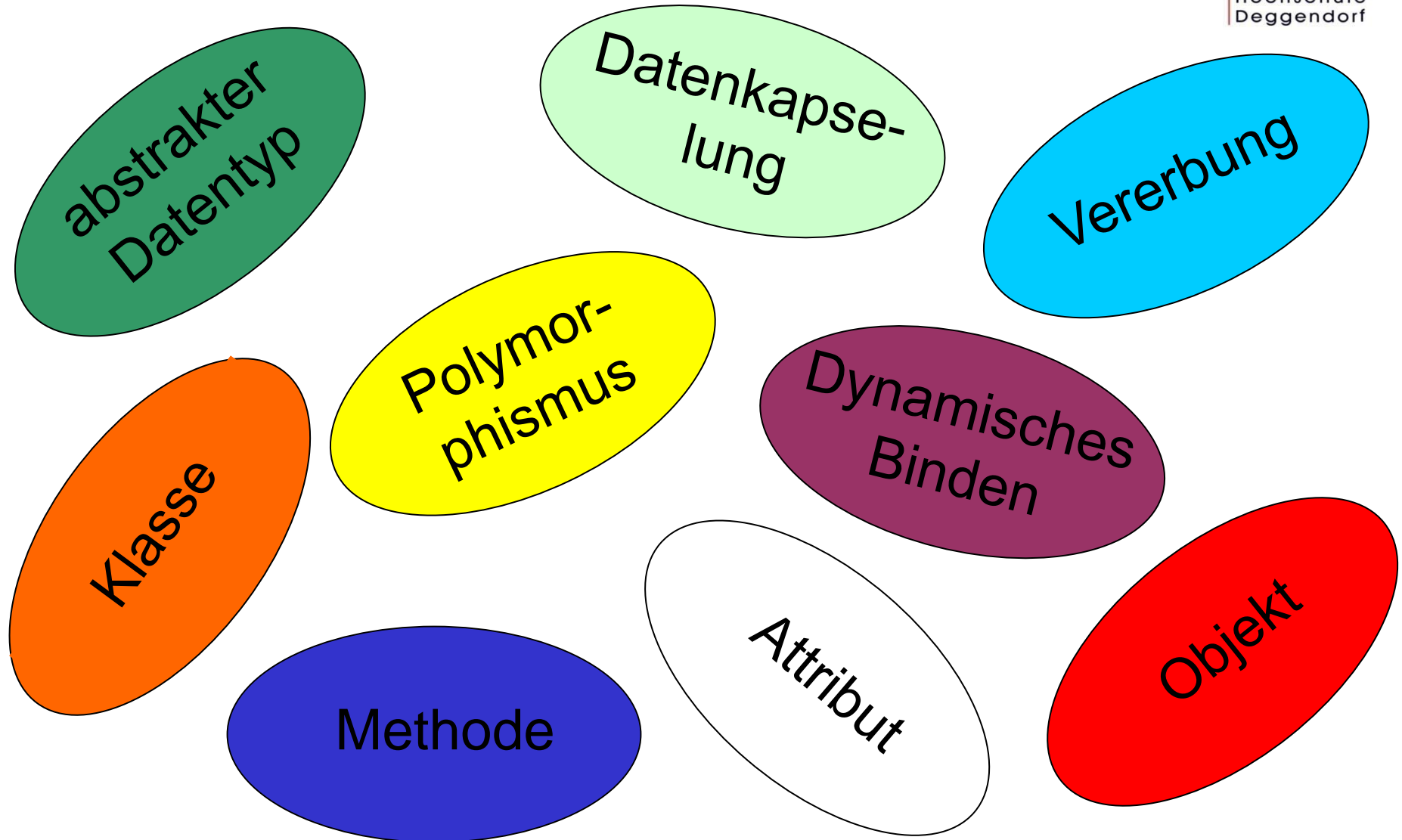


# Objektorientierte Programmierung

## Grundbegriffe Teil 1

Prof. Dr. Peter Jüttner

# Objektorientierung - Grundbegriffe



## Anmerkungen zu C++

- **C++**
  - **Obermenge der Sprache C<sup>\*)</sup>, d.h. ein C-Programm läßt sich mit einem C++-Compiler übersetzen und ist lauffähig**
  - **C- und C++-Teile können in einem Programm „gemischt“ werden, d.h. „alte“ C-Funktionen können in C++-Programmen weiterverwendet werden. Dies sollte bei neuen Programmen aber vermieden werden.**

