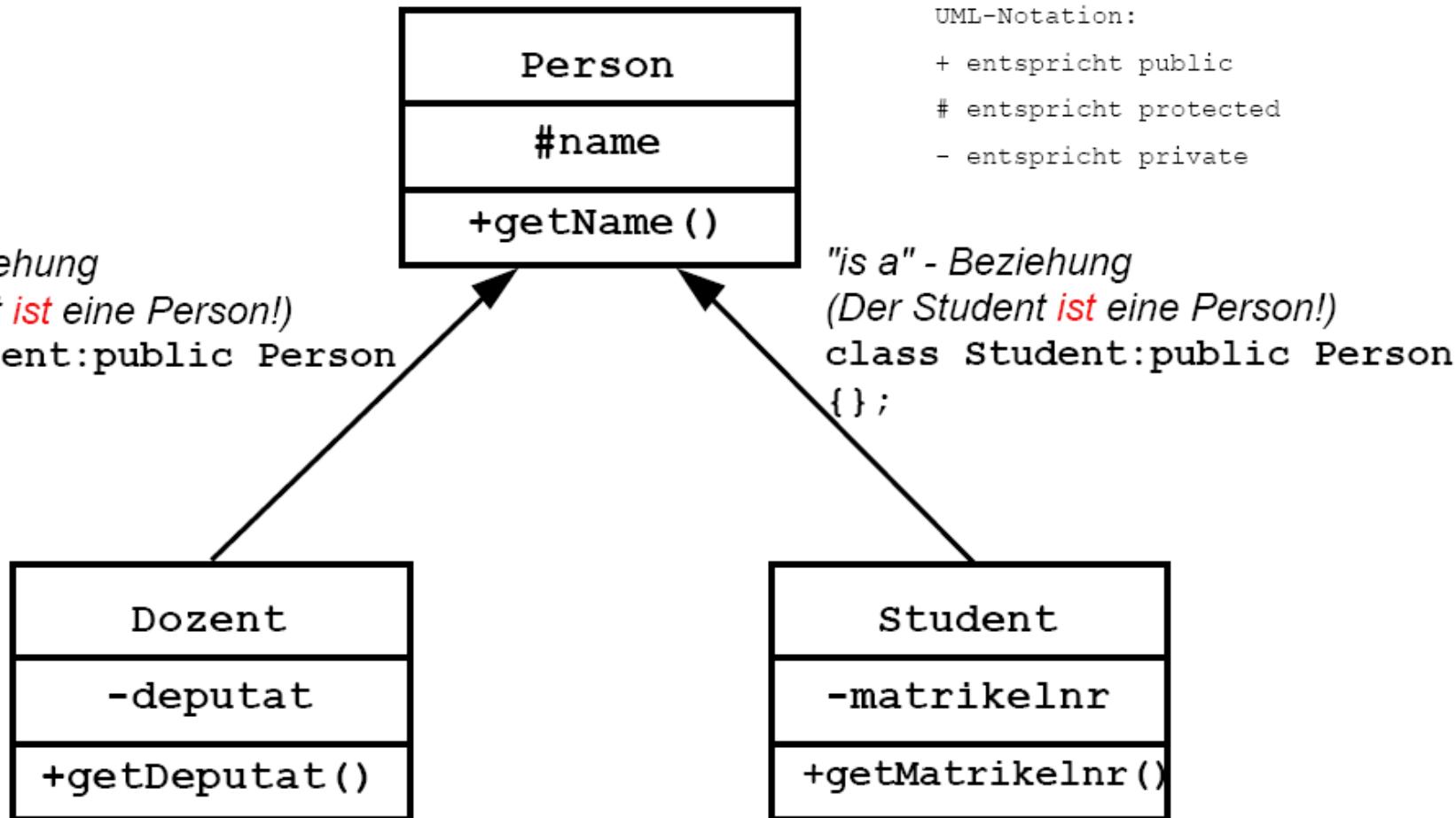
Vererbung

Klassenhierarchien



Vererbung

Vererbung: „ist ein“ - Beziehung



Vererbung

Vererbung auf Objektebene

d1 :: Dozent

"Harms"

7

Jedes Objekt der Klasse *Dozent* hat Zugriff auf die Elemente

name
getName ()
deputat
getDeputat ()

p1 :: Person

"Meier"

Jedes Objekt der Klasse *Person* hat Zugriff auf die Elemente

name

getName ()

protected

public

st1 :: Student

"Gross"

123456

Jedes Objekt der Klasse *Student* hat Zugriff auf die Elemente

name
getName ()
matrikelnr
getMatrikelnr ()

Quelle: HS Esslingen

628

Vererbung: Syntax

Erweiterung der einfachen Klassendefinition:

```
class Klassenname : BASISKLASSENLISTE  
{  
    // Elemente, die zu den geerbten hinzukommen sollen  
}
```

Basisklassenliste: Durch Komma getrennte Liste der Basisklassen

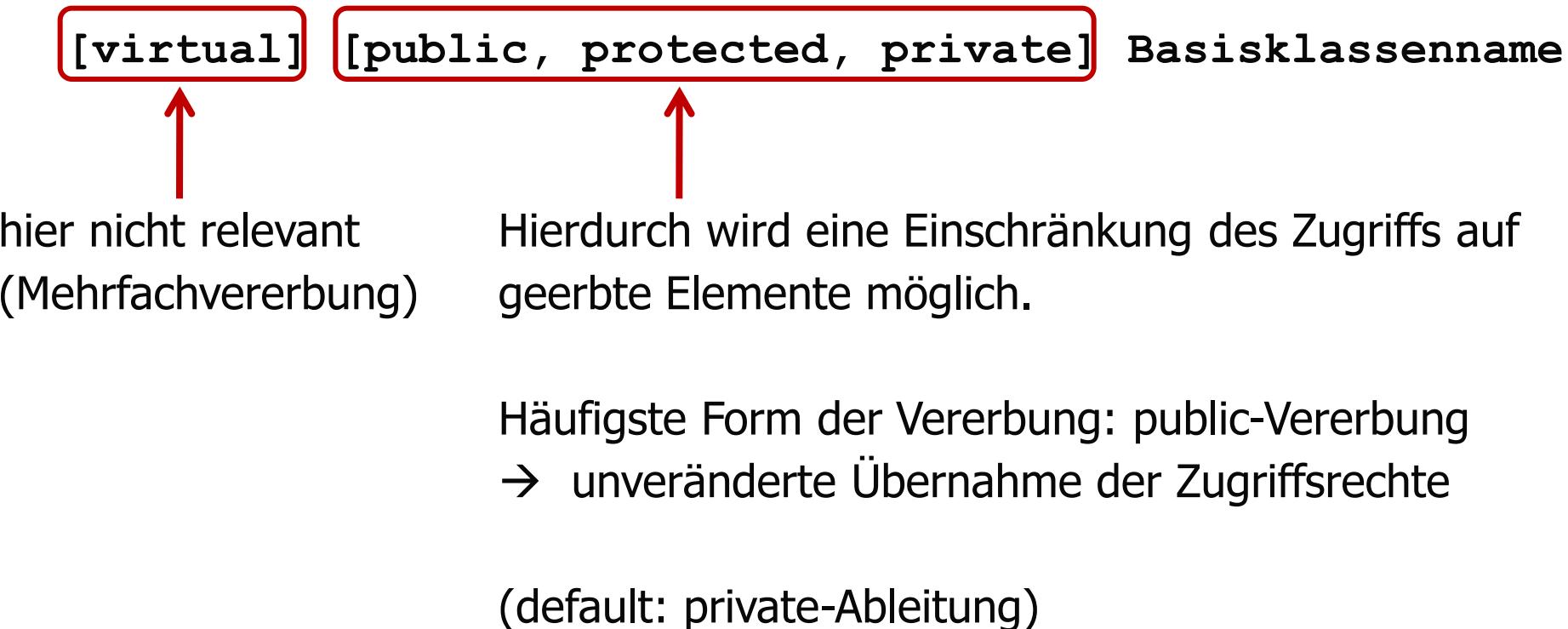
Warum Klassen?

- Mehrfachvererbung (siehe Kapitel "Grundlagen der OOP")
- Üblicherweise besteht die "Liste" aus nur einem Klassennamen

Vererbung: Syntax

Syntax eines Eintrags in der Basisklassenliste:

(Bemerkung: Elemente in [] sind optional)



Vererbung: Beispiel

```
class BankAccount {  
    int balance;  
    int accountNr;  
    int clientNr;  
  
public:  
    ...  
    int getBalance() {...}  
    void setAccountNr(int a) {...}  
    void setClientNr(int c) {...}  
    void deposit (int m) {...}  
    bool withdraw(int m) {...}  
};
```

```
class GiroAccount :  
    public BankAccount {  
        int maxOverdraft;  
  
public:  
    int getMaxOverdraft() {...}  
    void setMaxOverdraft(int d) {...}  
};
```

Falls erwünscht:
protected

- Klasse **GiroAccount** wird von **BankAccount** abgeleitet.
- **GiroAccount** erbt sämtliche Elemente von **BankAccount** (Attribute, Methoden)
- **GiroAccount** kann weitere Attribute und Methoden hinzufügen.
Hier: Attribut **maxOverdraft**, Methoden **get [set] maxOverdraft()**

Vererbung: Beispiel

```
class BankAccount {  
    int balance;  
    int accountNr;  
    int clientNr;  
  
public:  
    ...  
    int getBalance() {...}  
    void setAccountNr(int a) {...}  
    void setClientNr(int c) {...}  
    void deposit (int m) {...}  
    bool withdraw(int m) {...}  
};
```

```
class GiroAccount :  
    public BankAccount {  
        int maxOverdraft;  
  
public:  
    int getMaxOverdraft() {...}  
    void setMaxOverdraft(int d) {...}  
};
```

- In abgeleiteter Klasse: alle (public) Methoden der Basis- und abgeleiteten Klasse verfügbar
- Compiler sorgt für Aufruf der richtigen Methode

```
int main(...){  
    GiroAccount g;  
    g.setAccountNr(22); // BankAccount::setAccountNr(...)  
    g.setClientNr(123); // BankAccount::setClientNr(...)  
    g.setMaxOverdraft(500); // GiroAccount::setMaxOverdraft(...)  
    g.deposit(1000); // BankAccount::deposit()
```

Geht auch bei privatem `accountNr`, `clientNr`

Das Basisklassen-Unterobjekt

Objekte abgeleiteter Klassen enthalten ein (Unter-)Objekt ihrer Basisklasse



Dieses Unterobjekt enthält die von der Basisklasse geerbten Elemente.

Folge:

Vererbte Elemente werden zu einem Teil der abgeleiteten Klasse,
bleiben aber formal Elemente der Basisklasse.

Das Basisklassen-Unterobjekt

Objekte abgeleiteter Klassen enthalten ein (Unter-)Objekt ihrer Basisklasse



Begründung:

Konsequenter Einsatz der OOP-Philosophie auch bei Vererbungsbeziehungen

→ Vorteilhafte Konzepte der OOP bleiben erhalten

Beispiel: Kapselung, Geheimnisprinzip, Schutzmechanismen, Initialisierung, ...

Das Basisklassen-Unterobjekt

Vererbte Elemente werden zu einem Teil der abgeleiteten Klasse,
bleiben aber formal Elemente der Basisklasse.

Was bedeutet das für die Praxis?

- Grundsätzlich kann mit geerbten Elementen ebenso gearbeitet werden, wie mit solchen, die in der abgeleiteten Klasse direkt definiert wurden.
- In bestimmten Fällen muss die Basisklassenherkunft geerbter Elemente berücksichtigt werden.

Bsp.: Private Elemente der Basisklasse können nicht von Memberfunktionen verwendet werden, die in der abgeleiteten Klasse definiert wurden.



Basisklassen-Unterobjekt schottet seine privaten Variablen auch vor Zugriff durch abgeleitete Klassen ab.

Das Basisklassen-Unterobjekt

Vererbte Elemente werden zu einem Teil der abgeleiteten Klasse,
bleiben aber formal Elemente der Basisklasse.

Was bedeutet das für die Praxis?

- Grundsätzlich kann mit geerbten Elementen ebenso gearbeitet werden, wie mit solchen, die in der abgeleiteten Klasse direkt definiert wurden.
- In bestimmten Fällen muss die Basisklassenherkunft geerbter Elemente berücksichtigt werden.

Bsp.: Private Elemente der Basisklasse können nicht von Memberfunktionen verwendet werden, die in der abgeleiteten Klasse definiert wurden.

- Initialisierung der Basisklassen-Elemente durch Basisklassen-Konstruktor
- Objekte abgeleiteter Klassen können als Basisklassen-Objekte behandelt werden. Mehr dazu: später (Polymorphie)

Das Basisklassen-Unterobjekt

Zugriff auf die Elemente des Basisklassen-Unterobjekts

1. Zugriff in abgeleiteter Klasse auf geerbte **private**-Elemente nicht mögl.:
private schützt somit vor Zugriff
 - von außen
 - aus allen abgeleiteten Klassen
2. Zugriff in abgeleiteter Klasse auf geerbte **protected**-Elemente möglich:
 - **protected** schützt somit vor Zugriff von außen
 - **protected** ermöglicht aber Zugriff aus allen abgeleiteten Klassen
3. Zugriff in abgeleiteter Klasse auf geerbte **public**-Elemente möglich

Vererbung

Beispiel 1: Zugriff auf privates Element der Basisklasse

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

class basis
{
    int wert; // privates Element

public:
    basis() : wert(1) {}
    int get_wert() { return wert; }
    void set_wert(int n) { wert = n; }

};

class abgeleitet : public basis
{
public:
    void zuruecksetzen()
    {
        wert = 0; // ← Fehler!
    }
};

}
```

Fehler!

Zugriff auf privates Element der
Basisklasse ist nicht erlaubt

Vererbung

Beispiel 2: Zugriff auf protected-Element der Basisklasse

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

class basis
{
protected:
    int wert;

public:
    basis() : wert(1)      {}
    int get_wert()          { return wert; }
    void set_wert(int n)   { wert = n; }

};

class abgeleitet : public basis
{
public:
    void zuruecksetzen()
    {
        wert = 0; ←
    }
};
```

OK!

Zugriff auf protected-Element
der Basisklasse ist erlaubt

Vererbung

Beispiel 2: Zugriff auf protected-Element der Basisklasse

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

class basis
{
protected:
    int wert;

public:
    basis() : wert(1)      {}
    int get_wert()          { return wert; }
    void set_wert(int n)   { wert = n; }

};

class abgeleitet : public basis
{
public:
    void zuruecksetzen()
    {
        wert = 0; ←
    }
};
```

```
int main() {
    abgeleitet obj;
    cout << "Nach Initialisierung: "
         << obj.get_wert() << endl;
    obj.zuruecksetzen();
    cout << "Nach Zuruecksetzen: "
         << obj.get_wert() << endl;
    return 0; }
```

Ausgabe?

Nach Initialisierung: 1
Nach Zuruecksetzen: 0

OK!

Zugriff auf protected-Element
der Basisklasse ist erlaubt

Das Basisklassen-Unterobjekt

Instanzbildung

Konstruktoraufruf:

- Erzeugung eines Objekts einer abgeleiteten Klasse **x**:
Konstruktor der Basisklasse wird **vor** dem Konstruktor von **x** ausgeführt

Warum ist das sinnvoll?

→ Weil dadurch sichergestellt ist, dass vor der ersten Aktion in der abgeleiteten Klasse das Basisklassen-Unterobjekt korrekt initialisiert ist.

- Standardmechanismus: Aufruf des Standardkonstruktors
- Es besteht die Möglichkeit, d. Basisklassen-Konstruktor explizit aufzurufen

Wichtig, falls es keinen Standardkonstruktor gibt oder bei Überladung.

Vererbung

Beispiel: Konstruktoraufruf-Hierarchie

```
class abgeleitet : public basis
{
public:
    abgeleitet() {
        wert += 10;
    }

    void zuruecksetzen() {
        wert = 0;
    }
};

int main() {
    abgeleitet obj;

    cout << "Nach Initialisierung: " << obj.get_wert() << endl;
    system("PAUSE");
    return 0;
}
```

```
class basis
{
protected:
    int wert;

public:
    basis() : wert(1) {}
    int get_wert() { return wert; }
    void set_wert(int n) { wert = n; }
};
```

Ablauf:

1. Aufruf des Konstruktors der Basisklasse **basis()**
→ **wert = 1**
2. Aufruf des Konstruktors der abgel. Klasse **abgeleitet()**
→ **wert = 11**

Ausgabe:

```
Nach Initialisierung: 11
Drücken Sie eine beliebige Taste . . .
```

Vererbung

Beispiel: Aufruf eines bestimmten Basisklassen-Konstruktors

```
class abgeleitet : public basis
{
public:
    abgeleitet() : basis()
        wert += 10;
}

void zuruecksetzen() {
    wert = 0;
}

int main() {
    abgeleitet obj;

    cout << "Nach Initialisierung: " << obj.get_wert() << endl;

    system("PAUSE");
    return 0;
}
```

```
class basis
{
protected:
    int wert;

public:
    basis() : wert(1) {}
    basis(int w) : wert(w) {}
    int get_wert() { return wert; }
    void set_wert(int n) { wert = n; }
};
```

Aufruf des Standardkonstruktors

Redundant, weil dieser sowieso ausgeführt worden wäre.

Ausgabe:

Nach Initialisierung: 11

Vererbung

Beispiel: Aufruf eines bestimmten Basisklassen-Konstruktors

```
class abgeleitet : public basis
{
public:
    abgeleitet() : basis(0)
        wert += 10;
}

void zuruecksetzen()
{
    wert = 0;
}
};
```

```
int main()
{
    abgeleitet obj;
```

```
cout << "Nach Initialisierung: " << obj.get_wert() << endl;
```

```
system("PAUSE");
return 0;
}
```

```
class basis
{
protected:
    int wert;

public:
    basis() : wert(1)    {}
    basis(int w) : wert(w)  {}
    int get_wert()      { return wert; }
    void set_wert(int n) { wert = n; }
};
```

Aufruf des überladenen Konstruktors `basis(int w)`

Sorgt für Initialisierung mit dem Wert 0.

Ausgabe:

Nach Initialisierung: 10

Vererbung

Vererbung: Konstruktoren mit Parametern (Beispiel)

```
class Klasse2 : public Klasse1
{
protected:
    int zahl2;
public:
    Klasse2() : Klasse1(), zahl2(7) {}           Standardkonstruktor
    Klasse2(int z) : zahl2(z) {}                  (Aufruf redundant)
    Klasse2(int z1, int z2) : Klasse1(z1), zahl2(z2) {}
    void printK2() {cout << "\nzahl1 = " << zahl1;
                    cout << "\nzahl2 = " << zahl2;}
};
```

```
class Klasse1 {

protected:
    int zahl1;

public:
    Klasse1() : zahl1(1) {}
    Klasse1(int z) : zahl1(z) {}
    void setZahl1(int z)
    {
        zahl1 = z;
    }
    void printK1()
    {
        cout << "\nzahl1 = " << zahl1;
    }
};
```

Basisinitialisierung

Elementinitialisierung

Vererbung

Vererbung: Konstruktoren mit Parametern (Beispiel)

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

int main()
{
    Klasse1 o1;          // Standardkonstruktor K1
    Klasse1 o2(5);      // überladener Konstr. K1
    Klasse2 o3;          // Standardkonstruktor K2
    Klasse2 o4(8);      // überl. Konstr. K2, 1 Param.
    Klasse2 o5(2,3);    // überl. Konstr. K2, 2 Param.

    o1.printK1();
    o2.printK1();
    cout << endl;
    o3.printK2();
    o4.printK2();
    o5.printK2();

    system("PAUSE");
    return 0;
}
```

Vererbung

Vererbung: Konstruktoren mit Parametern (Beispiel)

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

int main()
{
    Klasse1 o1;
    Klasse1 o2(5);
    Klasse2 o3;
    Klasse2 o4(8);
    Klasse2 o5(2,3);

    o1.printK1();
    o2.printK1();
    cout << endl;
    o3.printK2();
    o4.printK2();
    o5.printK2();

    system("PAUSE");
    return 0;
}
```

```
class Klasse1 {
    ...
public:
    Klasse1() : zahl1(1) {}
    Klasse1(int z) : zahl1(z) {}
};
```

```
class Klasse2 : public Klasse1 {
    ...
public:
    Klasse2() : Klasse1(), zahl2(7) {}
    Klasse2(int z) : zahl2(z) {}
    Klasse2(int z1, int z2)
        : Klasse1(z1), zahl2(z2) {}
};
```

Ausgabe?