<sup>\*)</sup> es gibt nur ganz wenige Details in C, die nicht in C++ übernommen wurden.

## Anmerkungen zu C++

- **Objektorientierte Erweiterung von C**
  - ➔ **Unsauberkeiten von C weitgehend auch in C++**
  - ➔ **in C++ kann auch prozedural/funktional programmiert werden**
  - ➔ **in C++ kann „rein“ objektorientiert programmiert werden**

## Anmerkungen zu C++

- **Objektorientierte Erweiterung von C**
  - ➔ **C++ hat Elemente, die der reinen Lehre der Objektorientierung widersprechen**
  - ➔ **C++ erlaubt neue Unsauberkeiten**
  - ➔ **C++ relativ „nahe“ an Java und C# ➔ Umstieg relativ einfach**

## Vorbemerkungen

- **Grundbegriffe werden zunächst vom Aspekt der Theorie der Objektorientierung eingeführt**
- **Die Begriffe Datenkapselung und Abstrakte Datentypen gab es schon vor der Objektorientierung**
  - ➔ **grundlegende Begriffe der Objektorientierung**
  - ➔ **auch in der funktionalen Programmierung nützlich und anwendbar**

## Grundbegriffe – Datenkapselung

- **„Alte“ Idee der SW Technik**
- **konsequent umgesetzt in der objektorientierten Programmierung, d.h. Entwicklungstools (Sprachen, Compiler) unterstützen die Datenkapselung (Synonyme: Geheimnisprinzip, Data Encapsulation, Information Hiding)**