```
zahl1 = 1
zahl1 = 5
zahl1 = 1
zahl2 = 7
zahl1 = 1
zahl2 = 8
zahl1 = 2
zahl2 = 3
```

Das Basisklassen-Unterobjekt

Instanzauflösung

Destruktor:

- Auflösung eines Objekts einer abgeleiteten Klasse **x**:
Destruktor von **x** wird **vor** dem Destruktor der Basisklasse ausgeführt
(umgekehrte Reihenfolge wie Konstruktorausführung)
- Destruktor einer abgeleiteten Klasse ruft nach Ausführung seines Methodenkörpers den Destruktor der Basisklasse implizit auf

Initialisierung eines bzw. Zuweisung an ein Basisklassenobjekt

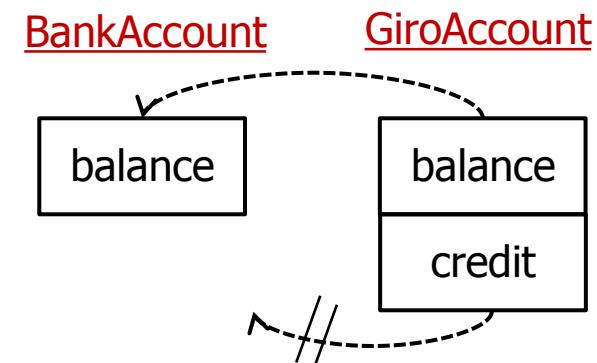
Beispiel:

```
class BankAccount {  
    int balance;  
    ...  
};
```

```
class GiroAccount : public BankAccount {  
    int credit;  
    ...  
};
```

```
int main(...) {  
    GiroAccount g;  
    BankAccount b = g;  
    // Initialisierung  
}
```

```
int main(...) {  
    BankAccount b;  
    GiroAccount g;  
    b = g; // Zuweisung  
}
```



- Objekt der abgeleiteten Klasse ist auch Objekt der Basisklasse
- Basisklassenobjekt kann mit abgeleitetem Objekt initialisiert werden
- Analog: abgeleitetes Objekt kann Basisklassenobjekt zugewiesen werden
- **Aber: Nur der Teil wird kopiert, der dem Basisklassenanteil entspricht!**

Initialisierung eines bzw. Zuweisung an ein abgeleitetes Objekt

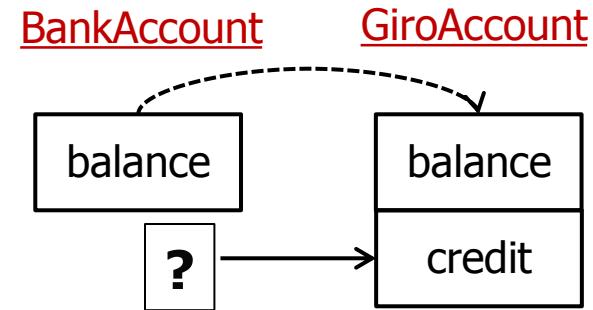
Beispiel:

```
class BankAccount {  
    int balance;  
    ...  
};
```

```
class GiroAccount : public BankAccount {  
    int credit;  
    ...  
};
```

```
int main(...) {  
    BankAccount b;  
    GiroAccount g = b;  
    // Fehler  
}
```

```
int main(...) {  
    BankAccount b;  
    GiroAccount g;  
    g = b; // Fehler  
}
```



- Umkehrung gilt nicht!
- Zuweisung / Initialisierung eines abgeleiteten Objektes durch Basisklassenobjekt nicht möglich, da sonst Teile undefiniert wären
- (Mögliche Lösung: geeigneten Kopierkonstruktor / Zuweisungsoperator definieren)

Vererbung: Auf welche Elemente greifen Methoden zu?

- Methode der Basisklasse: Zugriff nur auf Datenelemente der Basisklasse
- Methode der abgeleiteten Klasse: Zugriff auf
 - Datenelemente, die in der abgeleiteten Klasse definiert sind
 - Datenelemente, die in der Basisklasse definiert sind, **falls der Zugriffsschutz dies erlaubt** (siehe später)
- Wird von einer Methode einer abgeleiteten Klasse ein Name in der eigenen Klasse nicht gefunden, wird in der Basisklasse gesucht
- Auflösung von Namenskonflikten: Scope-Operator ::

Verdecken, Überschreiben, Überladen

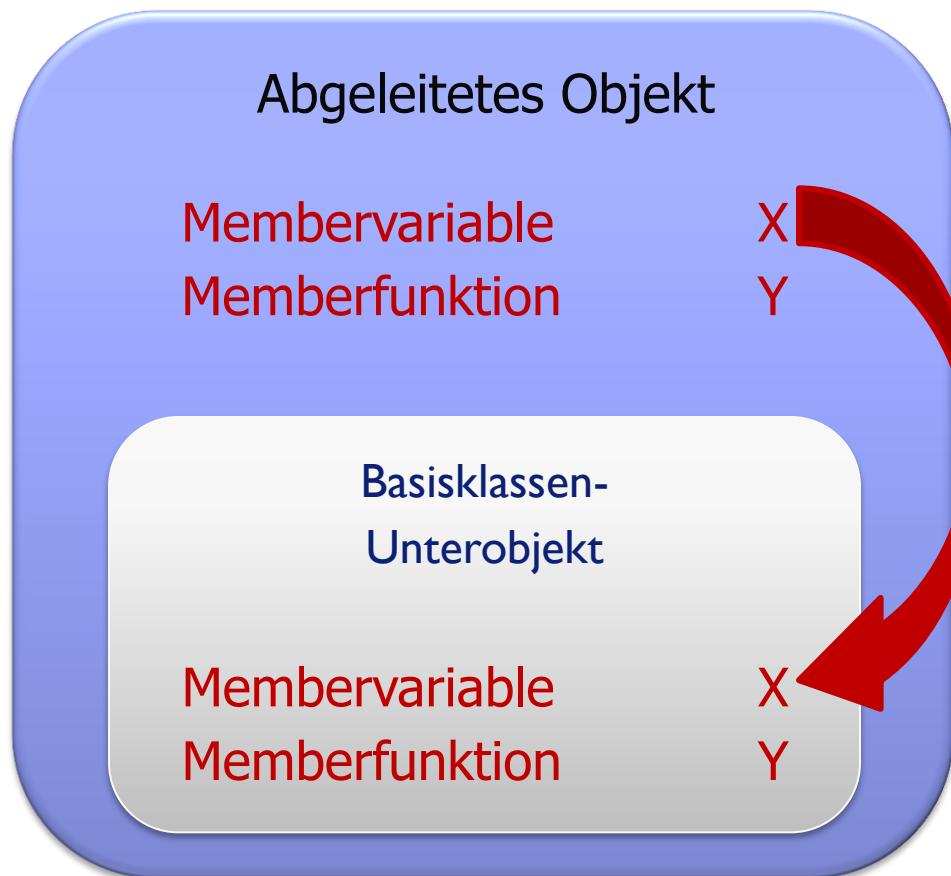
- Mechanismus der Vererbung reicht Attribute und Methoden einer Basisklasse an eine abgeleitete Klasse weiter
- In der Regel werden die geerbten Elemente um weitere Attribute und Methoden ergänzt.
- Zusätzlich zum Hinzufügen neuer Elemente können die geerbten Elemente *verändert* werden.

Je nach *Art* der Veränderung unterscheidet man:

- Verdeckung
- Überschreibung
- Überladung

Verdecken, Überschreiben, Überladen

Verdeckung



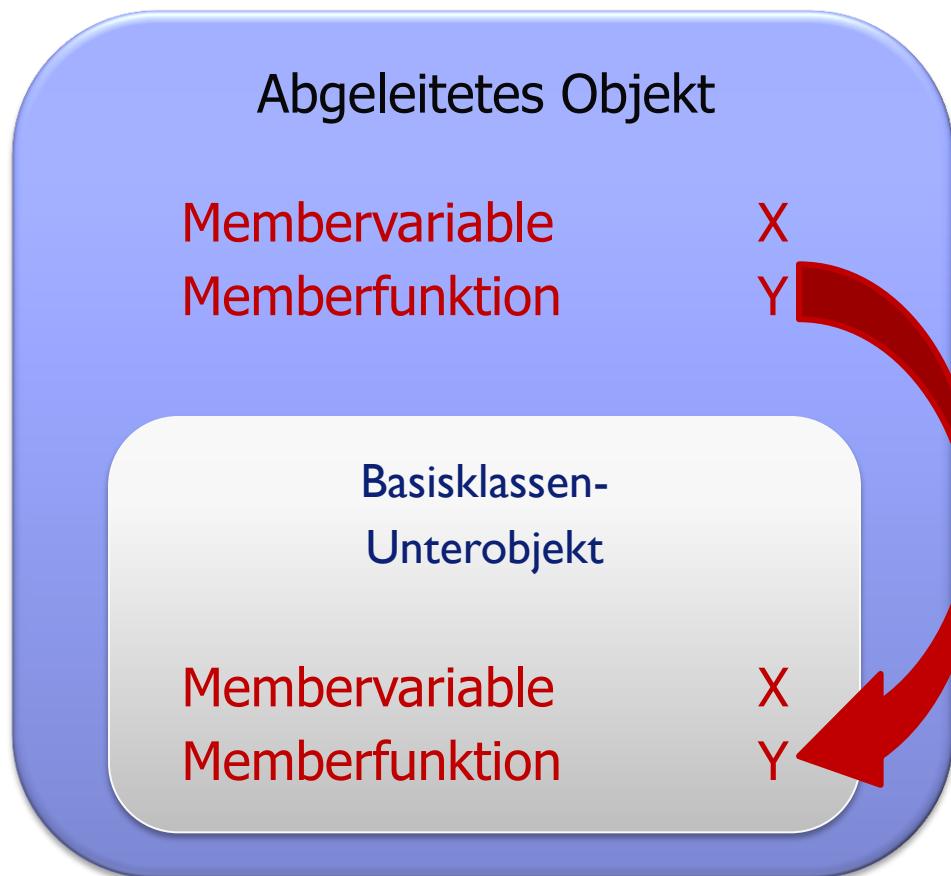
X der abgeleiteten Klasse **verdeckt**
X der Basisklasse

Situation: Eine Membervariable oder –funktion einer abgeleiteten Klasse hat denselben Namen wie ein entsprechendes Member der Basisklasse.

Folge: Das betreffende Member der Basisklasse ist in der abgeleiteten Klasse nicht direkt über den Namen ansprechbar – es ist **verdeckt**.

Verdecken, Überschreiben, Überladen

Verdeckung



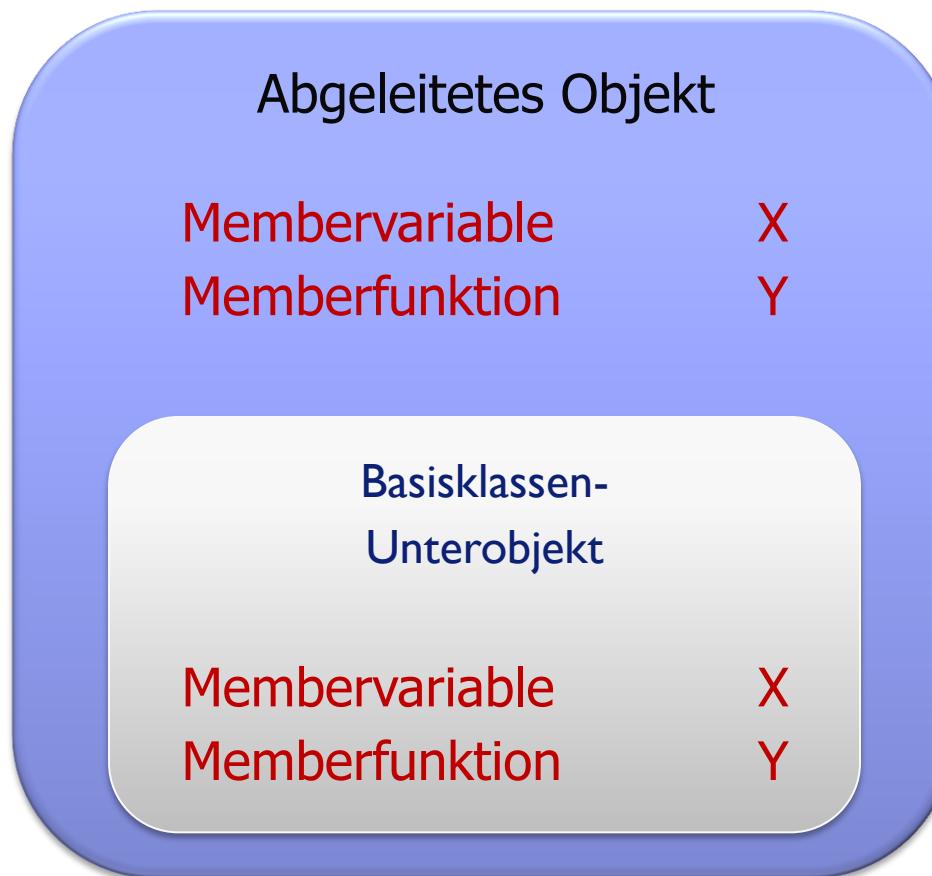
Y der abgeleiteten Klasse **verdeckt**
Y der Basisklasse

Situation: Eine Membervariable oder –funktion einer abgeleiteten Klasse hat denselben Namen wie ein entsprechendes Member der Basisklasse.

Folge: Das betreffende Member der Basisklasse ist in der abgeleiteten Klasse nicht direkt über den Namen ansprechbar – es ist **verdeckt**.

Verdecken, Überschreiben, Überladen

Verdeckung



Trotz der Verdeckung sind die Elemente im Basisklassen-Unterobjekt noch vorhanden
→ können mit Hilfe des *vollqualifizierten Namens* (Klassenname::X) verwendet werden
→ Beispiel auf der nächsten Seite

Beispiel:

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

class basis {
protected:
    int wert;
public:
    void print() {cout << wert;}
};

class abgeleitet : public basis
{
protected:
    int wert;
public:
    void zuruecksetzen()
    {
        wert = 1;
        basis::wert = 2;
    }
    void print() { cout << wert; } // verdeckt print() in Basisklasse
};
```

Membervariable `wert` in Klasse `abgeleitet` verdeckt die Membervariable `wert` in der Klasse `basis`.

Folge: Bei Verwendung von `wert` in `abgeleitet` wird die Variable der abgeleiteten Klasse verwendet – nicht die der Basisklasse.

Die Variable `wert` der Basisklasse ist jedoch noch vorhanden und kann mit Hilfe des vollqualifizierten Namens `basis::wert` verwendet werden.

Vererbung

Beispiel:

// Fortsetzung

int main()

{

 abgeleitet obj;

 obj.zuruecksetzen();

 cout << "abgeleitete Klasse: Wert = ";

 obj.print(); // ruft print() in abgeleiteter Klasse auf

 cout << endl;

 cout << "Basisklasse: Wert = ";

 obj.basis::print(); // ruft print() in Basisklasse auf

 cout << endl;

 system("PAUSE");

 return 0;

```
class basis {  
protected:  
    int wert;  
public:  
    void print()  
    {cout << wert;}  
};
```

```
class abgeleitet : public basis  
{  
    int wert;  
public:  
    void zuruecksetzen() {  
        wert = 1;  
        basis::wert = 2;  
    }  
    void print() { cout << wert; }  
};
```

Ausgabe?

abgeleitete Klasse: Wert = 1

Basisklasse: Wert = 2

Drücken Sie eine beliebige Taste . . .

Verdecken, Überschreiben, Überladen

Überladung

Geerbte Memberfunkt. können in abgeleiteten Klassen **nicht** überladen werden.

Grund:

Überladung ist in C++ an gemeinsamen Gültigkeitsbereich gebunden.

Basisklasse und abgeleitete Klasse bilden jeweils **eigenen Gültigkeitsbereich!**

Was passiert beim Versuch, eine geerbte Memberfunktion zu überladen?

- (d.h. in abgeleiteter Klasse: gleicher Name, geänderte Parameterliste)
- Erste überladene Version in abgeleiteter Klasse **verdeckt** die geerbte Fkt.
- Weitere überladene Versionen in abgeleiteter Klasse überladen nur die eine Version aus der abgeleiteten Klasse (und nicht die aus der Basisklasse).

Verdecken, Überschreiben, Überladen

Überschreibung

Falls Basisklassenimplementierung einer Methode nicht adäquat für abgeleitete Klasse: **Überschreibung**

- Methode behält Signatur bei (Name und Parameterliste)
- Erhält aber einen eigenen Anweisungsteil

Bsp.:

```
class basis
{
    int wert;

public:
    void print()
    {
        cout << wert;
    }
};
```

```
class abgeleitet : public basis
{
    int wert;

public:
    void print()
    {
        cout << wert << endl;
        cout << basis::wert << endl;
    }
};
```

überschreibt
basis::print()

Verdecken, Überschreiben, Überladen

Unterdrückung von Basisfunktionen durch Überschreibung

Falls Basisklassenmethode in abgel. Klasse nicht mehr verfügbar sein soll:

- Überschreibung der Funktion in abgeleiteter Klasse
- Deklaration als „private“

Bsp.:

```
class BankAccount  
{  
    ...  
};
```

```
class GiroAccount : public BankAccount  
{  
    int maxOverdraft;  
  
public:  
    void setMaxOverdraft(int draft)  
    { maxOverdraft = draft; }  
};
```

```
class PocketMoneyAccount : public GiroAccount  
{  
private:  
    void setMaxOverdraft(int draft) {}  
};
```

`setMaxOverdraft()`
über Objekt vom Typ
`PocketMoneyAccount`
nicht mehr zugreifbar

Vererbung

Weiteres Beispiel (a):

```
class Basis
{
protected:
    int x;

public:
    Basis(int x = 1) : x(x) {}
    void mal(int f)
    { cout << "Basis *" << endl; x*=f; }
    void print()
    { cout << "Basis: x = " << x << endl; }
};
```

```
class Abgeleitet : public Basis
{
    int y;

public:
    Abgeleitet(int x = 2, int y = 3) : Basis(x), y(y) {}

    void print() { cout << "Abgeleitet: x = " << x << ", y = " << y << endl; }
};
```

```
int main(int argc, char* argv[])
{
    Abgeleitet a;
    a.print();
    a.mal(2);
    a.print();
    return 0;
}
```

mal aus Basis-
klasse geerbt

Ausgabe

Abgeleitet: x = 2, y = 3
Basis *
Abgeleitet: x = 4, y = 3

Weiteres Beispiel (b):

```
class Basis
{
protected:
    int x;

public:
    Basis(int x = 1) : x(x) {}
    void mal(int f)
    { cout << "Basis *" << endl; x*=f; }
    void print()
    { cout << "Basis: x = " << x << endl; }
};
```

```
class Abgeleitet : public Basis
{
    int y;

public:
    Abgeleitet(int x = 2, int y = 3) : Basis(x), y(y) {}
    void mal(int f) { cout << "Abgeleitet *" << endl; x*=f; y*=f; }

    void print() { cout << "Abgeleitet: x = " << x << ", y = " << y << endl; }
};
```

```
int main(int argc, char* argv[])
{
    Abgeleitet a;
    a.print();
    a.mal(2);
    a.print();
    return 0;
}
```

mal in Abgeleitet
überschrieben

Ausgabe

Abgeleitet: x = 2, y = 3
Abgeleitet *
Abgeleitet: x = 4, y = 6

Vererbung

Weiteres Beispiel (c):

```
class Basis
{
protected:
    int x;

public:
    Basis(int x = 1) : x(x) {}

    void mal(int f)
    { cout << "Basis *" << endl; x*=f; }

    void print()
    { cout << "Basis: x = " << x << endl; }
};
```

```
class Abgeleitet : public Basis
{
    int y;

public:
    Abgeleitet(int x = 2, int y = 3) : Basis(x), y(y) {}

    void mal(int f, int g) { cout << "Abgeleitet * (2)" << endl; x*=f; y*=g; }

    void print() { cout << "Abgeleitet: x = " << x << ", y = " << y << endl; }
};
```

```
int main(int argc, char* argv[])
{
    Abgeleitet a;
    a.print();
    a.mal(2);
    a.print();

    return 0;
}
```

mal(int f)
aus Basisklasse
verdeckt durch
überladene Fkt.

Aufruf
a.Basis::mal(2);
ist möglich

Fehler: No matching function
for call to ,Abgeleitet::mal(int)'

Vererbung

Weiteres Beispiel (d):

```
class Basis
{
protected:
    int x;

public:
    Basis(int x = 1) : x(x) {}

    void mal(int f)
    { cout << "Basis *" << endl; x*=f; }

    void print()
    { cout << "Basis: x = " << x << endl; }
};
```

```
class Abgeleitet : public Basis
{
    int y;

public:
    Abgeleitet(int x = 2, int y = 3) : Basis(x), y(y) {}

    void mal(int f) { cout << "Abgeleitet *" << endl; x*=f; y*=f; }

    void mal(int f, int g) { cout << "Abgeleitet * (2)" << endl; x*=f; y*=g; }

    void print() { cout << "Abgeleitet: x = " << x << ", y = " << y << endl; }
};
```

```
int main(int argc, char* argv[])
{
    Abgeleitet a;
    a.print();
    a.mal(2);
    a.print();
    a.mal(2,3);
    a.print();
    return 0;
}
```

mal aus
Abgeleitet

mal aus
Abgeleitet
(überladen)

Ausgabe

Abgeleitet: x = 2, y = 3
Abgeleitet *
Abgeleitet: x = 4, y = 6
Abgeleitet * (2)
Abgeleitet: x = 8, y = 18

Vererbung

Weiteres Beispiel (e):

```
class Basis
{
protected:
    int x;

public:
    Basis(int x = 1) : x(x) {}

    void mal(int f)
    { cout << "Basis *" << endl; x*=f; }

    void print()
    { cout << "Basis: x = " << x << endl; }
};
```

```
class Abgeleitet : public Basis
{
    int y;

public:
    Abgeleitet(int x = 2, int y = 3) : Basis(x), y(y) {}

    void mal(int f, int g = 2){ cout << "Abgeleitet * (2)" << endl; x*=f; y*=g; }

    void print() { cout << "Abgeleitet: x = " << x << ", y = " << y << endl; }
};
```

```
int main(int argc, char* argv[])
{
    Abgeleitet a;
    a.print();
    a.mal(2);
    a.print();
    a.mal(2,3);
    a.print();
    return 0;
}
```

mal aus
Abgeleitet

mal aus
Abgeleitet
(überladen)

Ausgabe

Abgeleitet: x = 2, y = 3
Abgeleitet * (2)
Abgeleitet: x = 4, y = 6
Abgeleitet * (2)
Abgeleitet: x = 8, y = 18

Vorgabeargument

Zugriffsmodifizierer

- Elemente erben das Zugriffsrecht (Schutzattribut) aus der Basisklasse
- Schutz eines Basisklassenelementes kann in abgeleiteter Klasse **modifiziert** werden: **Zugriffsmodifizierer `public`, `protected` oder `private`**

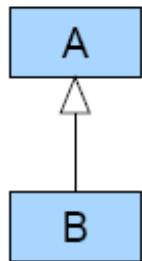
```
class abgeleitet : [public, protected, private] basis  
{ ... }
```

Mit **protected** oder **private** wird eine **Einschränkung** des Zugriffs auf geerbte Elemente möglich.
Default (falls keine Angabe): **private**

- Beispiele:

```
class abgeleitet : public basis {...};      // häufigste Form  
class abgeleitet : protected basis {...};  
class abgeleitet : private basis {...};  
class abgeleitet : basis {...}                // private Vererbung
```

Zugriffsmodifizierer und Zugriffsrechte



```

class A
{
public:
...
protected:
...
private:
...
};
  
```

```

class B : public A
{
public:
...
protected:
...
private:
...
};
  
```

Zusätzliche und overschriebene Methoden
und zusätzliche Daten.

Zugriffs-kontrolle	Zugriff nur für
private	eigene Klasse und Freunde (friend)
protected	eigene Klasse, Freunde und abgeleitete Klasse
public	alle

Daten sind meistens *private* oder *protected*.

Vererbungsart	Zugriff in Basisklasse	Zugriff in abgeleiteter Klasse
public	public protected private	public protected kein Zugriff
protected	public protected private	protected protected kein Zugriff
private	public protected private	private private kein Zugriff

- Ein bereits bestehender Zugriffsschutz kann also nur verstärkt werden!

Zugriffsmodifizierer, Zugriffsrechte:

Beispiel (1)

```
class Basis {  
    private: int x;  
    protected: int y;  
    public: int z;  
  
    Basis( int i, int j, int k )  
        { x=i; y=j; z=k; }  
    int getx(void) {return x;}  
    int gety(void) {return y;}  
    int getz(void) {return z;}  
};
```

```
int main( int argc, char* argv[] )  
{  
    Basis b1( 1, 2, 3 );  
  
    cout << b1.x << endl;  
    cout << b1.y << endl;  
    cout << b1.z << endl;  
    cout << b1.getx() << endl;  
    cout << b1.gety() << endl;  
    cout << b1.getz() << endl;  
  
    system("PAUSE");  
    return 0;  
}
```



Sind die Ausgaben möglich oder nicht?

Zugriffsmodifizierer, Zugriffsrechte:

Beispiel (1)

```
class Basis {  
    private: int x;  
    protected: int y;  
    public: int z;  
  
    Basis( int i, int j, int k )  
        { x=i; y=j; z=k; }  
    int getx(void) {return x;}  
    int gety(void) {return y;}  
    int getz(void) {return z;}  
};
```

```
int main( int argc, char* argv[] )  
{  
    Basis b1( 1, 2, 3 );  
  
    // cout << b1.x << endl; ←  
    // cout << b1.y << endl; ←  
    cout << b1.z << endl; // o.k.  
    cout << b1.getx() << endl; // o.k.  
    cout << b1.gety() << endl; // o.k.  
    cout << b1.getz() << endl; // o.k.  
  
    system("PAUSE");  
    return 0;  
}
```

Fehler: Zugriff von außen
auf privates Element

Fehler: Zugriff von außen
auf protected Element

Zugriffsmodifizierer, Zugriffsrechte:

Beispiel (1): public-Ableitung

```
class Ableitung : public Basis {  
public:  
    Ableitung( int i, int j, int k )  
        : Basis(i,j,k) {}  
    int getx(void) {return x;}  
    int gety(void) {return y;}  
    int getz(void) {return z;}  
};
```

```
class Basis {  
private:    int x;  
protected: int y;  
public:     int z;  
    Basis( int i, int j, int k )  
        { x=i; y=j; z=k; }  
    int getx(void) {return x;}  
    int gety(void) {return y;}  
    int getz(void) {return z;}  
};
```

Zugriffsmodifizierer, Zugriffsrechte:

Beispiel (1): public-Ableitung

```
class Ableitung : public Basis {
public:
    Ableitung( int i, int j, int k )
        : Basis(i,j,k) {}
    //    int getx(void) {return x;} ←
    int gety(void) {return y;}
    int getz(void) {return z;}
};
```

```
class Basis {
private:    int x;
protected: int y;
public:     int z;
Basis( int i, int j, int k )
    { x=i; y=j; z=k; }
int getx(void) {return x;}
int gety(void) {return y;}
int getz(void) {return z;}
};
```

Fehler: kein Zugriff aus abgeleiteter Klasse auf privates Element der Basisklasse

```
int main( int argc, char* argv[] ) {
Ableitung a1( 1, 2, 3 );
cout << a1.x << endl;
cout << a1.y << endl;
cout << a1.z << endl;
cout << a1.getx() << endl;
cout << a1.gety() << endl;
cout << a1.getz() << endl;
system("PAUSE");
return 0; }
```

Sind die Ausgaben möglich oder nicht?

Zugriffsmodifizierer, Zugriffsrechte:

Beispiel (1): public-Ableitung

```
class Ableitung : public Basis {
public:
    Ableitung( int i, int j, int k )
        : Basis(i,j,k) {}

    // int getx(void) {return x;}
    int gety(void) {return y;}
    int getz(void) {return z;}
};
```

```
class Basis {
private:    int x;
protected: int y;
public:     int z;

Basis( int i, int j, int k )
    { x=i; y=j; z=k; }

int getx(void) {return x;}
int gety(void) {return y;}
int getz(void) {return z;}
};
```

```
int main( int argc, char* argv[] ) {
Ableitung a1( 1, 2, 3 );
// cout << a1.x << endl;
// cout << a1.y << endl;
cout << a1.z << endl; // o.k.
cout << a1.getx() << endl;
cout << a1.gety() << endl; // o.k.
cout << a1.getz() << endl; // o.k.
system("PAUSE");
return 0; }
```

Fehler: Zugriff von außen über abgeleitete Klasse auf privates / protected Element

ACHTUNG: `getx()` aus abgeleiteter Klasse auskommentiert → `getx()` aus Basisklasse wird aufgerufen (o.k.)

Zugriffsmodifizierer, Zugriffsrechte:

Beispiel (1): protected-Ableitung

```
class Ableitung : protected Basis {
public:
    Ableitung( int i, int j, int k )
        : Basis(i,j,k) {}
//    int getx(void) {return x;} ←
    int gety(void) {return y;}
    int getz(void) {return z;}
};
```

```
class Basis {
private:    int x;
protected: int y;
public:     int z;
Basis( int i, int j, int k )
    { x=i; y=j; z=k; }
int getx(void) {return x;}
int gety(void) {return y;}
int getz(void) {return z;}
};
```

Fehler: kein Zugriff aus abgeleiteter Klasse auf privates Element der Basisklasse

```
int main( int argc, char* argv[] ) {
Ableitung a1( 1, 2, 3 );
cout << a1.x << endl;
cout << a1.y << endl;
cout << a1.z << endl;
cout << a1.getx() << endl;
cout << a1.gety() << endl;
cout << a1.getz() << endl;
system("PAUSE");
return 0; }
```

Sind die Ausgaben möglich oder nicht?

Zugriffsmodifizierer, Zugriffsrechte:

Beispiel (1): protected-Ableitung

```
class Ableitung : protected Basis {
public:
    Ableitung( int i, int j, int k )
        : Basis(i,j,k) {}

//    int getx(void) {return x;}
    int gety(void) {return y;}
    int getz(void) {return z;}
};
```

```
class Basis {
private:    int x;
protected: int y;
public:     int z;

Basis( int i, int j, int k )
    { x=i; y=j; z=k; }

int getx(void) {return x;}
int gety(void) {return y;}
int getz(void) {return z;}
};
```

```
int main( int argc, char* argv[] ) {
Ableitung a1( 1, 2, 3 );
//    cout << a1.x << endl;
//    cout << a1.y << endl;
//    cout << a1.z << endl;
//    cout << a1.getx() << endl;
cout << a1.gety() << endl; // o.k.
cout << a1.getz() << endl; // o.k.
system("PAUSE");
return 0; }
```

Fehler: Zugriff von außen über abgeleitete Klasse auf privates / protected Element

Fehler: public Basiselement in abgeleiteter Klasse **protected** → kein Zugriff von außen

Fehler: **getx()** aus Basiskl. in abgeleiteter Klasse **protected** → kein Zugriff von außen

gety(), getz() aus abgeleiteter Klasse!

Zugriffsmodifizierer, Zugriffsrechte:

Beispiel (1): private-Ableitung

```
class Ableitung : private Basis {
public:
    Ableitung( int i, int j, int k )
        : Basis(i,j,k) {}
    //    int getx(void) {return x;} ←
    int gety(void) {return y;}
    int getz(void) {return z;}
};
```

```
class Basis {
private:    int x;
protected: int y;
public:     int z;
Basis( int i, int j, int k )
    { x=i; y=j; z=k; }
int getx(void) {return x;}
int gety(void) {return y;}
int getz(void) {return z;}
};
```

Fehler: kein Zugriff aus abgeleiteter Klasse auf privates Element der Basisklasse

```
int main( int argc, char* argv[] ) {
Ableitung a1( 1, 2, 3 );
cout << a1.x << endl;
cout << a1.y << endl;
cout << a1.z << endl;
cout << a1.getx() << endl;
cout << a1.gety() << endl;
cout << a1.getz() << endl;
system("PAUSE");
return 0; }
```

Sind die Ausgaben möglich oder nicht?

Zugriffsmodifizierer, Zugriffsrechte:

Beispiel (1): private-Ableitung

```
class Ableitung : private Basis {
public:
    Ableitung( int i, int j, int k )
        : Basis(i,j,k) {}

    //    int getx(void) {return x;}
    int gety(void) {return y;}
    int getz(void) {return z;}
};
```

```
class Basis {
private:    int x;
protected: int y;
public:     int z;

    Basis( int i, int j, int k )
        { x=i; y=j; z=k; }

    int getx(void) {return x;}
    int gety(void) {return y;}
    int getz(void) {return z;}
};
```

```
int main( int argc, char* argv[] ) {
    Ableitung a1( 1, 2, 3 );
    //    cout << a1.x << endl;
    //    cout << a1.y << endl;
    //    cout << a1.z << endl;
    //    cout << a1.getx() << endl;
    cout << a1.gety() << endl; // o.k.
    cout << a1.getz() << endl; // o.k.
    system("PAUSE");
    return 0; }
```

Fehler: Zugriff von außen über abgeleitete Klasse auf **private** Elemente

Fehler: public Basiselement in abgeleiteter Klasse **private** → kein Zugriff von außen

Fehler: **getx()** aus Basiskl. in abgeleiteter Klasse **private** → kein Zugriff von außen

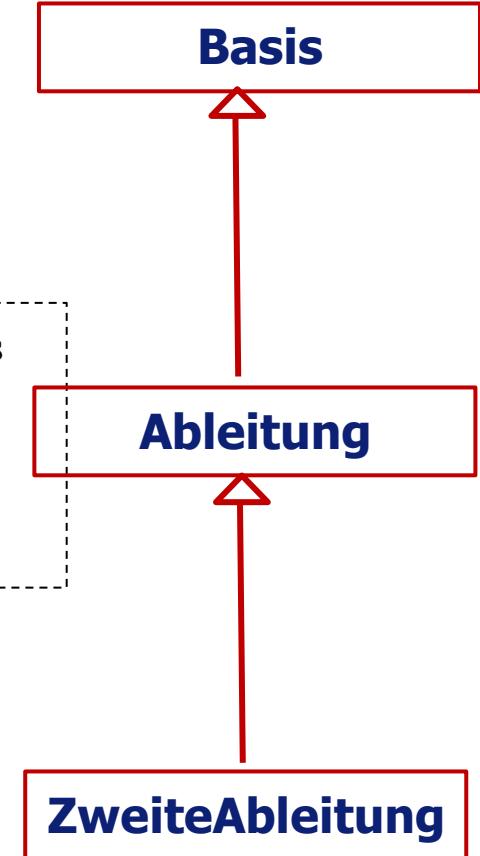
Zugriffsmodifizierer und Zugriffsrechte: Beispiel (2)

```
class Basis  
{  
...  
};
```

```
class Ableitung : <Zugriffsmodifizierer> Basis  
{  
...  
};
```

```
class ZweiteAbleitung : public Ableitung  
{  
...  
};
```

Vererbungsart für ZweiteAbleitung
im folgenden Beispiel immer **public**



Zugriff bei public-Ableitung (1)

```
class Ableitung : public Basis {  
public:  
    Ableitung( int i, int j, int k ) : Basis(i,j,k) {}  
//    int getx(void) {return x;} // Fehler, kein Zugriff auf priv. Element von Basis  
    int gety(void) {return y;} // o.k.; Zugriff auf protected Element von Basis  
    int getz(void) {return z;} // o.k.; Zugriff auf public Element von Basis  
};
```

```
class Basis {  
private:    int x;  
protected: int y;  
public:     int z;  
    Basis( int i, int j, int k )  
        { x=i; y=j; z=k; }  
    int getx(void) {return x;}  
    int gety(void) {return y;}  
    int getz(void) {return z;}  
};
```

```
class ZweiteAbleitung : public Ableitung {  
public:  
    ZweiteAbleitung( int i, int j, int k ) : Ableitung(i,j,k) {}  
//    int getx(void) {return x;} // Fehler, kein Zugriff auf priv. Element von Basis  
    int gety(void) {return y;} // o.k.; Zugriff auf protected Element von Basis  
    int getz(void) {return z;} // o.k.; Zugriff auf public Element von Basis  
};
```

Zugriff bei public-Ableitung (2)

```
class ZweiteAbleitung : public Ableitung {  
public:  
    ZweiteAbleitung( int i, int j, int k )  
        : Ableitung(i,j,k) {}  
    //    int getx(void) {return x;}  
    int gety(void) {return y;}  
    int getz(void) {return z;}  
};
```

```
int main( int argc, char* argv[] ) {  
    ZweiteAbleitung a2(1,2,3);  
    cout << a2.x << endl;  
    cout << a2.y << endl;  
    cout << a2.z << endl;  
    cout << a2.getx() << endl;  
    cout << a2.gety() << endl;  
    cout << a2.getz() << endl;  
    system("PAUSE");  
    return 0; }
```

Sind die Ausgaben möglich oder nicht?

Zugriff bei public-Ableitung (2)

```
class ZweiteAbleitung : public Ableitung {  
public:  
    ZweiteAbleitung( int i, int j, int k )  
        : Ableitung(i,j,k) {}  
    //    int getx(void) {return x;}  
    int gety(void) {return y;}  
    int getz(void) {return z;}  
};
```

```
int main( int argc, char* argv[] ) {  
    ZweiteAbleitung a2(1,2,3);  
    //    cout << a2.x << endl;  
    //    cout << a2.y << endl;  
    cout << a2.z << endl;      // o.k.  
    cout << a2.getx() << endl;  
    cout << a2.gety() << endl; // o.k.  
    cout << a2.getz() << endl; // o.k.  
    system("PAUSE");  
    return 0; }
```

Fehler: kein Zugriff auf privates / protected Element von Basis

ACHTUNG: `getx()` aus `ZweiteAbleitung` und `Ableitung` auskommentiert → `getx()` aus Basisklasse wird aufgerufen (o.k.)

Zugriff bei **protected**-Ableitung (1)

```
class Ableitung : protected Basis {  
public:  
    Ableitung( int i, int j, int k ) : Basis(i,j,k) {}  
//    int getx(void) {return x;} // Fehler, kein Zugriff auf priv. Element von Basis  
    int gety(void) {return y;}   // o.k.; protected Element von Basis hier protected  
    int getz(void) {return z;}   // o.k.; public Element von Basis hier protected  
};
```

```
class Basis {  
private:    int x;  
protected: int y;  
public:     int z;  
    Basis( int i, int j, int k )  
        { x=i; y=j; z=k; }  
    int getx(void) {return x;}  
    int gety(void) {return y;}  
    int getz(void) {return z;}}
```

```
class ZweiteAbleitung : public Ableitung {  
public:  
    ZweiteAbleitung( int i, int j, int k ) : Ableitung(i,j,k) {}  
//    int getx(void) {return x;} // Fehler, kein Zugriff auf priv. Element von Basis  
    int gety(void) {return y;}   // o.k.; protected Elem. von Basis bleibt protected  
    int getz(void) {return z;}   // o.k.; public Element von Basis bleibt protected  
};
```

Zugriff bei protected-Ableitung (2)

```
class ZweiteAbleitung : public Ableitung {  
public:  
    ZweiteAbleitung( int i, int j, int k ) : Ableitung(i,j,k) {}  
//    int getx(void) {return x;} // Fehler, kein Zugriff auf priv. Element von Basis  
    int gety(void) {return y;} // o.k.; protected Elem. von Basis bleibt protected  
    int getz(void) {return z;} // o.k.; public Element von Basis bleibt protected  
};
```

```
int main( int argc, char* argv[] ) {  
    ZweiteAbleitung a2(1,2,3);  
    cout << a2.x << endl;  
    cout << a2.y << endl;  
    cout << a2.z << endl;  
    cout << a2.getx() << endl;  
    cout << a2.gety() << endl;  
    cout << a2.getz() << endl;  
    system("PAUSE");  
    return 0; }
```



Sind die Ausgaben möglich oder nicht?

Zugriff bei protected-Ableitung (2)

```
class ZweiteAbleitung : public Ableitung {  
public:  
    ZweiteAbleitung( int i, int j, int k ) : Ableitung(i,j,k) {}  
//    int getx(void) {return x;} // Fehler, kein Zugriff auf priv. Element von Basis  
    int gety(void) {return y;} // o.k.; protected Elem. von Basis bleibt protected  
    int getz(void) {return z;} // o.k.; protected Elem. von Basis bleibt protected  
};
```

```
int main( int argc, char* argv[] ) {  
    ZweiteAbleitung a2(1,2,3);  
//    cout << a2.x << endl; ←  
//    cout << a2.y << endl; ←  
//    cout << a2.z << endl; ←  
//    cout << a2.getx() << endl; ←  
    cout << a2.gety() << endl; // o.k.  
    cout << a2.getz() << endl; // o.k.  
    system("PAUSE");  
    return 0; }
```

Fehler: kein Zugriff auf privates / protected Element von Basis

Fehler: Zugriff auf **z** über (Zweite)Ableitung **protected**

Fehler: Zugriff auf Basisklassen-**getx ()** über (Zweite)Ableitung **protected**

Zugriff bei **private**-Ableitung (1)

private-Ableitung: Zugriff auf Basiselemente
über Klasse Ableitung ist **private**
→ *kein Zugriff mehr über ZweiteAbleitung*

```
class Ableitung : private Basis {  
public:  
    Ableitung( int i, int j, int k ) : Basis(i,j,k) {}  
//    int getx(void) {return x;} // Fehler, kein Zugriff auf priv. Element von Basis  
    int gety(void) {return y;} // o.k.; protected Element von Basis hier private  
    int getz(void) {return z;} // o.k.; public Element von Basis hier private  
};  
  
class ZweiteAbleitung : public Ableitung {  
public:  
    ZweiteAbleitung( int i, int j, int k ) : Ableitung(i,j,k) {}  
//    int getx(void) {return x;} // Fehler, kein Zugriff auf priv. Element von Basis  
//    int gety(void) {return y;} // Fehler, kein Zugriff auf y über ZweiteAbleitung  
//    int getz(void) {return z;} // Fehler, kein Zugriff auf z über ZweiteAbleitung  
};
```

```
class Basis {  
private:    int x;  
protected: int y;  
public:     int z;  
    Basis( int i, int j, int k )  
        { x=i; y=j; z=k; }  
    int getx(void) {return x;}  
    int gety(void) {return y;}  
    int getz(void) {return z;}}
```

Zugriff bei **private**-Ableitung (2)

```
class ZweiteAbleitung : public Ableitung {  
public:  
    ZweiteAbleitung( int i, int j, int k ) : Ableitung(i,j,k) {}  
//    int getx(void) {return x;} // Fehler, kein Zugriff auf priv. Element von Basis  
//    int gety(void) {return y;} // Fehler, protected Elemt. von Basis bleibt private  
//    int getz(void) {return z;} // Fehler, public Element von Basis bleibt private  
};
```

```
int main( int argc, char* argv[] ) {  
    ZweiteAbleitung a2(1,2,3);  
    cout << a2.x << endl;  
    cout << a2.y << endl;  
    cout << a2.z << endl;  
    cout << a2.getx() << endl;  
    cout << a2.gety() << endl;  
    cout << a2.getz() << endl;  
    system("PAUSE");  
    return 0; }
```



Sind die Ausgaben möglich oder nicht?