## Grundbegriffe – Datenkapselung

### Motivation

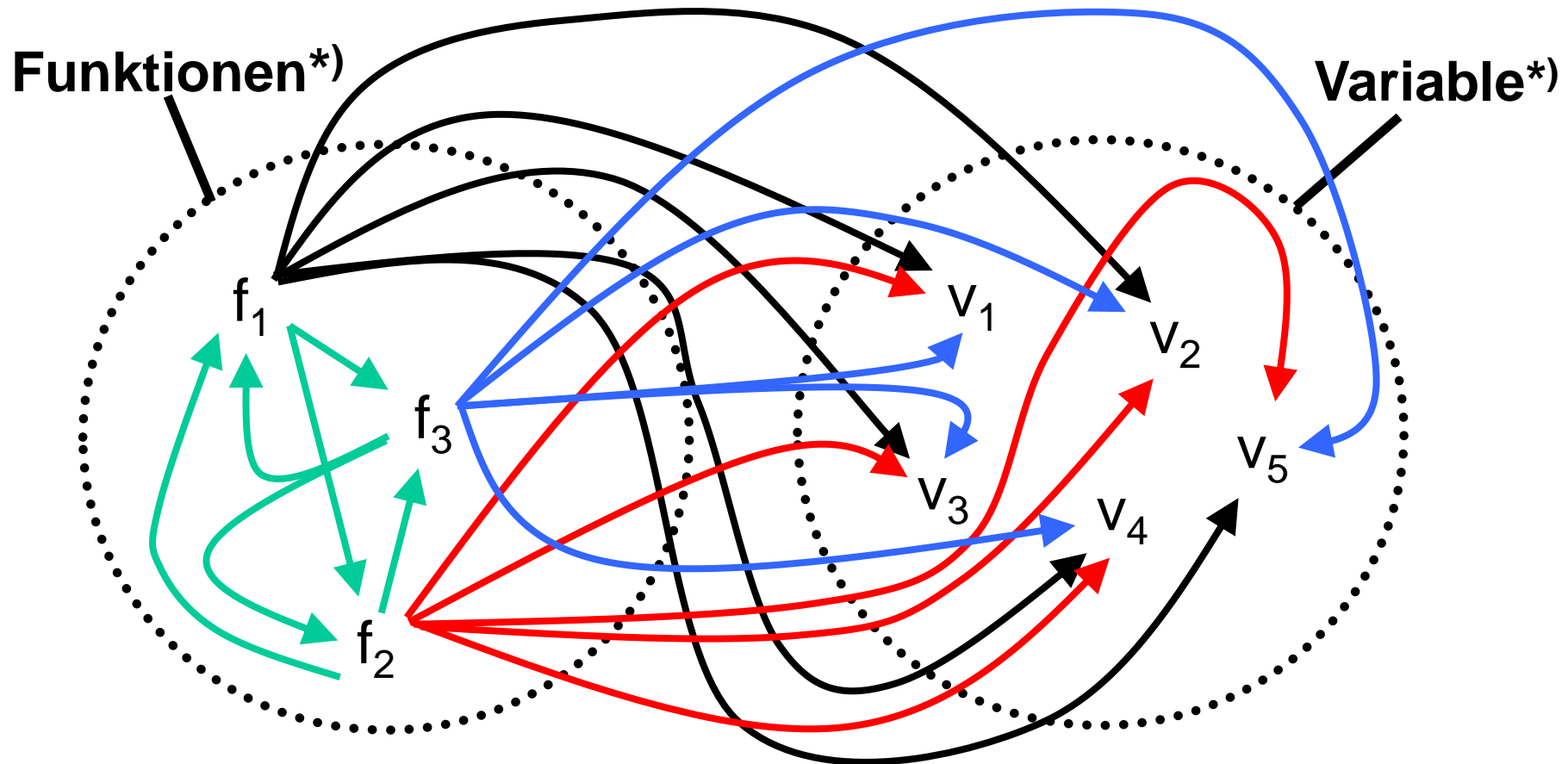
**In der prozeduralen/funktionalen Programmierung sind Variablen in der Regel in ihrem Kontext global und von allen Prozeduren des Kontext aus manipulierbar**

- erhöhte Komplexität**
- vergrößert Abhängigkeiten**
- Änderungen schwierig („jeder kann auf alles zugreifen“, jeder ist potentiell von allem abhängig)**



# Grundbegriffe – Datenkapselung

## Prozedurale Programmierung ohne Datenkapselung



\*) im Hauptprogramm

→ Datenzugriff bzw. Funktionsaufruf

## Grundbegriffe – Datenkapselung

- **Reduzierung / Minimierung von Abhängigkeiten durch Datenkapselung**
- **Lesende oder schreibende Zugriffe auf Daten möglichst lokal nur innerhalb einer Einheit (in der OOP Klasse) erfolgen**
- **Außerhalb einer Einheit sollen Daten (insbesondere ihre Realisierung, in der OOP die Attribute einer Klasse) nach außen nicht bekannt sein.**

## Grundbegriffe – Datenkapselung

- **in prozeduralen / funktionalen Sprachen (C, Pascal) in der Verantwortung des Programmierers**
- **Verschiedene objektorientierte Sprachen setzen die Datenkapselung unterschiedlich um.**

# Grundbegriffe – Datenkapselung

- **Nutzen ??**

## Grundbegriffe – Datenkapselung

### Vorteile\*)

- **So lange die Schnittstelle einer Einheit (Klasse) „nach außen“ nicht geändert wird, kann die Implementierung der Einheit geändert werden ohne Einfluss auf andere Teile**
  - ➔ **Änderungen bleiben lokal**
  - ➔ **Änderungen betreffen andere Programmteile weniger**

\*) Datenkapselung ist unabhängig von Programmierparadigma immer sinnvoll!