Zugriff bei **private**-Ableitung (2)

```
class ZweiteAbleitung : public Ableitung {  
public:  
    ZweiteAbleitung( int i, int j, int k ) : Ableitung(i,j,k) {}  
//    int getx(void) {return x;} // Fehler, kein Zugriff auf priv. Element von Basis  
//    int gety(void) {return y;} // Fehler, protected Elemt. von Basis bleibt private  
//    int getz(void) {return z;} // Fehler, public Element von Basis bleibt private  
};
```

```
int main( int argc, char* argv[] ) {  
    ZweiteAbleitung a2(1,2,3);  
//    cout << a2.x << endl;  
//    cout << a2.y << endl;  
//    cout << a2.z << endl;  
//    cout << a2.getx() << endl;  
    cout << a2.gety() << endl;  
    cout << a2.getz() << endl;  
    system("PAUSE");  
    return 0; }
```

Fehler: kein Zugriff auf privates / protected Element von Basis

Fehler: nicht zugreifbar aus **ZweiteAbleitung**

Fehler: kein Zugriff auf Basisklassen-**getx()**

ACHTUNG: **gety()**, **getz()** aus Klasse **Ableitung** werden aufgerufen

Wirkung der Zugriffsmodifizierer

- Zugriff auf Datenelemente der Basisklasse bei verschiedener Ableitung

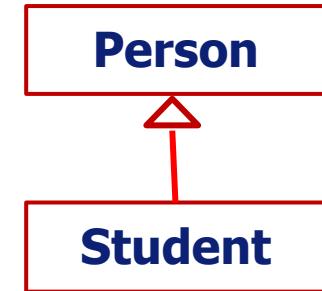
```
class Ableitung : <Zugriffsmodifizierer> Basis  
class ZweiteAbleitung : public Ableitung
```

Schutzattribut in Basisklasse	Art der Ableitung		
	public	protected	private
public	Allgem. Zugriff auch von außen	Ableitung, ZweiteAbleitung	Ableitung
protected	Ableitung, ZweiteAbleitung	Ableitung, ZweiteAbleitung	Ableitung
private	—	—	—

- Beispiel: bei **protected** – Ableitung haben sowohl die Ableitung als auch die ZweiteAbleitung Zugriff auf **public** und **protected** – Basisklassenelemente, jedoch nicht auf **private** - Basisklassenelemente

Private- / protected-Ableitung: Bemerkungen

- Public-Ableitung: „**ist ein**“
 - Ein Student ist eine Person
- Private-/protected-Ableitung: „**ist implementiert in Form von**“
 - Objekt der abgeleiteten Klasse wird *nicht* in das einer Basisklasse konvertiert:
Ein Student ist *keine* Person
 - abgeleitete Klasse erbt die *Implementierung* der Basisklasse (d.h. kann Methoden nutzen, die in der Basisklasse implementiert sind), aber *nicht* die *Schnittstelle* (d.h. verhält sich nach außen nicht wie die Basisklasse)
 - Private-/protected-Ableitung ist eher eine Design-/Implementierungsstrategie; wird selten angewendet; kann ggf. auch durch Komposition realisiert werden
 - Siehe die weiterführende Literatur



Vererbung und Klassentemplates: Beispiel (1)

```
// Klasse basis
class basis
{
protected:
    int wert;

public:
    basis(int w = 0) : wert(w) {}
    void print() { cout << wert << endl; }
};
```

Beispiel für Vererbung
(zunächst ohne Templates)

```
// Klasse abgeleitet
class abgeleitet : public basis
{
private:
    int anzahl;

public:
    abgeleitet( int a = 1, int w = 0 ) : basis(w), anzahl(a) {}
    void print() { cout << wert << ", " << anzahl << endl; }
};
```

Vererbung und Klassentemplates: Beispiel (1)

```
int main(int argc, char* argv[])
{
    basis b(2);
    abgeleitet a(3,5);
    a.print();
    b.print();

    return 0;
}
```

Beispiel für Vererbung
(zunächst ohne Templates)

Ausgabe?

5 , 3
2

Jetzt: Klassentemplate (Übungsaufgabe!)

1. Nur die abgeleitete Klasse als Template (Basisklasse bleibt, wie sie ist):
Wie zuvor: `template<class T> class abgeleitet : public basis {...};`
2. Nur die Basisklasse als Template (abgeleitete Klasse *kein* Template):
 - Bei Ableitung muß **konkreter Typ für Basisklasse** angegeben werden:

```
template<class T> class basis {...};
class abgeleitet : public basis<double> {...};
```

Typangabe

Vererbung und Klassentemplates: Beispiel (2)

```
template<class T>
class basis
{
protected:
    T wert;

public:
    basis(T w = 0) : wert(w) {}
    virtual void print() { cout << wert << endl; }
};
```

Jetzt: *Beide Klassen als Template*

```
template<class T>
class abgeleitet : public basis<T>
{
private:
    T anzahl;

public:
    abgeleitet( T a = 1, T w = 0 ) : basis<T>(w), anzahl(a) {}
    void print() { cout << wert << ", " << anzahl << endl; }
};
```

Compiler:
error: `wert' undeclared

???

Vererbung und Klassentemplates: Beispiel (2): Erklärung

```
template<class T>
class basis
{
protected:
    T wert;

public:
    basis(T w = 0) : wert(w) {}
    virtual void print() { cout << wert <<
};
```

basis<T> hängt von T ab
("dependent name")

In abgeleitet<T>::print():
der Name wert hängt *nicht* von T ab
("non-dependent name");
gilt für Attribute *und* Methoden)

```
template<class T>
class abgeleitet : public basis<T>
{
private:
    T anzahl;

public:
    abgeleitet( T a = 1, T w = 0 ) : basis<T> (w), anzahl(a) {}
    void print() { cout << wert << ", " << anzahl << endl; }
};
```

Um „non-dependent names“ aufzulösen,
schaut der Compiler *nicht* in „dependent“
Basisklassen → findet wert nicht

Vererbung und Klassentemplates: Beispiel (2): Lösung 1

```
template<class T>
class basis
{
protected:
    T wert;

public:
    basis(T w = 0) : wert(w) {}
    virtual void print() { cout << wert << endl; }
};
```

Jetzt: *Beide Klassen als Template*

```
template<class T>
class abgeleitet : public basis<T>
{
private:
    T anzahl;

public:
    abgeleitet( T a = 1, T w = 0 ) : basis<T>(w), anzahl(a) {}
    void print() { cout << this->wert << ", " << anzahl << endl; }
};
```

Lösung 1: **this** – Pointer

- **this** is dependent in einem Klassentemplate
- Daher ist **this->wert** dependent und wird erst bei Instantiierung der Klasse nachgesehen

Vererbung und Klassentemplates: Beispiel (2): Lösung 2

```
template<class T>
class basis
{
protected:
    T wert;

public:
    basis(T w = 0) : wert(w) {}
    virtual void print() { cout << wert << endl; }
};
```

Jetzt: *Beide Klassen als Template*

```
template<class T>
class abgeleitet : public basis<T>
{
private:
    T anzahl;

public:
    abgeleitet( T a = 1, T w = 0 ) : basis<T>(w), anzahl(a) {}
    using basis<T>::wert;
    void print() { cout << wert << ", " << anzahl << endl; }
};
```

Lösung 2: **using** – Direktive
– Namensauflösung wird explizit gemacht...

Vererbung und Klassentemplates: Beispiel (2): Lösung 3

```
template<class T>
class basis
{
protected:
    T wert;

public:
    basis(T w = 0) : wert(w) {}
    virtual void print() { cout << wert << endl; }
};
```

Jetzt: *Beide Klassen als Template*

```
template<class T>
class abgeleitet : public basis<T>
{
private:
    T anzahl;

public:
    abgeleitet( T a = 1, T w = 0 ) : basis<T>(w), anzahl(a) {}
    void print() { cout << basis<T>::wert << ", " << anzahl << endl; }
};
```

Lösung 3: Auflösung über Namensraum
– Aber: Ggf. unerwünschte Effekte, falls *Methode* (statt Attribut) und *virtuell* (siehe später)

Vererbung und Klassentemplates: Beispiel (2)

```
int main(int argc, char* argv[])
{
    basis<double> b(2);
    abgeleitet<double> a(3,5);
    a.print();
    b.print();

    return 0;
}
```

Jetzt: *Beide Klassen als Template*

Ausgabe?

5 , 3
2

- Namensproblem auch, wenn die Basisklasse einen geschachtelten Typ (z.B. eine Klasse) als Attribut enthält
- Siehe hierzu auch

<http://www.parashift.com/c++-faq-lite/nondependent-name-lookup-members.html>

<http://www.parashift.com/c++-faq-lite/nondependent-name-lookup-types.html>

Vererbung, Komposition oder Nutzung?

Zur Erinnerung: **Komposition** ist eine Hat-ein-Beziehung

Beispiel: Ein Pferd *hat ein* Bein, ein Auto *hat einen* Motor

→ Kompositionen drücken *ein dauerhaftes Nutzungsverhältnis* aus

Technische Realisierung

Eine Klasse X definiert eine Membervariable vom Typ einer anderen Klasse Y.

Beispiel:

```
class X {  
    ...  
};
```

```
class Y {  
    X var;  
    // ...  
};
```

"Ein **Y** *hat ein* **X**."

Vererbung, Komposition oder Nutzung?

Nutzung bedeutet, dass ein Objekt einer Klasse X ein Objekt einer Klasse Y nur bei Bedarf erzeugt und verwendet. Y ist somit nicht automatisch dauerhaft Teil von X wie bei der Komposition.

Beispiel: Ein Mensch nutzt ein Objekt Glas zum Trinken.

Das Glas ist nicht dauerhaft Bestandteil des Menschen (→ keine Komposition)

Ein Mensch ist kein Spezialfall eines Glases (→ keine Vererbung)

Vererbung, Komposition oder Nutzung?

Technische Realisierung

In einer Memberfunktion einer Klasse X wird bei Bedarf ein Objekt vom Typ einer anderen Klasse Y erzeugt.

Beispiel:

"Ein **Y nutzt ein X.**"

```
class X {  
    ...  
};
```

```
class Y {  
    void f1(X param) {...}  
  
    void f2() {  
        x lokales_Objekt;  
    }  
};
```

Vererbung, Komposition oder Nutzung?

Welche dieser Beziehungen ist für zwei gegebene Klassen nun die richtige?

Entscheidung anhand der Art der Beziehung:

- Ist-ein-Beziehung → Vererbung
- Hat-ein-Beziehung → Komposition
- Nutzt-ein-Beziehung → Benutzung

Mögliche Strategie zur Erkennung einer Vererbungsbeziehung:

Annahme:

Es besteht eine Beziehung zwischen möglicher Basisklasse X und mögl. abgeleiteter Klasse Y

Frage:

Kann man ein Objekt der Klasse Y auch als ein Objekt seiner Basisklasse X betrachten?

- Ja → Vererbung
- Nein → Komposition oder Benutzung

Vererbung, Komposition oder Nutzung?

Beispiel 1:

- Eine Klasse **motor** sei gegeben
- Eine Klasse **auto** muss geschrieben werden, wobei Methoden und Variablen von **motor** sinnvollerweise verwendet werden sollen.
- Lösung?
 - Komposition, weil ein Auto einen Motor *hat*
 - Einen Motor kann man nicht als Auto betrachten

Vererbung, Komposition oder Nutzung?

Beispiel 2:

- Nun sei die Klasse `auto` gegeben
- Eine Klasse `oldtimer` muss geschrieben werden, wobei Methoden und Variablen von `auto` und damit auch von `motor` sinnvollerweise verwendet werden sollen.
- Lösung?
 - Vererbung, weil ein Oldtimer ein Auto ist. (`motor` wird mitgeerbt)
 - Einen Oldtimer kann man auch einfach als Auto betrachten.

Übersicht (1)

Objektorientierte Programmierung (OOP):

- Einführung
- Objekt und Klasse
- Attribute, Methoden
- Geheimnisprinzip, Kapselung
- Vererbung
- Klassifikation
- Beziehungen zwischen Klassen
- Polymorphie

Schnellkurs C++:

- Einführung
- **Bekannte Sprachmittel:**
 - Variablen, Datentypen, Operatoren, Kontrollstrukturen, Arrays, Strukturen
- **Neue Sprachmittel:**
 - Referenzen
 - Funktionen: Vorgabeargumente, Überladung, Templates
 - Namensräume
 - Ein- und Ausgabe
 - Strings
 - Typumwandlung

Objektorientierte Programmierung mit C++:

- Klassen, Vererbung, Polymorphie, Mehrfachvererbung

Übersicht (2)

Objektorientierte Programmierung mit C++:

- **Klassen**
 - Klasse als Datentyp
 - Member: Instanzvariablen / -methoden, Klassenvariablen / -methoden, Konstruktor, Destruktor, this-Zeiger
 - Sichtbarkeit und Kapselung, Zugriffsspezifizierer, friends
 - Designempfehlungen: const, get/set,
 - Klassentemplates
 - Operatorüberladung für Klassenobjekte
 - Objektverwaltung: Erzeugen, Vergleichen, Kopieren, Auflösen, Komposition...
- **Vererbung**
 - Syntax und Einsatz
 - Basisklassen-Unterobjekt
 - Verdecken, Überschreiben, Überladen
 - Zugriffsmodifizierer, Zugriffsrechte

Übersicht (3)

Objektorientierte Programmierung mit C++:

- Polymorphie
 - Frühe und späte Bindung
 - Virtuelle Funktionen, virtueller Destruktor
 - Abstrakte Methoden
 - Abstrakte Klassen
- Mehrfachvererbung
- Fehlerbehandlung (exception handling)

Objektorientierte Programmierung mit C++: Polymorphie

Polymorphie

Bedeutung des Wortes: Vielgestaltigkeit (s. „Grundbegriffe der OOP“)

Polymorphie (im engeren Sinn): Bei Aufruf ein und derselben Methode über einen Basisklassenzeiger wird – abhängig vom Objekttyp – die Methode der Basisklasse oder aber die entsprechende Methode der abgeleiteten Klasse ausgeführt

Polymorphie in C++ existiert unter folgenden Bedingungen:

- Vererbung und Überschreibung von Basisklassenmethoden in abgeleiteten Klassen
- Indirekter Zugriff auf Objekte über Zeiger oder Referenzen
- Aktivierung der späten (dynamischen) Bindung (siehe gleich)

Wiederholung: Das Basisklassen-Unterobjekt

Vererbte Elemente werden zu einem Teil der abgeleiteten Klasse,
bleiben aber formal Elemente der Basisklasse.



Was bedeutet das für die Praxis?

- ... (s. oben)
- Objekte abgeleiteter Klassen können als Basisklassen-Objekte behandelt werden.

```
int main(void)
{
    abgeleitet abObj;

    basis baObj = abObj;
    basis *baPtr = &abObj;

    return 0;
}
```

Methodenzugriff über Zeiger

- Wiederholung: Zeiger müssen im allgemeinen genau von dem Typ sein, auf dessen Objekt sie zeigen
 - Sonst: Syntaxfehler
- Möglich daher:
 - Basisklassenzeiger auf Basisklassenobjekt
 - Zeiger der abgeleiteten Klasse auf Objekt der abgeleiteten Klasse
- Neu: **Basisklassenzeiger auf Objekt der abgeleiteten Klasse**
 - Objekt wird als ein Objekt der Basisklasse interpretiert
 - Es werden auf die Daten des abgeleiteten Objekts die Methoden der Basisklasse angewendet
 - Methodenaufruf hängt vom Zeigertyp, nicht vom Objekttyp ab

Polymorphie

Einheitliche Objektverwaltung Basisklasse und abgeleitete Klasse

- Objekt einer abgeleiteten Klasse ist auch ein Objekt der Basisklasse
- Ermöglicht **einheitliche Verwaltung** von unterschiedlichen Objekten einer Ableitungshierarchie
- Beispiel: Array von BankAccount-Zeigern, die auf Objekte vom Typ BankAccount bzw. GiroAccount zeigen

```
class BankAccount {  
...  
};
```

```
class GiroAccount :  
    public BankAccount {  
...  
};
```

```
int main(...) {  
    BankAccount* pAccount[2*100];  
  
    for ( int i = 0; i < 2*100; ) {  
        pAccount[i++] = new BankAccount(...);  
        pAccount[i++] = new GiroAccount(...);  
    }  
    ...  
}
```

- Annahme: BankAccount und GiroAccount definieren jeweils eine Funktion `func`
- Welche Funktion wird beim Aufruf `pAccount[i] ->func()` ausgeführt?

Quelle: Microconsult

Funktionsüberschreibung bei Vererbung und Bindung

- Objekt einer abgeleiteten Klasse ist auch ein Objekt der Basisklasse
- Ein Zeiger vom Typ „Zeiger auf Basisklasse“ kann also auch auf ein Objekt einer abgeleiteten Klasse zeigen
- Annahme: Basisklassenmethode wird in abgeleiteter Klasse überschrieben
- Wird diese Methode nun über einen Zeiger / Referenz aufgerufen, gibt es zwei Möglichkeiten:
 - Methode der Basisklasse wird aufgerufen (da Zeiger vom Typ Basisklasse): **frühe oder statische Bindung**
 - Zur Laufzeit wird der Typ des Objekts geprüft, auf das der Zeiger zeigt, und die entsprechende Methode aufgerufen:
späte / dynamische Bindung

Polymorphie

Funktionsüberschreibung bei Vererbung: Beispiel

```
class basis
{
public:
    void func() {
        cout << "Basis ";
    }
};
```

Basisklasse

```
class abgeleitet : public basis
{
};
```

abgeleitete Klasse

```
int main(void)
{
    abgeleitet abObj;

    basis baObj = abObj;
    baObj.func();

    basis *baPtr = &abObj;
    baPtr->func();

    return 0;
}
```

- Was passiert, falls `func()` nur in der Basisklasse `basis` definiert wurde?
→ Kein Problem: Da `baObj` Basisklassenobjekt und `*baPtr` Basisklassenzeiger ist, wird `func()` nur an Basisklassenobjekten aufgerufen, d.h. `func()` aus der Basisklasse wird verwendet

Ausgabe: Basis Basis

Polymorphie

```
class basis
{
};

};
```

Basisklasse

```
class abgeleitet : public basis
{
public:
    void func() {
        cout << "Abgeleitet ";
    }
};
```

abgeleitete Klasse

```
int main(void)
{
    abgeleitet abObj;

    basis baObj = abObj;
    baObj.func();

    basis *baPtr = &abObj;
    baPtr->func();

    return 0;
}
```

- Was passiert, falls `func()` nur in der Klasse `abgeleitet` definiert wurde?

Problem: `func()` existiert nicht in der Klasse `basis`. `abObj` ist zwar eigentlich ein Objekt der Klasse `abgeleitet`, wird aber einer (Zeiger-) Variablen vom Typ der Basisklasse zugewiesen.

Im Normalfall bestimmt der Datentyp einer Variablen oder eines Zeigers, welche Funktionen an ihr (ihm) aufgerufen werden können. Hier Datentyp Basisklasse → `func()` nicht definiert.

→ Compiler-Fehlermeldung

Polymorphie

```
class basis
{
public:
    void func() {
        cout << "Basis ";
    }
};
```

Basisklasse

```
int main(void)
{
    abgeleitet abObj;

    basis baObj = abObj;
    baObj.func();

    basis *baPtr = &abObj;
    baPtr->func();

    return 0;
}
```

```
class abgeleitet : public basis
{
public:
    void func() {
        cout << "Abgeleitet ";
    }
};
```

abgeleitete Klasse

- Was passiert, falls **func()** in **beiden** Klassen definiert wurde (Überschreibung)?

Normalfall: Datentyp einer Variablen oder eines Zeigers bestimmt, welche Funktionsdefinition an ihr (ihm) aufgerufen wird. Hier sind sowohl der Zeiger **baPtr** als auch die Variable **baObj** vom Datentyp der Basisklasse

→ **func()** aus der Basisklasse wird verwendet

Sogenannte **frühe Bindung**

Ausgabe: Basis Basis

Polymorphie

```
class basis
{
public:
    virtual void func() {
        cout << "Basis ";
    }
};
```

Basisklasse

```
int main(void)
{
    abgeleitet abObj;

    basis baObj = abObj;
    baObj.func();

    basis *baPtr = &abObj;
    baPtr->func();
```

Späte Bindung gibt es in C++ nur für **Zeiger** auf Objekte. **baObj** ist jedoch Variable und kein Zeiger. Daher Verwendung von **func()** aus Basisklasse.

```
class abgeleitet : public basis
{
public:
    void func() {
        cout << "Abgeleitet ";
    }
};
```

abgeleitete Klasse

Sonderfall: Verwendung des Schlüsselwortes **virtual** bei der Funktionsdefinition in **basis**

→ aktiviert die **späte Bindung**

Folge: nun bestimmt der **Typ des Objekts**, auf den ein Zeiger weist, welche Funktion im Falle einer Überschreibung ausgeführt wird: nämlich die des Objekts Datentyps (hier also die Funktion in der abgeleiteten Klasse).

Zeiger **abgeleitet** wird verwendet

Ausgabe: **Basis** **Abgeleitet**

Frühe und späte Bindung

- Frühe Bindung / statische Bindung:
 - Anhand des *Typs der Variablen* (Objekt, Zeiger) wird vom Compiler bereits *zum Übersetzungszeitpunkt* statisch („früh“) festgelegt, welche Funktion aufgerufen wird (Basisklasse oder abgeleitete Klasse)
- Späte Bindung / dynamische Bindung:
 - Compiler erzeugt Code, der erst *zur Laufzeit* dynamisch („spät“) anhand des *Typs des Objekts* die richtige Funktion feststellt (bei Verwendung von Zeigern oder Referenzen)
 - Realisierung: Schlüsselwort `virtual` bei Deklaration in Basisklasse
 - Langsamere Ausführungszeiten als bei statischer Bindung!

Virtuelle Funktionen

- Schlüsselwort **virtual** bei Funktionsdeklaration in Basisklasse
- Bei Überschreibung in abgeleiteter Klasse muß Schlüsselwort **virtual** nicht wiederholt werden (wird aber zu besserer Lesbarkeit empfohlen)
- Verlangsamt den Code!
- Aktiviert die **späte (dynamische) Bindung**: Wird eine virtuelle Funktion *über einen Zeiger oder eine Referenz* der Basisklasse aufgerufen, wird die zugehörige Elementfunktion der Klasse aufgerufen, von der das Objekt (auf das der Zeiger oder die Referenz verweist) tatsächlich ist

Polymorphie

- Polymorphie (im engeren Sinn):
 - Fähigkeit, den Typ eines Objektes zur Laufzeit zu ermitteln
 - Aufruf der entsprechenden *redefinierten* Methoden
 - Abhängig vom tatsächlichen Objekttyp nimmt ein Zeiger bzw. eine Referenz auf ein Objekt (und damit die entsprechende Methode) verschiedene Gestalt an
- Polymorphie im engeren Sinn existiert nur
 - Für streng redefinierte Methoden
 - Bei einem indirekten Zugriff über Zeiger oder Referenzen
 - Im Falle der späten (dynamischen) Bindung
- Polymorphie (im weiteren Sinn): Funktionspolymorphie (Funktionsüberladung)
 - Funktionen (Methoden) haben gleichen Namen, aber unterschiedl. Signatur

Polymorphie

Polymorphie: Beispiel

```
class Person {  
protected: string name;  
  
public:  
    Person (string s = "") : name(s) {}  
    virtual void print()  
    { cout << "Person: " << name << endl; }  
};
```

```
class Student : public Person {  
protected: int matNr;  
  
public:  
    Student (string s, int m) : Person(s), matNr(m) {}  
    virtual void print()  
    { cout << "Student: " << name << " MatNr: " << matNr << endl; }  
};
```

```
class Dozent : public Person {  
protected: string vorlesung;  
  
public:  
    Dozent (string s, string v) : Person(s), vorlesung(v) {}  
    virtual void print()  
    { cout << "Dozent: " << name << " Vorlesung: " << vorlesung << endl; }  
};
```

Polymorphie

Polymorphie: Beispiel

Einheitliche Verwaltung von Person / Student / Dozent in Array aus Zeigern auf „Person“

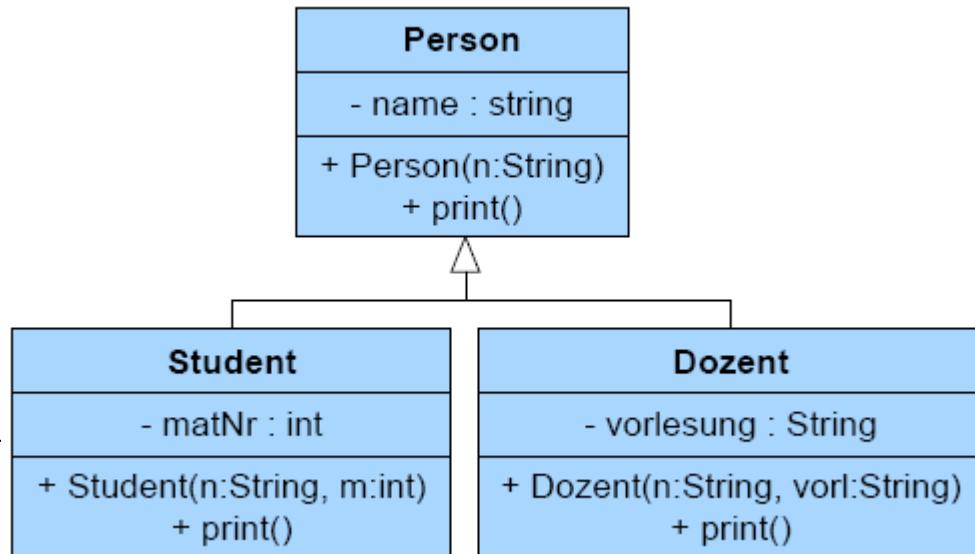
```
int main( int argc, char* argv[] )
{
    Person* p[4];

    p[0] = new Student("Maier", 1312);
    p[1] = new Person("Mueller");
    p[2] = new Student("Schulze", 2435);
    p[3] = new Dozent("Meyer", "C++");

    for (int i = 0; i < 4; i++)
    {
        p[i]->print();
    }

    for (int i = 0; i < 4; i++)
        delete p[i];

    return 0;
}
```



Quelle: Bittel

Ausgabe

Student: Maier MatNr: 1312
Person: Mueller
Student: Schulze MatNr: 2435
Dozent: Meyer Vorlesung: C++
Press any key to continue . . .

Polymorphie

Polymorphie: Beispiel

```
class Person {  
protected: string name;  
  
public:  
    Person (string s = "") : name(s) {}  
    void print() // virtual  
    { cout << "Person: " << name << endl; }  
};
```

```
int main( int argc, char* argv[] )  
{  
    Person* p[4];  
  
    p[0] = new Student("Maier", 1312);  
    p[1] = new Person("Mueller");  
    p[2] = new Student("Schulze", 2435);  
    p[3] = new Dozent("Meyer", "C++");  
  
    for (int i = 0; i < 4; i++)  
    {  
        p[i]->print();  
    }  
  
    for (int i = 0; i < 4; i++)  
        delete p[i];  
  
    return 0;  
}
```

Ausgabe **ohne** **virtual**:

Person: Maier
Person: Mueller
Person: Schulze
Person: Meyer
Press any key to continue . . .

Virtueller Destruktor

```
class A
{
...
public:
...
~A() {...}
};
```

```
class B : public A
{
...
public:
...
~B() {...}
};
```

```
int main( ... )
{
    A* pB = new B;
    ...
    delete pB; // ruft nur ~A() auf!
    ...
}
```

- Falls Objekte einer abgeleiteten Klasse über einen Basisklassenzeiger zerstört werden, wird i.a. nur der Destruktor der Basisklasse aufgerufen
 - Da A::~A() nicht virtuell ist → statische Bindung an A::~A()
- B::~B() wird daher nicht aufgerufen und es entsteht ein Speicherleck (falls in B zusätzlicher Speicher alloziert wurde)
- Daher sollten die Destruktoren **virtual** deklariert werden
 - Um sicherzugehen, daß auch die Destruktoren der abgeleiteten Klassen aufgerufen werden

Virtueller Destruktor

```
class A
{
...
public:
...
virtual ~A() { ... }
};
```

```
class B : public A
{
...
public:
...
virtual ~B() { ... }
};
```

```
int main( ... )
{
    A* pB = new B;
    ...
    delete pB; // ruft ~B(),
                  ~A() auf!
    ...
}
```

- Der vom Compiler zugewiesene Ersatzdestruktur ist nicht virtuell!
- Daher: **Bei Basisklassen sollten die Destruktoren virtuell definiert werden!**
 - Schaltet dynamische Bindung für den Destruktor ein
 - Gewährleistet, daß auch für Zeiger von Basisklassentypen, die auf Objekte einer abgeleiteten Klasse verweisen, der korrekte Destruktor aufgerufen wird (nämlich der aus der abgeleiteten Klasse)

Abstrakte Methoden (pure virtual function)

- Im Falle von Vererbung kann es sein, daß eine Methode in der Basisklasse nur den Zweck hat, in abgeleiteten Klassen überschrieben zu werden
 - D.h. es gibt keine gemeinsame Grundfunktionalität in der Basisklasse, sondern nur jeweils eigene Implementierungen in der abgeleiteten Klasse
- Eine solche Funktion kann als **abstrakte Funktion (pure virtual function)** definiert werden, gekennzeichnet durch den Zusatz "**=0**"

```
virtual rückgabetyp funktionsname(PARAMETERLISTE) = 0;
```

```
virtual double flaeche() = 0;
```

Aufgabe der abstrakten Methode:

- Weitervererbung als Schnittstellenvorgabe an abgeleitete Klassen
- Aufgabe der abgeleiteten Klasse: Überschreibung der geerbten rein virtuellen Methoden und Ausstattung mit eigenem Anweisungsteil

Abstrakte Klassen

- Ein Klasse, die mindestens eine rein virtuelle Methode enthält, heißt **abstrakte Klasse**
- Von einer abstrakten Klasse lassen sich keine Instanzen bilden!
 - Da abstrakte Funktion keine Implementierung hat, lässt Compiler dies nicht zu
- Zeiger oder Referenz auf abstrakte Klasse ist aber erlaubt!

```
class A ←———— Abstrakte Klasse
{
    public:
        virtual double getFlaeche() = 0;
    ...
};                                Rein virtuelle Methode
```

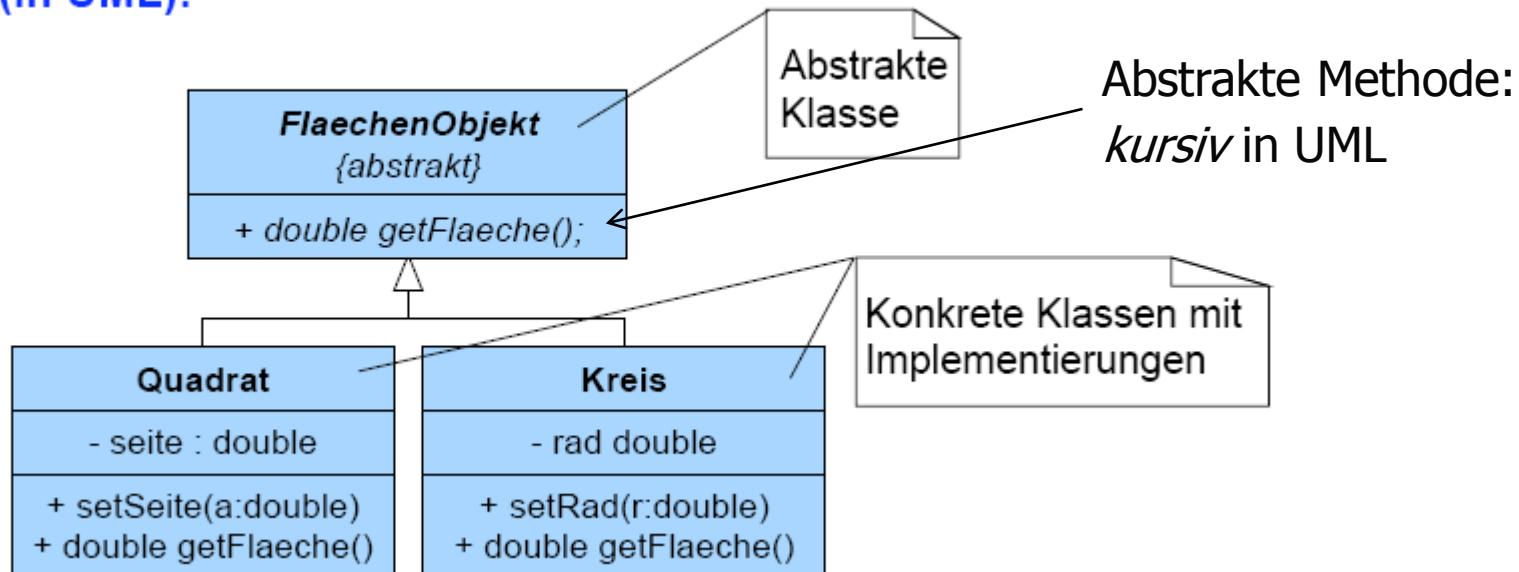
```
int main(...)
{
    A a;      // nicht erlaubt
    A* pa;   // o.k.
    ...
};
```

- Falls eine abstrakte Methode geerbt, aber nicht in der abgeleiteten Klasse überschrieben wird, enthält die abgeleitete Klasse eine abstrakte Methode und wird selbst zur abstrakten Klasse

Abstrakte Klassen: Beispiel (1)

- Abstrakte Basisklasse **FlaechenObjekt**
 - Legt fest, daß alle von ihr abgeleiteten Klassen über die Eigenschaft Fläche verfügen sollen
 - Abgeleitete Klassen können dann tatsächliche Figuren der 2-dim. Ebene beschreiben, im Beispiel Quadrat und Kreis

Beispiel (in UML):



Polymorphie

Abstrakte Klassen: Beispiel (2)

Abstrakte Klasse

```
class FlaechenObjekt
```

```
{
```

```
public:
```

```
virtual double getFlaeche() = 0;
```

```
};
```

Rein virtuelle Methode

```
class Quadrat : public FlaechenObjekt {
```

```
double seite;
```

```
public:
```

```
void setSeite( double a ) { seite = a; }
```

```
virtual double getFlaeche()
```

```
{ return seite * seite; }
```

```
};
```

```
class Kreis : public FlaechenObjekt {
```

```
double rad;
```

```
public:
```

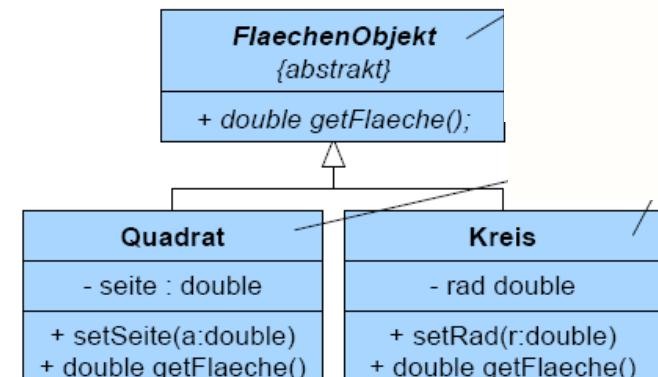
```
void setRad( double r ) { rad = r; }
```

```
virtual double getFlaeche()
```

```
{ return PI * rad * rad; }
```

```
};
```

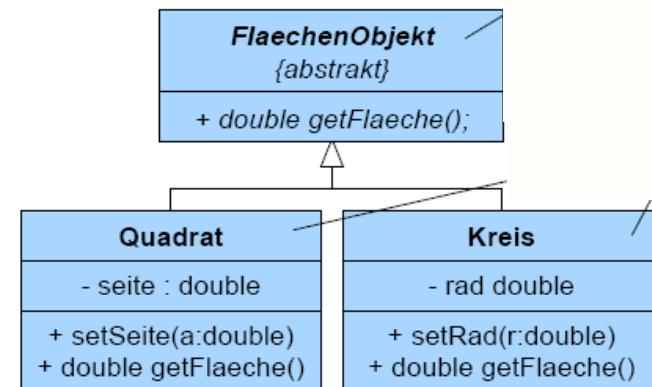
Quelle: Bittel



Polymorphie

Abstrakte Klassen: Beispiel (3)

```
int main ( int argc, char* argv[] ) {  
  
    Quadrat* q1 = new Quadrat;  
    q1->setSeite(5);  
  
    Kreis* k = new Kreis;  
    k->setRad(3);  
  
    Quadrat* q2 = new Quadrat;  
    q2->setSeite(4);  
  
    FlaechenObjekt* f[3];  
    f[0] = q1; f[1] = k; f[2] = q2;  
  
    for (int i = 0; i < 3; i++ )  
        cout << f[i]->getFlaeche() << endl;  
  
    ...  
}
```



Ausgabe

25	←	Quadrat::getFlaeche()
28.2743	←	Kreis::getFlaeche()
16	←	Quadrat::getFlaeche()

Press any key to continue . . .

Beispiel Flaechenobjekt (ausführlicher)

- Ziel: Veranschaulichung der Prinzipien Vererbung, Polymorphie und abstrakte Klassen am Beispiel der Klasse **FlaechenObjekt**
- Aufgabe: Implementierung von Klassen für geometrische Objekte **Punkt**, **Kreis**, **Rechteck** etc.

Punkt
– x: float
– y: float

Kreis
– center_x: float
– center_y: float
– radius: float

Rechteck
– ecke_x: float
– ecke_y: float
– breite: float
– laenge: float

Kreis
– center_x: float
– center_y: float
– radius: float

Rechteck
– ecke_x: float
– ecke_y: float
– breite: float
– laenge: float

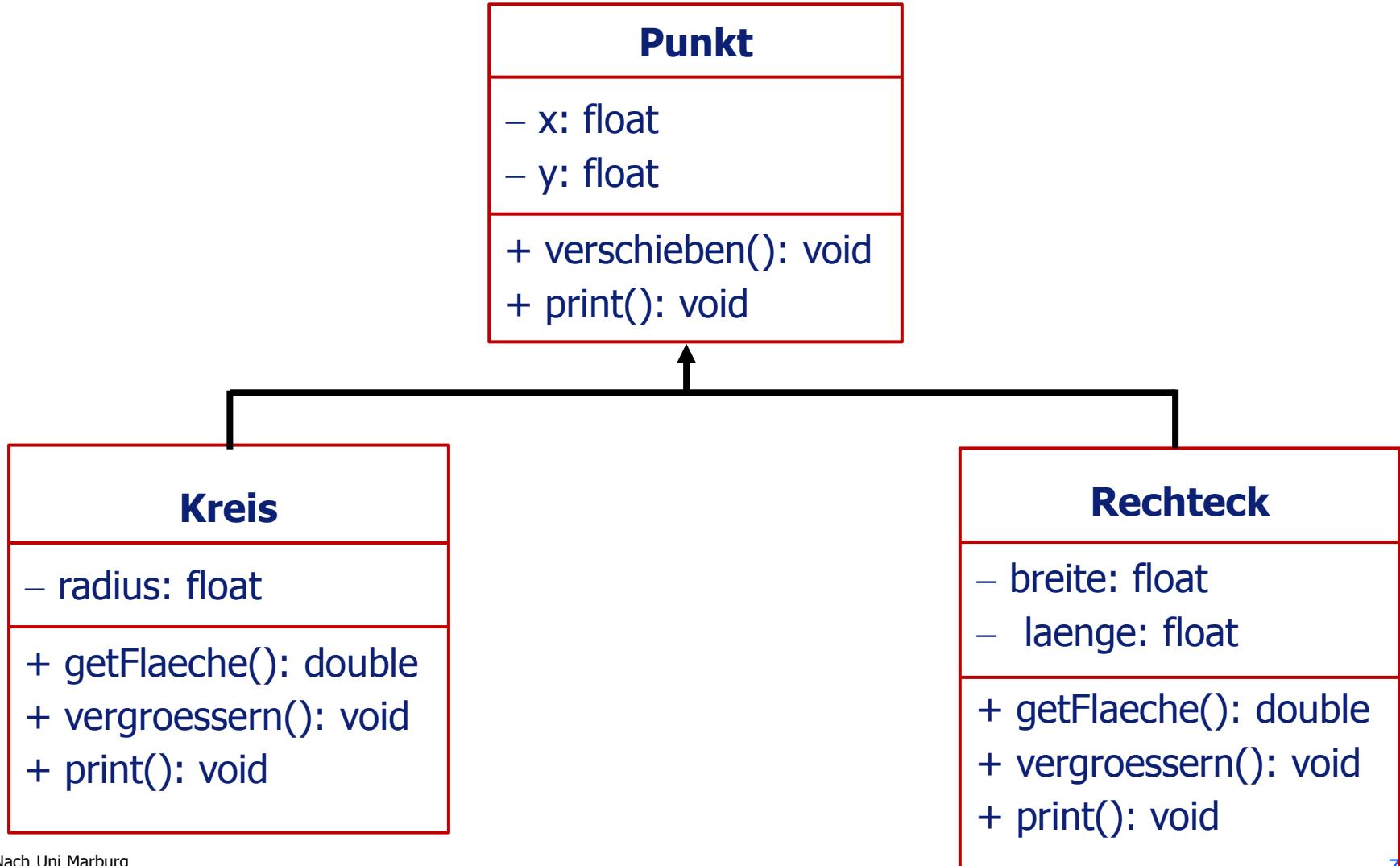
Kreis
+ verschieben(): void
+ getFlaeche(): double
+ vergroessern(): void
+ print(): void

Rechteck
+ verschieben(): void
+ getFlaeche(): double
+ vergroessern(): void
+ print(): void