## Grundbegriffe – Datenkapselung

### Vorteile\*)

- **Der Verwender einer Einheit (i.e. Aufrufer der Funktionen der Einheit (Klasse) muss nur die Schnittstelle, nicht aber die Implementierung kennen oder gar verstehen**
- **Reduktion von Komplexität (weniger oder nur noch definierte Möglichkeiten auf andere Elemente in der Software zuzugreifen)**
- **Verbesserte Testbarkeit der Software**

\*) Datenkapselung ist unabhängig von Programmierparadigma immer sinnvoll!

## Grundbegriffe – Datenkapselung

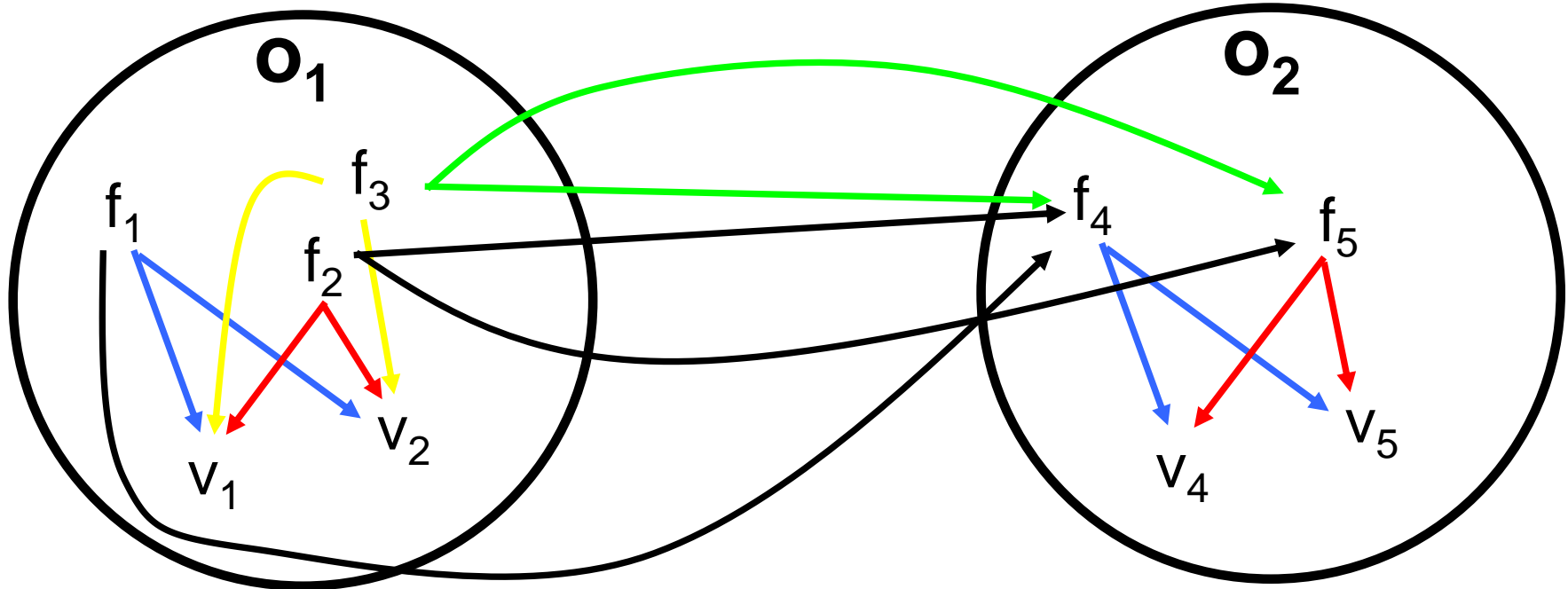
### Nachteile

- **geringfügig höhere Ressourcenverbrauch\*)  
(Laufzeit, Speicherplatz)**
- **zusätzlicher Codieraufwand  
(Zugriffsfunktionen)**