Polymorphie

Beispiel Flaechenobjekt

- Möglichkeit 1: Ableitung von **Kreis**, **Rechteck** von der Klasse **Punkt**

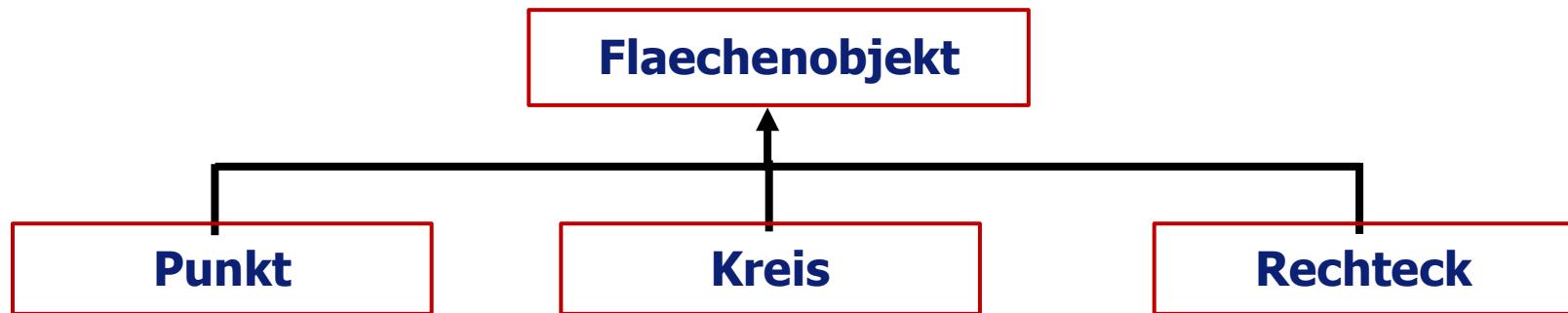


Beispiel Flaechenobjekt

- Möglichkeit 1: Ableitung von **Kreis**, **Rechteck** von der Klasse **Punkt**
- Vorteile:
 - Durch Vererbung: weniger Programmieraufwand
 - Bsp Kreis: Mit Vererbung müssen nur 6 (statt 13) Attribute und Methoden implementiert werden
 - Durch Datenkapselung: Zugriffskontrolle
- Nachteile dieser Realisierung:
 - *Ist* ein Kreis ein Punkt? *Ist* ein Rechteck ein Punkt?
 - Oder *hat* ein Kreis bzw. ein Rechteck einen Punkt...?!
 - Was ist bei Erweiterung der Klasse Punkt?
 - Bsp.: Zusätzliches Attribut **zeichensymbol** (zur Darstellung, z.B. "x")
- Wird an alle abgeleiteten Klassen weitervererbt
- Evt. unerwünscht

Beispiel Flaechenobjekt

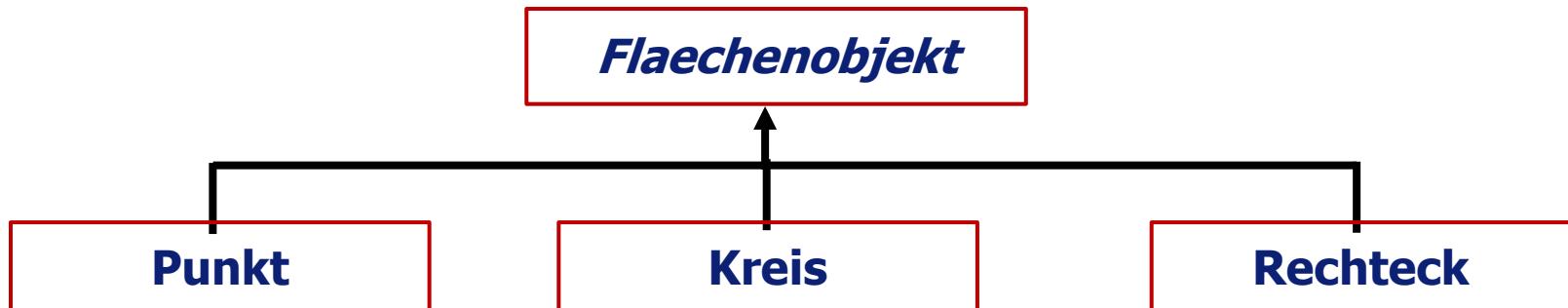
- Möglichkeit 2: Einführung einer Klasse **Flaechenobjekt**, von der die Klassen **Kreis**, **Rechteck** und **Punkt** abgeleitet werden



- Vorteile:
 - Kreis, Rechteck, Punkt *sind* (ggf. entartete) Flaechenobjekte
 - Bei Erweiterung der Klasse Punkt: Zusätzliche Attribute werden *nicht* an Kreis oder Rechteck weitervererbt

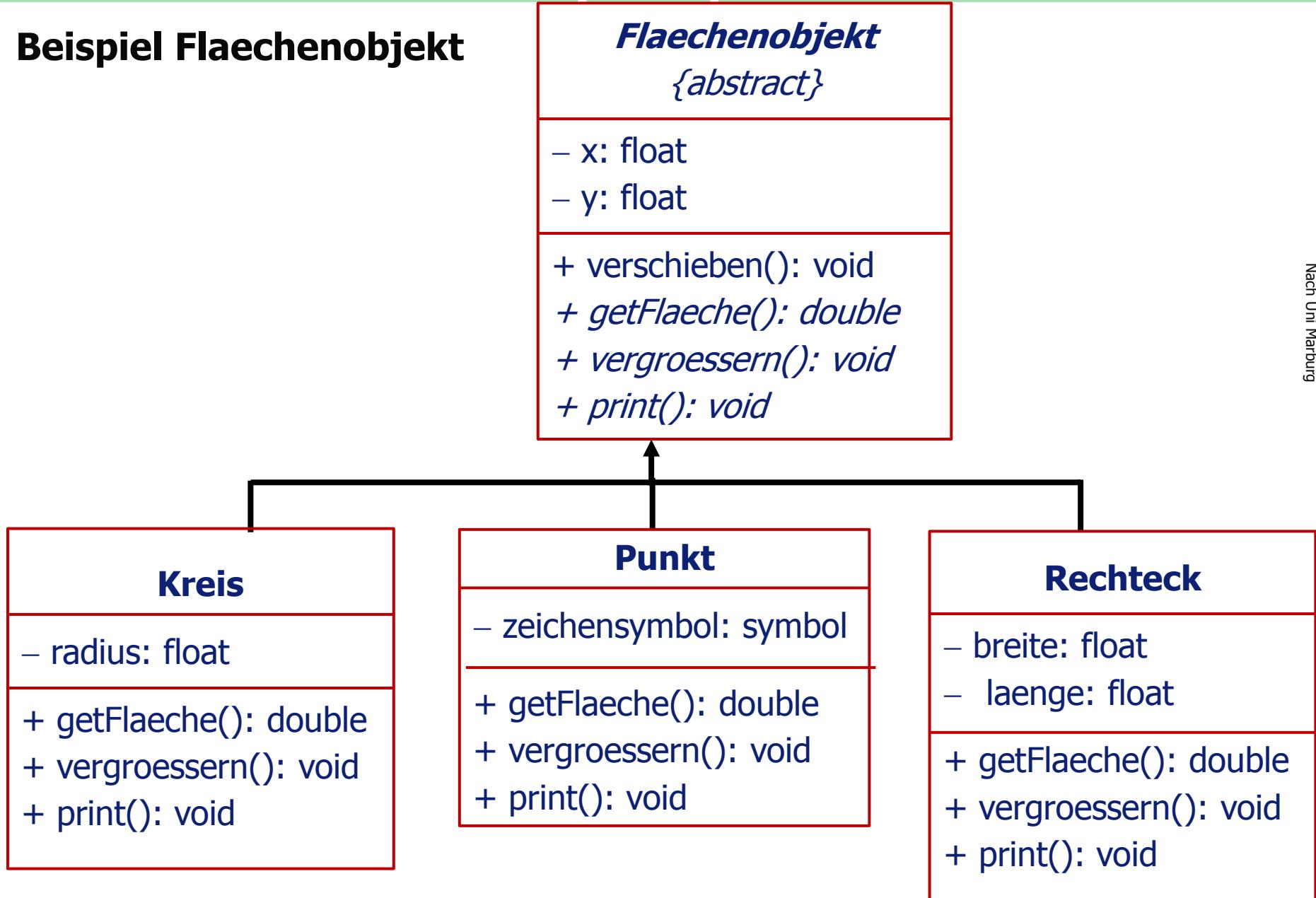
Beispiel Flaechenobjekt

- **Flaechenobjekt:**
 - abstrakter Begriff, soll nicht instanziert werden
 - Gibt Schnittstelle vor (Eigenschaften, die eine von **Flaechenobjekt** abgeleitete Klasse haben soll)
- Abstrakte Klasse



Polymorphie

Beispiel Flaechenobjekt



Polymorphie

Beispiel Flaechenobjekt

flaechenobjekt.h

```
#ifndef _FLAECHENOBJEKT_H_
#define _FLAECHENOBJEKT_H_

class Flaechenobjekt {
protected:
    float x, y;

public:

    Flaechenobjekt( float x = 0.0, float y = 0.0 ) : x(x), y(y) {}

    float getX() { return x; }

    float getY() { return y; }

    void setX( float x ) { this->x = x; }

    void setY( float y ) { this->y = y; }

    void verschieben ( float dx, float dy ) { x += dx; y += dy; }

    virtual double getFlaeche() const = 0;

    virtual void vergroessern( float f ) = 0;

    virtual void print() const = 0;
};

#endif
```

Zur Sichtbarkeit in abgeleiteten Klassen

Nicht-virtuelle Methode

Rein virtuelle Methoden

Abstrakte Methoden müssen in abgeleiteten Klassen
implementiert werden (sonst: abgeleitete Klasse abstrakt!)

Polymorphie

Beispiel Flaechenobjekt

punkt.h

```
#ifndef _PUNKT_H_
#define _PUNKT_H_

enum symbol { cross, raute, dreieck, kreis }; // zur Darstellung

class Punkt : public Flaechenobjekt {
    symbol zeichensymbol; ← spezifisch für Klasse Punkt

public:
    Punkt( float x = 0, float y = 0 ) : Flaechenobjekt(x, y),
                                              zeichensymbol( cross ) {}

    virtual double getFlaeche() const { return 0; }
    virtual void vergroessern( float f ) {};
    virtual void print() const;
    friend std::ostream& operator<<(std::ostream& os, const Punkt& p);
    void setZeichensymbol( symbol s ) { zeichensymbol = s; }
    symbol getZeichensymbol() { return zeichensymbol; }

};
```

Weiterleitung

Auch „triviale“ Methoden (`getFlaeche`, `vergroessern`) müssen implementiert werden; sonst wäre Klasse `Punkt` abstrakt

Beispiel Flaechenobjekt

punkt.h (Fortsetzung)

```
// ... Fortsetzung

inline void Punkt::print() const
{
    std::cout << "Punkt: Koordinaten (" << x << ", " << y << ")" <<
    std::endl;
}

inline std::ostream& operator<<( std::ostream& os, const Punkt& p )
{
    os << "Punkt: Koordinaten(" << p.x << ", " << p.y << ")";
    return os;
}

#endif
```

Außerhalb der Klassendefinition: kein „virtual“

Polymorphie

Beispiel Flaechenobjekt

kreis.h

```
#ifndef _KREIS_H_
#define _KREIS_H_

#include <iostream>
extern const double PI;

class Kreis : public Flaechenobjekt {
    float radius;
public:
    Kreis( float x, float y, float r ) : Flaechenobjekt(x, y),
                                             radius(r) { }

    float getRadius() { return radius; }
    void setRadius( float r ) { radius = r; }
    virtual double getFlaeche() const { return PI * radius * radius; }
    virtual void vergroessern( float f );
    virtual void print() const;
    friend std::ostream& operator<<(std::ostream& os, const Kreis& k);
};

// ... Fortsetzung
```

Weiterleitung

~~virtual void vergroessern(float f);~~

~~virtual void print() const;~~

~~friend std::ostream& operator<<(std::ostream& os, const Kreis& k);~~

virtual kann hier fehlen
(da in Basisklasse vorhanden)

Beispiel Flaechenobjekt

kreis.h (Fortsetzung)

```
// ... Fortsetzung

inline void Kreis::vergroessern( float f ) {
    std::cout << "Kreis: vergroessern" << std::endl;
    radius *= f;
}

inline void Kreis::print() const {
    std::cout << "Kreis: Mittelpunkt: (" << x << ", " << y << ")",
              Radius: " << radius << std::endl;
    std::cout << "Flaeche Kreis: " << getFlaeche() << std::endl;
}

inline std::ostream& operator<<( std::ostream& os, const Kreis& k ) {
    os << "Kreis: Mittelpunkt: (" << k.x << ", " << k.y << ")",
        Radius: " << k.radius << std::endl;
    os << "Flaeche Kreis: " << k.getFlaeche();
    return os;
}

#endif
```

Wegen **const Kreis& k**:
getFlaeche() muß **const** sein!

Polymorphie

Beispiel Flaechenobjekt

rechteck.h

```
#ifndef _RECHTECK_H_
#define _RECHTECK_H_

#include <iostream>

class Rechteck : public Flaechenobjekt {
    float breite, laenge;

public:
    Rechteck( float x, float y, float b, float l ) :
        Flaechenobjekt(x, y), breite(b), laenge(l) {}

    float getBreite() { return breite; }
    float getLaenge() { return laenge; }
    void setBreite( float b ) { breite = b; }
    void setLaenge( float l ) { laenge = l; }
    virtual double getFlaeche() const { return breite * laenge; }
    virtual void vergroessern( float f );
    virtual void print() const;
    friend std::ostream& operator<<( std::ostream& os, const
                                         Rechteck& r );
}; // ... Fortsetzung
```

Weiterleitung

Beispiel Flaechenobjekt

rechteck.h (Fortsetzung)

```
// ... Fortsetzung

inline void Rechteck::vergroessern( float f ) {
    // Vergroessern bei festem Zentrum (x,y: linke untere Ecke)
    std::cout << "Rechteck: vergroessern" << std::endl;
    verschieben( 0.5 * breite *( f - 1 ), 0.5 * laenge * ( f - 1 ) );
    breite *= f;
    laenge *= f;
}

inline void Rechteck::print() const {
    std::cout << "Rechteck: Mittelpunkt: (" << x << ", " << y <<
    "), Breite: " << breite << ", Laenge = " << laenge << std::endl;
    std::cout << "Flaeche Rechteck: " << getFlaeche() << std::endl;
}

inline std::ostream& operator<<(std::ostream& os, const Rechteck& r){
    os <<"Rechteck: Mittelp.: (" << r.x << ", "<< r.y <<"), Breite: "
        << r.breite << ", Laenge: " << r.laenge << std::endl;
    os << "Flaeche Rechteck: " << r.getFlaeche(); return os; }

#endif
```

Polymorphie

Beispiel Flaechenobjekt

main.cpp

```
using namespace std;
const double PI = 3.1415927;

int main(int argc, char *argv[])
{
    // Flaechenobjekt f; // Fehler: abstrakte Klasse
    // Flaechenobjekt* pf = new Flaechenobjekt[3]; // Fehler
    Flaechenobjekt *pf[3];
    pf[0] = new Punkt( 1, 2 );
    pf[1] = new Kreis( 1, 2, 3 );
    pf[2] = new Rechteck( 1, 2, 3, 4 );

    for ( int i = 0; i < 3; i++ ) {
        pf[i]->vergroessern( 10.0 );
        pf[i]->print();
    }

    for ( int i = 0; i < 3; i++ ) delete pf[i];
    return EXIT_SUCCESS;
}
```

```
#include <iostream>
#include "flaechenobjekt.h"
#include "punkt.h"
#include "kreis.h"
#include "rechteck.h"
```

Fehler: **Flaechenobjekt**
kann nicht instanziert werden

Einheitliche Verwaltung über
Basisklassenzeiger

Beispiel Flaechenobjekt

Polymorphie:

Trotz gemeinsamer Verwaltung
mit **Basisklassenzeigern**:
Aufruf der Methoden der
abgeleiteten Klassen

Ausgabe:

Punkt: Koordinaten (1, 2)

Kreis: vergroessern

Kreis: Mittelpunkt: (1, 2), Radius: 30

Flaeche Kreis: 2827.43

Rechteck: vergroessern

Rechteck: Mittelpunkt: (14.5, 20), Breite: 30, Laenge = 40

Flaeche Rechteck: 1200

Drücken Sie eine beliebige Taste . . .

Polymorphie

Beispiel Flaechenobjekt

main.cpp

```
using namespace std;
const double PI = 3.1415927;

int main(int argc, char *argv[])
{
    Flaechenobjekt *pf;
    pf = new Kreis( 1, 2, 3 );

    // pf->setRadius(10.0); ← Fehler: class Flaechenobjekt
    // has no member named 'setRadius'

    Kreis* pk;
    if ( pk = dynamic_cast<Kreis*>(pf) )
        pk->setRadius(10.0);

    pf->print();

    delete pf;
    return EXIT_SUCCESS;
}
```

```
#include <iostream>
#include "flaechenobjekt.h"
#include "punkt.h"
#include "kreis.h"
#include "rechteck.h"
```

Zugriff über Basisklassenzeiger auf Methode der abgeleiteten Klasse?

Fehler: class Flaechenobjekt
has no member named 'setRadius'

Umwandlung auf Zeiger des abgeleiteten
Objekts mit **dynamic_cast**;
Falls Konvertierung nicht möglich:
Nullzeiger wird zurückgeliefert

Polymorphie

Beispiel Flaechenobjekt

main.cpp

```
using namespace std;
const double PI = 3.1415927;

int main(int argc, char *argv[])
{
    Flaechenobjekt *pf;
    pf = new Kreis( 1, 2, 3 );

//    pf->setRadius(10.0);

    Kreis* pk;
    if ( pk = dynamic_cast<Kreis*>(pf) )
        pk->setRadius(10.0);

    pf->print();

    delete pf;
    return EXIT_SUCCESS;
}
```

```
#include <iostream>
#include "flaechenobjekt.h"
#include "punkt.h"
#include "kreis.h"
#include "rechteck.h"
```

Zugriff über Basisklassenzeiger auf
Methode der abgeleiteten Klasse?

Ausgabe?

```
Kreis: Mittelpunkt: (1, 2), Radius: 10
Flaeche Kreis: 314.159
Drücken Sie eine beliebige Taste . . .
```

Übersicht (1)

Objektorientierte Programmierung (OOP):

- Einführung
- Objekt und Klasse
- Attribute, Methoden
- Geheimnisprinzip, Kapselung
- Vererbung
- Klassifikation
- Beziehungen zwischen Klassen
- Polymorphie

Schnellkurs C++:

- Einführung
- **Bekannte Sprachmittel:**
 - Variablen, Datentypen, Operatoren, Kontrollstrukturen, Arrays, Strukturen
- **Neue Sprachmittel:**
 - Referenzen
 - Funktionen: Vorgabeargumente, Überladung, Templates
 - Namensräume
 - Ein- und Ausgabe
 - Strings
 - Typumwandlung

Objektorientierte Programmierung mit C++:

- Klassen, Vererbung, Polymorphie, Mehrfachvererbung

Übersicht (2)

Objektorientierte Programmierung mit C++:

- **Klassen**
 - Klasse als Datentyp
 - Member: Instanzvariablen / -methoden, Klassenvariablen / -methoden, Konstruktor, Destruktor, this-Zeiger
 - Sichtbarkeit und Kapselung, Zugriffsspezifizierer, friends
 - Designempfehlungen: const, get/set,
 - Klassentemplates
 - Operatorüberladung für Klassenobjekte
 - Objektverwaltung: Erzeugen, Vergleichen, Kopieren, Auflösen, Komposition...
- **Vererbung**
 - Syntax und Einsatz
 - Basisklassen-Unterobjekt
 - Verdecken, Überschreiben, Überladen
 - Zugriffsmodifizierer, Zugriffsrechte

Übersicht (3)

Objektorientierte Programmierung mit C++:

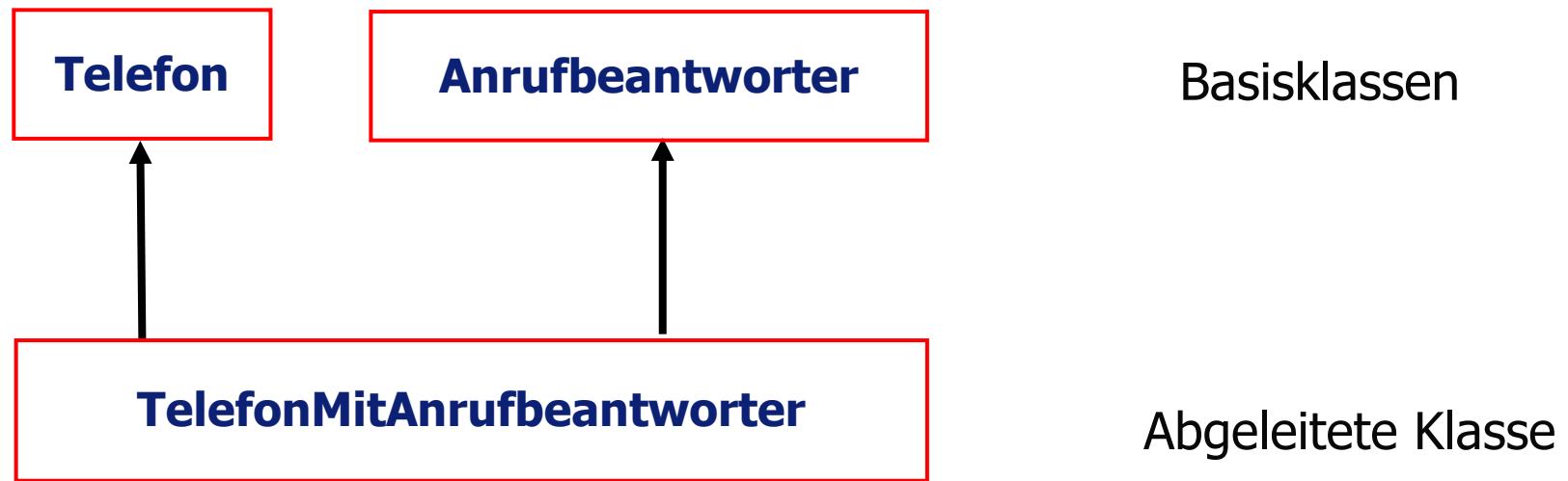
- Polymorphie
 - Frühe und späte Bindung
 - Virtuelle Funktionen, virtueller Destruktor
 - Abstrakte Methoden
 - Abstrakte Klassen
- Mehrfachvererbung
- Fehlerbehandlung (exception handling)

Mehrfachvererbung

Mehrfachvererbung

Mehrfachvererbung

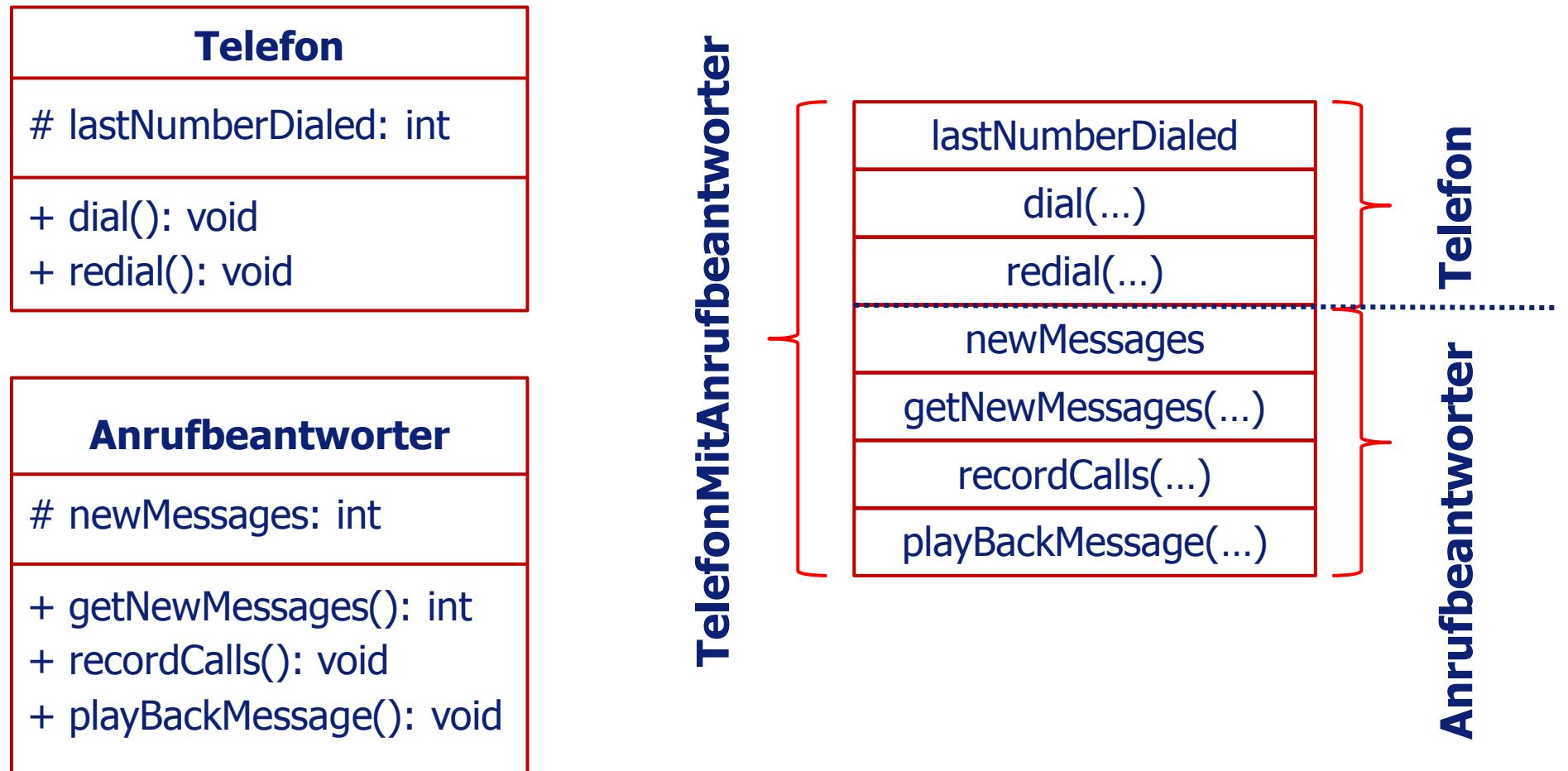
- Eine Klasse kann von mehreren Klassen gleichzeitig abgeleitet werden:
Mehrfachvererbung



Mehrfachvererbung

Mehrfachvererbung

- Eine Klasse, die von mehreren Klassen abgeleitet ist, erbt alle Datenelemente und Elementfunktionen aller Basisklassen



Quelle: Microconsult

Mehrfachvererbung

Syntax und Beispiel

Basisklassen durch
Kommata getrennt

```
class TelefonMitAnrufbeantworter : public Telefon,  
                                    public Anrufbeantworter
```

```
{  
...  
};
```

TelefonMitAb.

```
int main( int argc, char* argv[] ) {  
    TelefonMitAnrufbeantworter tab;  
    tab.dial( 12345 );  
    tab.redial();  
    tab.recordCalls();  
    int msgs = tab.getNewMessages();  
    for ( int i = 0; i < msgs; i++ ) {  
        tab.playBackMessage(i);  
    }  
    ...  
}
```

von **Telefon**

von **Anrufbeantworter**

lastNumberDialed: int
newMessages: int

+ dial(): void
+ redial(): void
+ getNewMessages(): int
+ recordCalls(): void
+ playBackMessage(): void

Interpretation als Basisklassenobjekt

- Ein Objekt, das von mehreren Basisklassen erbt, ist auch ein Objekt jeder seiner Basisklassen
- Deshalb kann die Adresse eines solchen Objektes problemlos Zeigern seiner Basisklassen zugewiesen werden
 - Aber: die zugewiesene Adresse ist i.a. nicht identisch mit der physikalischen Startadresse des Gesamtobjektes

```
int main( int argc, char* argv[] )  
{  
    TelefonMitAnrufbeantworter tab;  
  
    Telefon *pTel = &tab;  
    Anrufbeantworter *pAB = &tab;  
    ...  
}
```

Mehrfachvererbung

Mehrfachvererbte Objekte als Funktionsaufrufparameter

- Ein Objekt einer Klasse, die von mehreren Basisklassen abgeleitet ist, kann behandelt werden wie ein Objekt einer der Basisklassen

```
class TelefonMitAnrufbeantworter : public Telefon,  
                                    public Anrufbeantworter {...};
```

```
void f( Telefon& t ) {  
    t.dial( 12345 ); }
```

```
void g( Anrufbeantworter& ab ) {  
    int msgs = ab.getNewMessages(); }
```

```
int main( int argc, char* argv[] )  
{  
    TelefonMitAnrufbeantworter tab;  
  
    f(tab);  
    g(tab);  
    ...  
}
```

tab ist Telefon

tab ist Anrufbeantworter

Namenskollisionen

- Gleichnamige Elemente in mehreren Basisklassen:

Telefon

- ...

+ displayTime(): void

Anrufbeantworter

- ...

+ displayTime(): void

```
class TelefonMitAnrufbeantworter : public Telefon,  
                                    public Anrufbeantworter {...};
```

```
int main( int argc, char* argv[] )  
{  
    TelefonMitAnrufbeantworter tab;  
  
    // tab.displayTime(); ←  
    tab.Telofon::displayTime(); ←  
    tab.Anrufbeantworter::displayTime(); ←  
    ...  
}
```

Fehler: nicht eindeutig!

Mehrdeutigkeitsauflösung durch
Angabe des Klassennamens

Namenskollisionen

- Auflösung der Mehrdeutigkeit auch durch
 - **using**-Direktive

```
class TelefonMitAnrufbeantworter : public Telefon,  
                                    public Anrufbeantworter  
{  
public:  
    using Telefon::displayTime;  
};
```

```
int main( int argc, char* argv[] )  
{  
    TelefonMitAnrufbeantworter tab;  
    tab.displayTime();  
    ...  
}
```

o.k.



Namenskollisionen

- Auflösung der Mehrdeutigkeit auch durch
 - Überschreiben

```
class TelefonMitAnrufbeantworter : public Telefon,
                                         public Anrufbeantworter  {
    bool recording;

public:
    TelefonMitAnrufbeantworter() : recording(false) {}
    void recordCalls()
    { recording = true; Anrufbeantworter::recordCalls(); }
    void stopRecording() {
        if ( recording )
            { recording = false; Anrufbeantworter::stopRecording(); }

void displayTime() {           ←
    recording ? Anrufbeantworter::displayTime()
               : Telefon::displayTime() ; }

};
```

Überschreiben von
displayTime()

Namenskollisionen

- Auflösung der Mehrdeutigkeit auch durch
 - Überschreiben

```
int main( int argc, char* argv[] )  
{  
    TelefonMitAnrufbeantworter tab;           ← Konstruktor → recording = false  
  
    tab.displayTime();                         ← Telefon::displayTime()  
  
    tab.recordCalls();                        ← recording = true  
  
    tab.displayTime();                         ← Anrufb::displayTime()  
  
    tab.stopRecording();                      ← recording = false  
  
    tab.displayTime();                         ← Telefon::displayTime()  
    ...  
    return 0;  
}
```

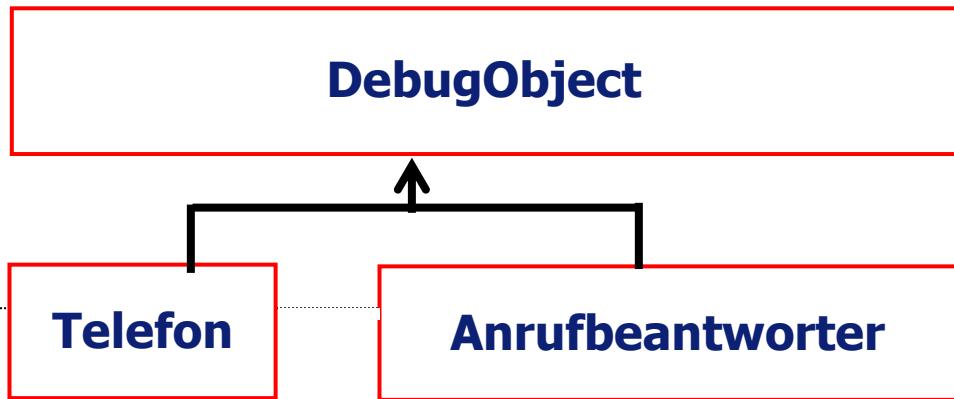
Quelle: Microconsult

Mehrfachvererbung

Mehrfachvererbungshierarchie

- Leite Telefon und Anrufbeantworter von Klasse `DebugObject` ab:

```
class DebugObject {  
    static int objCnt;  
    int objNr;  
  
protected:  
    DebugObject() { objNr = ++objCnt; }  
    ~DebugObject() { objCnt--; }  
  
public:  
    static void displayNrOfObjects()  
        { cout << "Existing objects: " << objCnt << endl; }  
    void displayDebugInfo()  
        { cout << "Number of this object: " << objNr << endl; }  
};
```



Mehrfachvererbungshierarchie

- Leite Telefon und Anrufbeantworter von Klasse `DebugObject` ab:

```
class Telefon : public DebugObject {};
```

```
class Anrufbeantworter : public DebugObject {};
```

```
int DebugObject::objCnt = 0;

int main( int argc, char* argv[] )
{
    Telefon *ptel = new Telefon;
    ptel->displayDebugInfo(); // Objekt Nummer 1

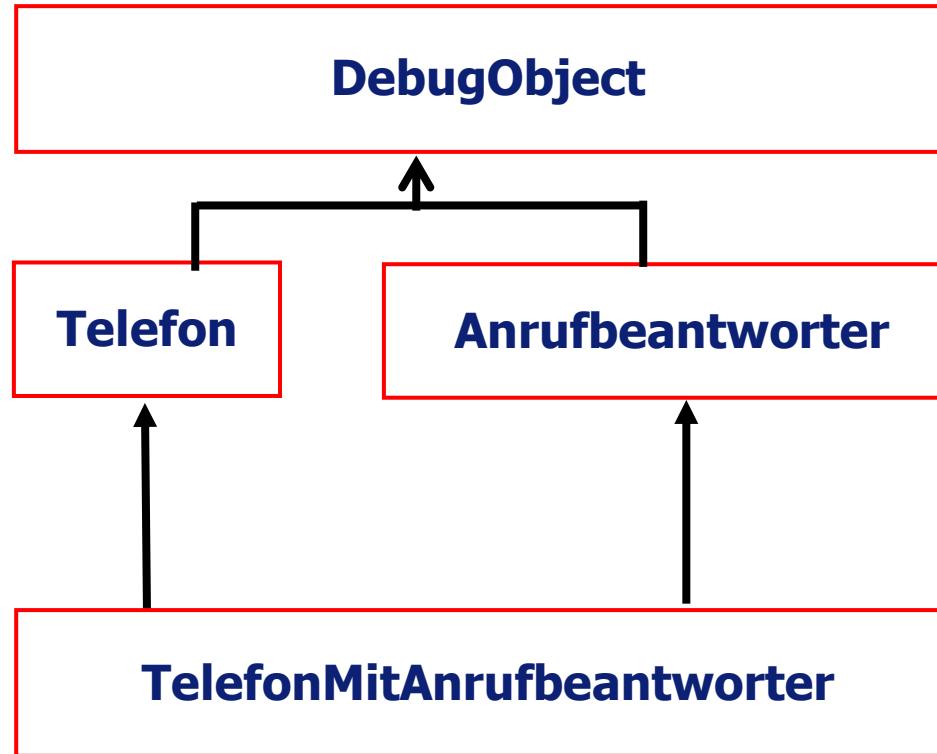
    Anrufbeantworter *pab = new Anrufbeantworter;
    pab->displayDebugInfo(); // Objekt Nummer 2

    delete ptel;
    DebugObject::displayNrOfObjects(); // Existierende Objekte: 1
    delete pab;
    return 0;
}
```

Mehrfachvererbung

Mehrfachvererbung

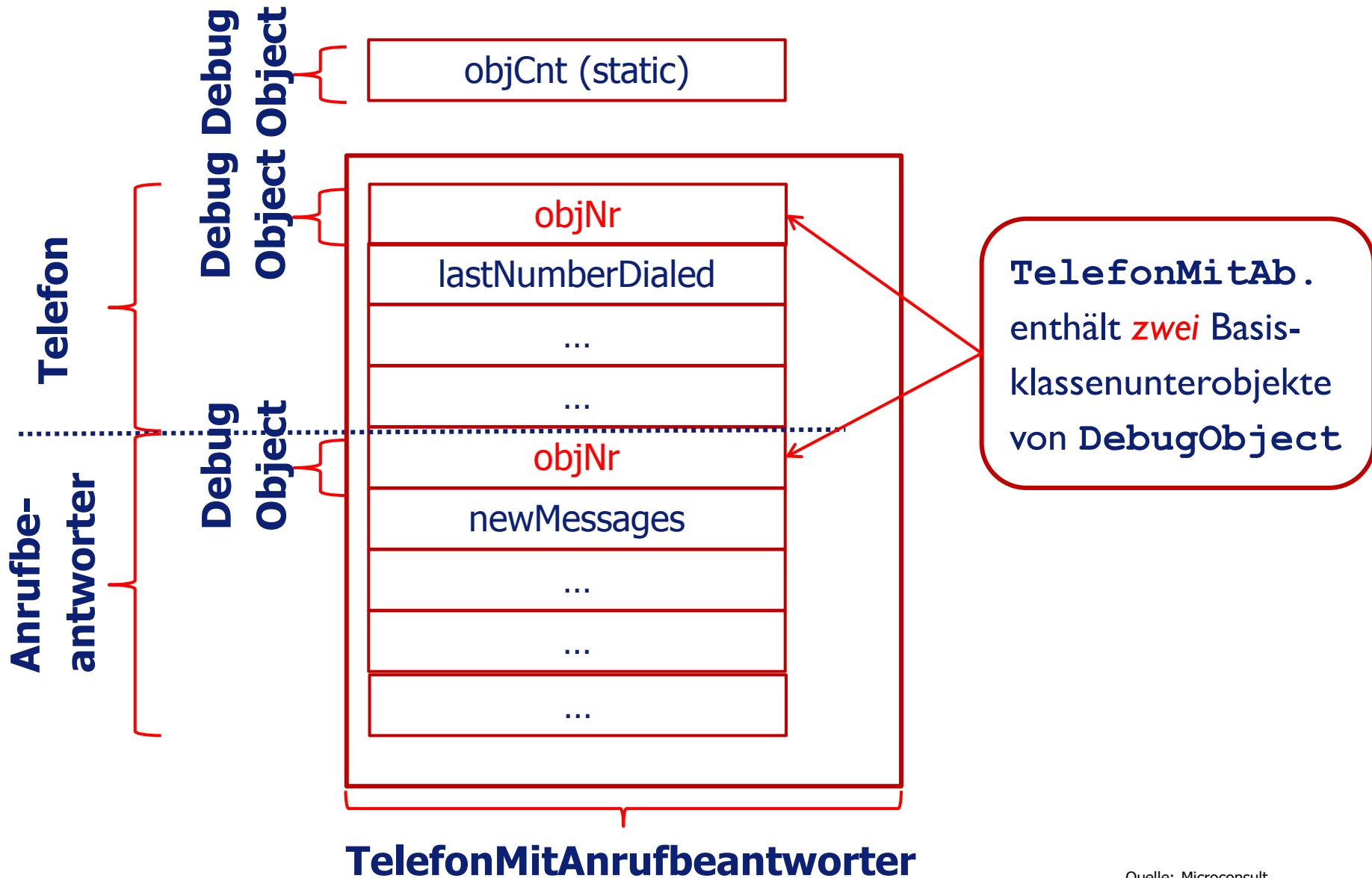
- Gesamte Ableitungshierarchie:



- Wie oft existieren die Elemente von **DebugObject** in einem Objekt **TelefonMitAnrufbeantworter**?

Mehrfachvererbung

Redundanzen



Quelle: Microconsult

Mehrfachvererbung

Mehrfachvererbungshierarchie

- Leite Telefon und Anrufbeantworter von Klasse `DebugObject` ab:

```
class Telefon : public DebugObject {};
```

```
class Anrufbeantworter : public DebugObject {};
```

```
class TelefonMitAnrufbeantworter : public Telefon,  
                                    public Anrufbeantworter {...};
```

```
int DebugObject::objCnt = 0;  
  
int main( int argc, char* argv[] )  
{  
    TelefonMitAnrufbeantworter tab;  
    // tab.displayDebugInfo();  
  
    DebugObject::displayNrOfObjects();  
  
    return 0;  
}
```

Fehler: mehrdeutig (da zwei Unter-
objekte aus `DebugObject` vorhanden)

Liefert 2! (da Konstruktor
zweimal aufgerufen wird)

Mehrfachvererbung: **virtual** ableiten

- Ableitung **virtual**: Bei Verwendung der Klasse in Mehrfachableitung wird Speicherplatz nur für *ein* Unterobjekt der Basisklasse angelegt

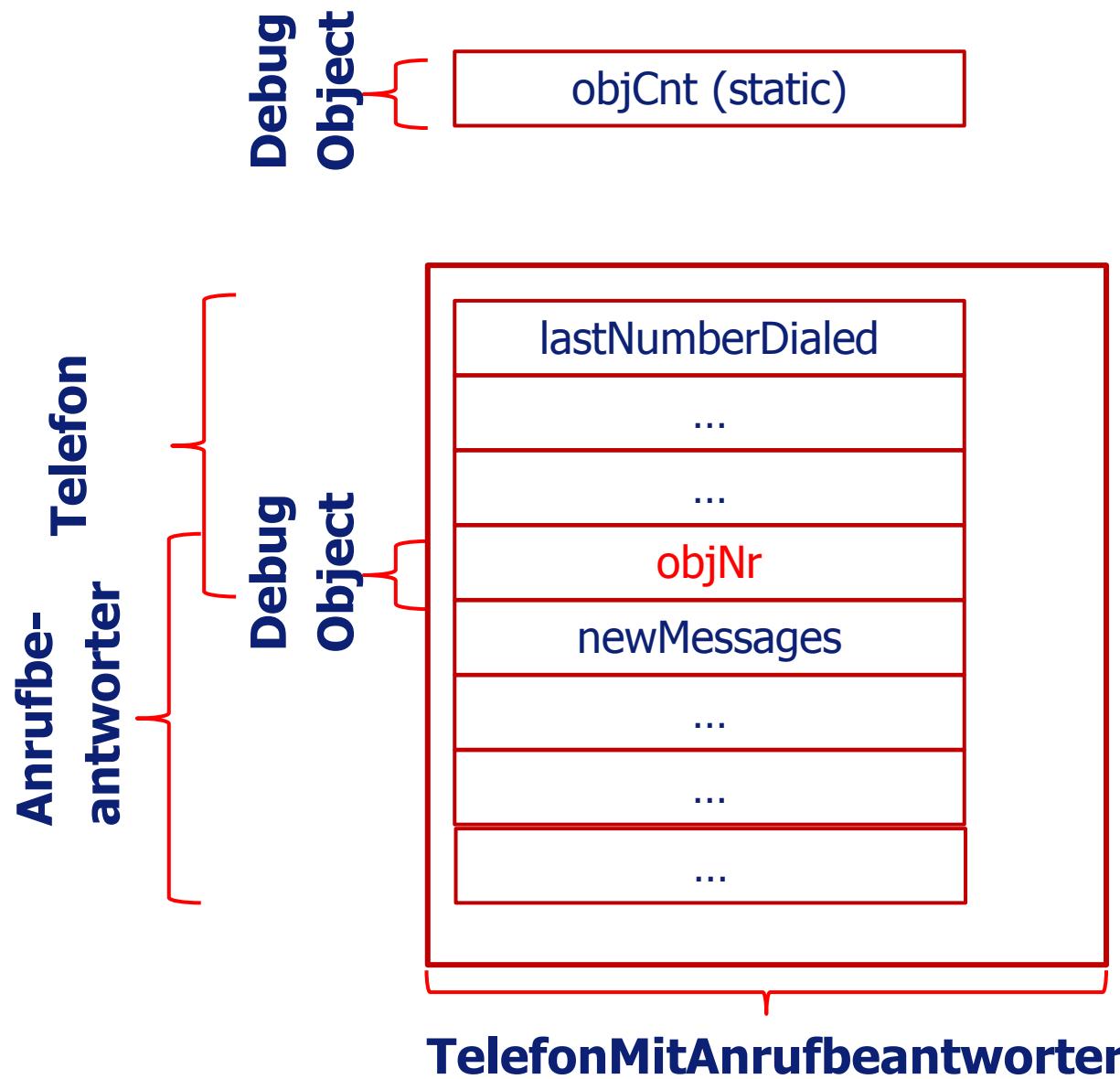
```
class Telefon : virtual public DebugObject {};
```

```
class Anrufbeantworter : virtual public DebugObject {};
```

(im nachfolgenden Beispiel muß **virtual** bei beiden Ableitungen stehen!)

Mehrfachvererbung

Mehrfachvererbung: virtual ableiten



Quelle: Microconsult

Mehrfachvererbung

Mehrfachvererbung: virtual ableiten

- Leite Telefon und Anrufbeantworter von Klasse `DebugObject` ab:

```
class Telefon : virtual public DebugObject {};
```

```
class Anrufbeantworter : virtual public DebugObject {};
```

```
class TelefonMitAnrufbeantworter : public Telefon,  
                                    public Anrufbeantworter {...};
```

```
int DebugObject::objCnt = 0;  
  
int main( int argc, char* argv[] )  
{  
    TelefonMitAnrufbeantworter tab;  
    tab.displayDebugInfo(); ← o.k., eindeutig  
  
    DebugObject::displayNrOfObjects(); ← o.k., liefert 1  
  
    return 0;  
}
```

Quelle: Microconsult

766

Mehrfachvererbung: Zusammenfassung

- Mögliche Folgen von Mehrfachvererbung:
 - Mehrdeutigkeitsprobleme
 - Notwendigkeit von virtueller Vererbung
- Virtuelle Vererbung bringen Kosten an Speicherplatz und Geschwindigkeit sowie Komplexität von Initialisierung und Zuweisung mit sich
 - Virtuelle Basisklassen sollten daher ggf. keine Daten enthalten
- Daher Vorsicht mit Mehrfachvererbung!

Fehlerbehandlung

Fehler

- Logische Fehler ↔ Debugger
- Syntax-Fehler ↔ Kompiler
- Laufzeitfehler ↔ Exceptions

Schlüsselwörter beim Exception Handling:

- **try**
- **catch**
- **throw**

Fehlerbehandlung

- Vorteil von C++ über C: verbesserte Möglichkeiten zur Fehlerbehandlung
- **Exception Handling** (Ausnahmebehandlung) erleichtert die Trennung von Fehlerbehandlungscode und übrigem Programm.
- Bei einer Aufrufhierarchie mehrerer Funktionen und Auftreten des Fehlers in der untersten Hierarchieebene müssen die mittleren Ebenen keinen Fehlerbehandlungcode beinhalten.
- Terminologie:
 - Funktion, in der der Fehler auftritt, **wirft** (throw) die Ausnahme
 - Funktion, die mit möglichem Fehler rechnet, beinhaltet Code, um evtl. geworfene Ausnahmen zu **fangen** (catch).

Exception Handling

Beispiel: Ohne Exception handling

```
double division( int z, int n )
{
    return z / n;
}

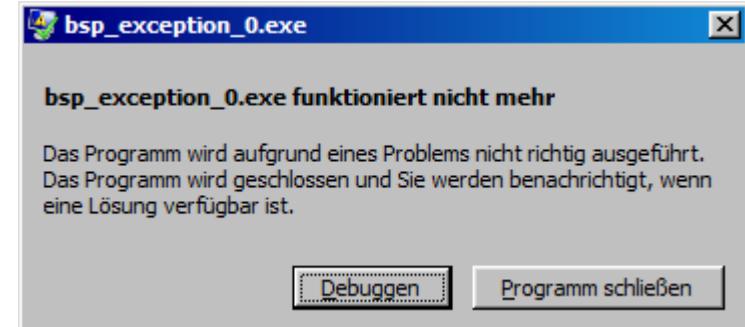
int main(int argc, char *argv[])
{
    int zaehler = 1, nenner = 0;
    double ergebnis;

    ergebnis = division( zaehler, nenner );

    cout << "Programm ist durchgelaufen!" << endl;

    return 0;
}
```

Ausgabe?



Exception Handling

Beispiel: Mit Exception handling

```
double division( int z, int n ) {  
    if ( n == 0 )  
        throw(n);  
    return z / n;  
}
```

Werfen der Ausnahme

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

```
    int zaehler = 1, nenner = 0;
```

```
    double ergebnis;
```

```
    try
```

```
{
```

```
        ergebnis = division( zaehler, nenner );
```

```
}
```

```
    catch(...)
```

```
{
```

```
        cout << "Fehler" << endl;
```

```
}
```

```
    cout << "Programm ist durchgelaufen!" << endl;
```

```
    return 0;
```

Versuch der Ausführung des Codes,
der möglicherweise Exception wirft

Abfangen der Ausnahme

Exception Handling

Beispiel: Mit Exception handling

```
double division( int z, int n ) {  
    if ( n == 0 )  
        throw(n);  
    return z / n;  
}  
  
int main(int argc, char *argv[]){  
    int zaehler = 1, nenner = 0;  
    double ergebnis;  
    try  
    {  
        ergebnis = division( zaehler, nenner );  
    }  
    catch(...)  
    {  
        cout << "Fehler" << endl;  
    }  
    cout << "Programm ist durchgelaufen!" << endl;  
    return 0;
```

Ausgabe?

Fehler

Programm ist durchgelaufen!

Drücken Sie eine beliebige
Taste . . .

Kein Laufzeitfehler mehr!

Exception Handling

Beispiel: Mit Exception handling

```
double division( int z, int n ) {  
    if ( n == 0 )  
        throw(n);  
    return z / n;  
}  
  
int main(int argc, char *argv[]){  
    int zaehler = 1, nenner = 0;  
    double ergebnis;  
    try  
    {  
        ergebnis = division( zaehler, nenner );  
    }  
    catch(...)  
    {  
        cout << "Fehler" << endl;  
    }  
    cout << "Programm ist durchgelaufen!" << endl;  
    return 0;
```

Jedes **try** hat ein **catch**
(direkt dahinter)

Exception Handling

Beispiel: Mit Exception handling

```
double division( int z, int n ) {  
    if ( n == 0 )  
        throw(n);  
    return z / n;  
}  
  
int main(int argc, char *argv[]){  
    int zaehler = 1, nenner = 0;  
    double ergebnis;  
    try  
    {  
        ergebnis = division( zaehler, nenner );  
    }  
    catch(...) {  
        cout << "Fehler" << endl;  
    }  
    cout << "Programm ist durchgelaufen!" << endl;  
    return 0;
```

3 Punkte: beliebige Ausnahme
wird abgefangen

Exception Handling

Beispiel: Mit Exception handling

```
double division( int z, int n ) {  
    if ( n == 0 )  
        throw(n);  
    return z / n;  
}  
  
int main(int argc, char *argv[]){  
    int zaehler = 1, nenner = 0;  
    double ergebnis;  
    try  
    {  
        ergebnis = division( zaehler, nenner );  
    }  
    catch( int e ) ← Exception e wird abgefangen  
    {  
        cout << "Nenner darf nicht " << e << " sein" << endl;  
    }  
    cout << "Programm ist durchgelaufen!" << endl;  
    return 0;  
}
```

Ausgabe?

Nenner darf nicht 0 sein
Programm ist durchgelaufen!
Drücken Sie eine beliebige
Taste . . .

Exception Handling

Beispiel: Mit Exception handling

```
double division( int z, int n ) {  
    if ( n == 0 )  
        throw(n);  
    return z / n;  
}  
  
int main(int argc, char *argv[]){  
    int zaehler = 1, nenner = 0;  
    double ergebnis;  
    try  
    {  
        ergebnis = division( zaehler, nenner );  
    }  
    catch( int e ) ←  
    {  
        cout << "Nenner darf nicht " << e << " sein" << endl;  
    }  
    cout << "Programm ist durchgelaufen!" << endl;  
    return 0;
```

Datentyp muß sichtbar sein;
catch(0) nicht möglich

Exception Handling

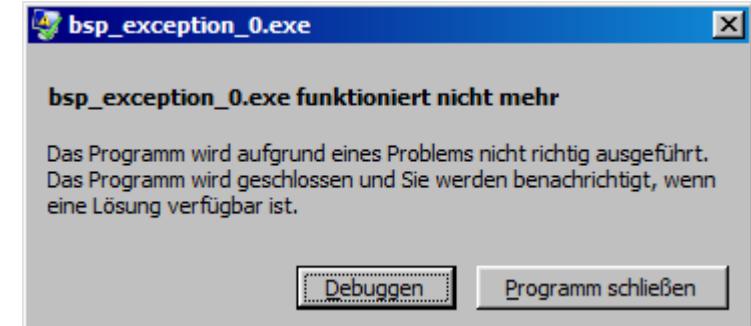
Beispiel: Mit Exception handling

```
double division( int z, int n ) {  
    if ( n == 0 )  
        throw( 0.0 );  
    return z / n;  
}
```

double wird geworfen

```
int main(int argc, char *argv[])  
{  
    int zaehler = 1, nenner = 0;  
    double ergebnis;  
  
    try  
    {  
        ergebnis = division( zaehler, nenner );  
    }  
    catch( int e )  
    {  
        cout << "Nenner darf nicht " << e << " sein" << endl;  
    }  
    cout << "Programm ist durchgelaufen!" << endl;  
    return 0;  
}
```

Ausgabe?



Nur ein int wird abgefangen;
kein double!

Exception Handling

Mehrere Arten von Exceptions abfangen

```
int main(int argc, char *argv[])
{
    int zaehler = 1, nenner = 0;
    double ergebnis;
    try
    {
        ergebnis = division( zaehler, nenner );
        cout << "Test" << endl; // wird im Exception-Fall nicht ausgeführt
    }
    catch( int e )
    {
        cout << "Nenner darf nicht " << e << " sein" << endl;
    }
    catch(...) ←
    {
        cout << "unbekannte Exception" << endl;
    }
    cout << "Programm ist durchgelaufen!" << endl;
    return 0;
}
```

Es wird ein catch-Block abgearbeitet;
daher erst die Spezialfälle abarbeiten,
zum Schluß ggf. den allgemeinen Fall

Falls keine int-Exception kommt

Eigene Exception-Klasse

Sinnvoll ist die Verwendung einer Fehlerklasse, die für die Beschreibung des Fehlers nützliche Informationen enthält:

```
class MyException
{
    private:
        string message;

    public:
        MyException (string s) : message(s) { }
        string getMessage() {return message; }
        void setMessage (string s) {message = s; }
};
```

Exception Handling

Eigene Exception-Klasse

```
#include "MyException.h"
```

```
double division( int z, int n ) {
    if ( n == 0 )
        throw MyException("Division durch Null");
    return z / n;
}

int main(int argc, char *argv[])
{
    int zaehler = 1, nenner = 0;
    double ergebnis;
    try
    {
        ergebnis = division( zaehler, nenner );
    }
    catch( MyException e )           ← Eigener Exception-Typ
    {
        cout << "Exception message: " << e.getMessage() << endl;
    }
    cout << "Programm ist durchgelaufen!" << endl;
    return 0;
}
```

Eigene Exception-Klasse

Ausgabe?

Exception message: Division durch Null

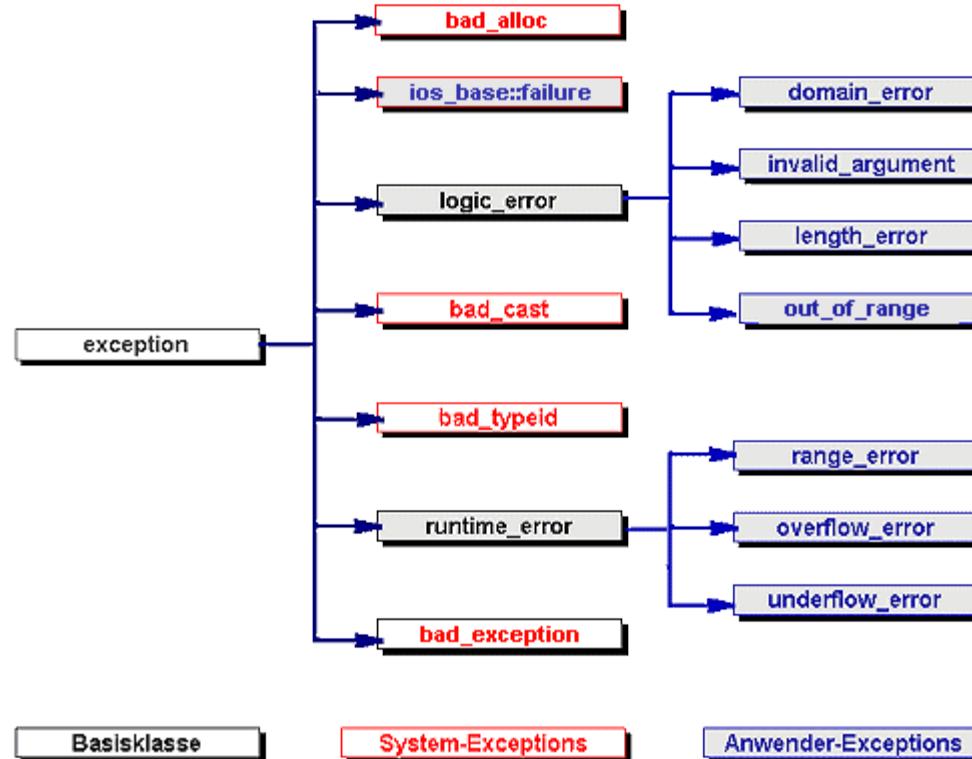
Programm ist durchgelaufen!

Drücken Sie eine beliebige Taste . . .

Exception Handling

Standard-Exception-Klassen

- Die C++ Standardbibliothek stellt Standard Exception Klassen bereit:



- Können z.B. als Ausgangspunkt eigener Exception-Klassen dienen
- Siehe z.B. http://www.cpp-tutor.de/cpp/le17/std_exception.html

Zusammenfassung

Übersicht (1)

Objektorientierte Programmierung (OOP):

- Einführung
- Objekt und Klasse
- Attribute, Methoden
- Geheimnisprinzip, Kapselung
- Vererbung
- Klassifikation
- Beziehungen zwischen Klassen
- Polymorphie

Schnellkurs C++:

- Einführung
- **Bekannte Sprachmittel:**
 - Variablen, Datentypen, Operatoren, Kontrollstrukturen, Arrays, Strukturen
- **Neue Sprachmittel:**
 - Referenzen
 - Funktionen: Vorgabeargumente, Überladung, Templates
 - Namensräume
 - Ein- und Ausgabe
 - Strings
 - Typumwandlung

Objektorientierte Programmierung mit C++:

- Klassen, Vererbung, Polymorphie, Mehrfachvererbung

Übersicht (2)

Objektorientierte Programmierung mit C++:

- **Klassen**
 - Klasse als Datentyp
 - Member: Instanzvariablen / -methoden, Klassenvariablen / -methoden, Konstruktor, Destruktor, this-Zeiger
 - Sichtbarkeit und Kapselung, Zugriffsspezifizierer, friends
 - Designempfehlungen: const, get/set,
 - Klassentemplates
 - Operatorüberladung für Klassenobjekte
 - Objektverwaltung: Erzeugen, Vergleichen, Kopieren, Auflösen, Komposition...
- **Vererbung**
 - Syntax und Einsatz
 - Basisklassen-Unterobjekt
 - Verdecken, Überschreiben, Überladen
 - Zugriffsmodifizierer, Zugriffsrechte

Übersicht (3)

Objektorientierte Programmierung mit C++:

- Polymorphie
 - Frühe und späte Bindung
 - Virtuelle Funktionen, virtueller Destruktor
 - Abstrakte Methoden
 - Abstrakte Klassen
- Mehrfachvererbung
- Fehlerbehandlung (exception handling)

Programmieren in C++

1. Einleitung: Warum objektorientierte Programmierung?

- Softwarekomplexität, Beispiel aus der Industrie
- Probleme und Grenzen der prozeduralen Programmierung, Beispiel
- Grundkonzepte und Vorteile der objektorientierten Programmierung

2. Grundbegriffe der objektorientierten Programmierung

1. Objekt
2. Klasse
3. Attribute, Methoden
4. Darstellung von Objekten und Klassen mittels UML
5. Prinzipien der objektorientierten Programmierung
6. Geheimnisprinzip, Kapselung, Zugriffsarten
7. Klassifikation
8. Beziehungen zwischen Klassen: Assoziation, Aggregation, Komposition
9. Vererbung
10. Polymorphie

Programmieren in C++

3. C++ -Schnellkurs

- Allgemeines zu C++
 - Entstehung, Entwurfsziele, Einsatz
- Das erste C++-Programm
 - Bibliotheken, Präprozessor-Direktiven, Eintrittsfunktion, Definition / Deklaration, Kommentare, Programmlayout
- Grundkonzepte C++
 - Variablen und Datentypen, Typumwandlung, Operatoren, Kontrollstrukturen, Funktionen, Arrays / Aufzählungen / Strukturen, Zeiger
- C++: neue Sprachmittel
 - Referenzen, Funktionen (Vorgabeargumente, Überladung, Templates, inline), Namensräume, dynamische Speicherverwaltung, Ein- / Ausgabe, Strings (Einführung), (Klassen)

4. Objektorientierte Programmierung mit C++: Klassen

5. Objektorientierte Programmierung mit C++: Vererbung

6. Objektorientierte Programmierung mit C++: Polymorphie

Das erste C++ Programm

C++ Programme bestehen aus:

- Präprozessor-Direktiven
- Eintrittsfunktion
- Anweisungen
- Deklarationen und Definitionen
- Kommentaren
- Funktionen (später)
- Namensräumen (neu in C++; später)

Grundkonzepte C++

- Variable
- Datentypen, Typumwandlung
- Operatoren
- Kontrollstrukturen
- Funktionen
- Arrays, Aufzählungen, Strukturen
- Zeiger

Die meisten dieser Grundkonzepte sind in C++ ähnlich wie in C. Einige Unterschiede (z.B. Typumwandlung) wurden erläutert.

C++: Neue Sprachmittel

- Referenzen
- Funktionen: Vorgabeargumente, Überladung, Templates, inline
- Namensräume
- Dynamische Speicherverwaltung
- Ein- und Ausgabe
- Strings
- Klassen (später)

Programmieren in C++

4. Objektorientierte Programmierung mit C++: **Klassen**

- Die Klasse als Datentyp
 - Von der Struktur zur Klasse, Definition, Bestandteile, Objekt, Warum Einheit Variable-Funktion?
- Member einer Klasse
 - Instanzvariablen und Instanzmethoden, this-Zeiger, Konstruktor und Destruktor, Klassenvariablen und Klassenmethoden, Code-Organisation: Implementierungs-, Headerdatei
- Sichtbarkeit und Kapselung
 - Klasse als Gültigkeitsbereich, Kapselung von Attributen, Zugriffsspezifizierer, friends
- Designempfehlungen
 - const Instanzmethoden, get/set Memberfunktionen
- Klassentemplates
- Operatorüberladung für Klassenobjekte
- Objektverwaltung
 - Erzeugen, Vergleichen, Kopieren, Ein-/Ausgeben, Auflösen
 - Komposition

Programmieren in C++

5. Objektorientierte Programmierung mit C++: Vererbung

- Einführung
- Syntax
- Beispiel
- Basisklassen-Unterobjekt
- Verdecken, Überschreiben, Überladen
- Zugriffsmodifizierer und Zugriffsrechte
- Vererbung, Komposition oder Nutzung?

6. Objektorientierte Programmierung mit C++: Polymorphie

7. Objektorientierte Programmierung mit C++: Mehrfachvererbung

8. Exceptions

Sources

- HS Esslingen: <http://www.it.hs-esslingen.de/~harms/oop/>
- Bittel: <http://www-home.fh-konstanz.de/~bittel/prog2/prog2.htm>
- RRZN: <http://www.rrzn.uni-hannover.de/cplus.html>
- Microconsult: Unterlagen, Objektorientiertes Programmieren mit C++, MicroConsult GmbH, München