\*) gilt tendenziell für objektorientierte Programmierung

## Grundbegriffe – Datenkapselung

### Objektorientierte Programmierung mit Datenkapselung



Objekte  $O_1$  der Klasse  $K_1$ ,  $O_2$  der Klasse  $K_2$

$K_1$  hat die Methoden  $f_1$ ,  $f_2$ ,  $f_3$  und die Attribute  $v_1$ ,  $v_2$

$K_2$  hat die Methoden  $f_4$ ,  $f_5$  und die Attribute  $v_4$  und  $v_5$

Der Zugriff auf Attribute erfolgt ausschließlich über Methoden der eigenen Klasse



## Motivation

**Zum Schluss dieses Abschnitts ...**

**Noch Fragen ??**

## Abstrakter Datentyp

# Was ist ein Datentyp ?

# Abstrakter Datentyp

## Vorbemerkungen

- **Datentypen werden definiert durch ihre Struktur und den Operationen, die auf dem Datentyp arbeiten**
- **Operationen werden definiert durch ihre Schnittstelle (Signatur), ihre Spezifikation und ihre Implementierung**

# Abstrakter Datentyp

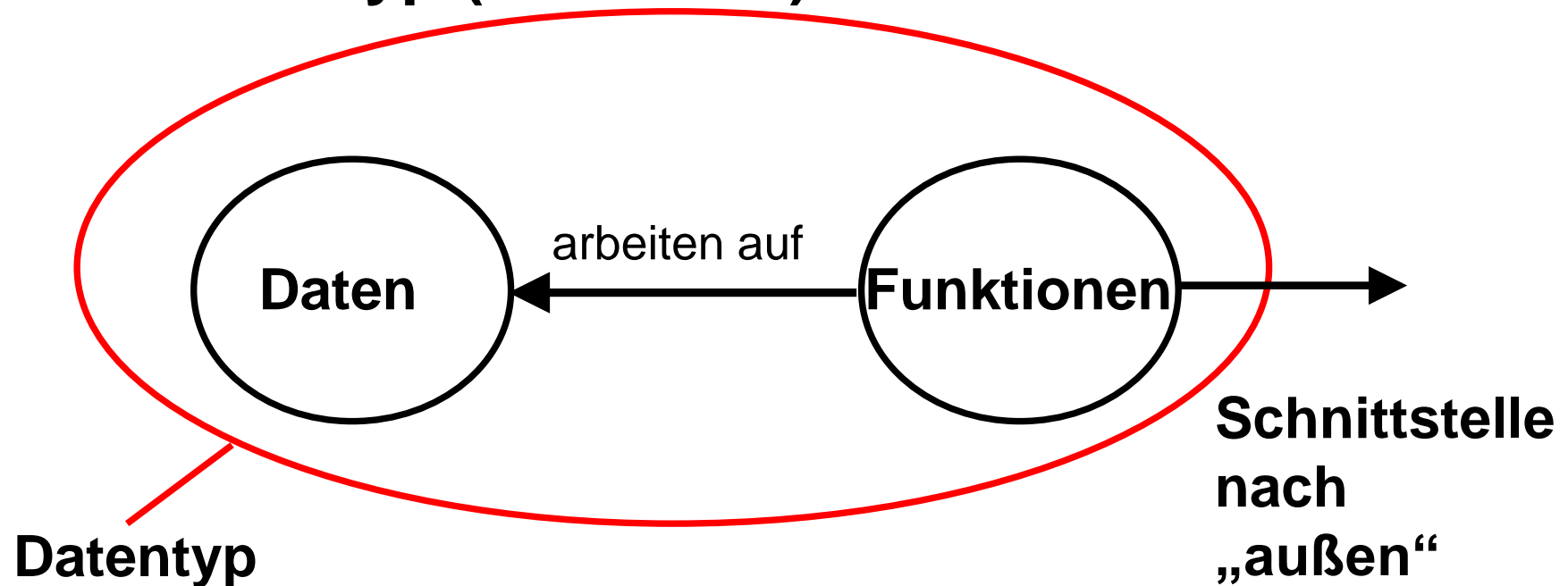
## Vorbemerkungen

- **Einfache Datentypen und deren Operationen wurden bereits eingeführt, z.B. char, short, int, long, float, +, -, \*, ...**
- **Komplexe Datentypen können in C aus einfachen den einfachen Datentypen, mittels Pointern, Strukturierung und Arrays gebildet werden, z.B. struct s ..., int feld[10]**

# Abstrakter Datentyp

## Motivation

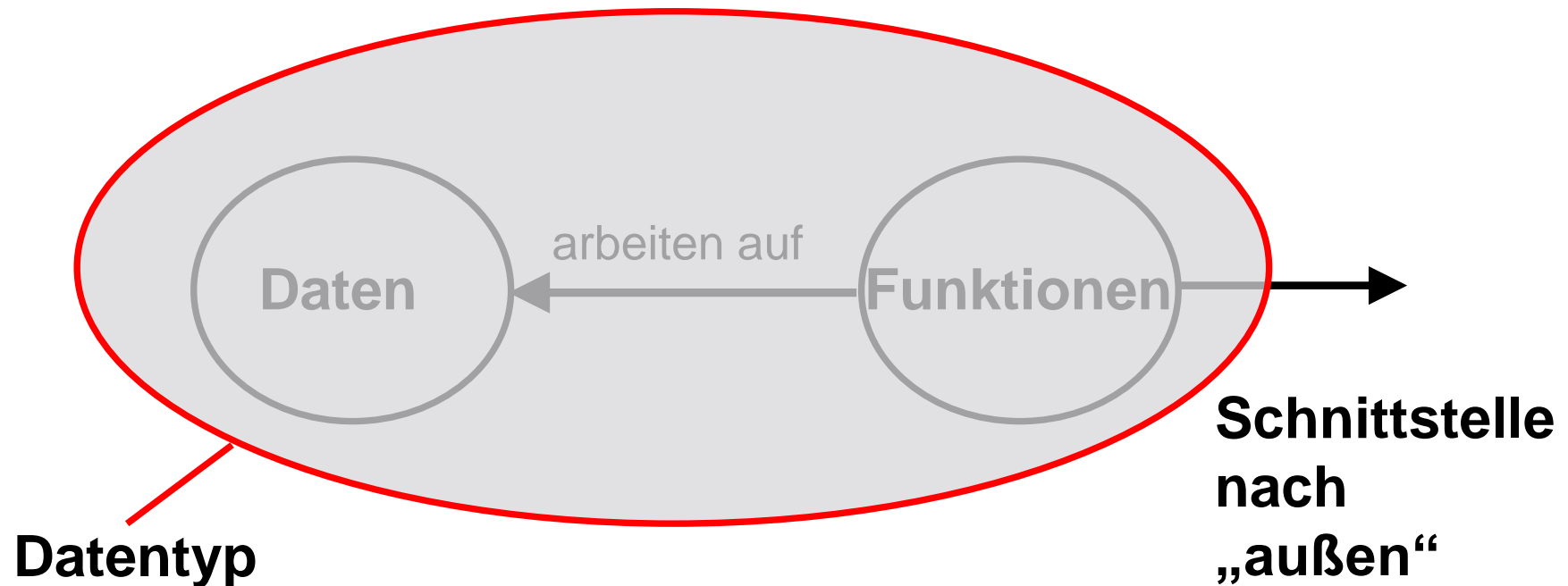
- Wir betrachten im Sinn der Objektorientierung Daten und die darauf operierenden Funktionen als Datentyp (→ Klasse)



# Abstrakter Datentyp

## Motivation

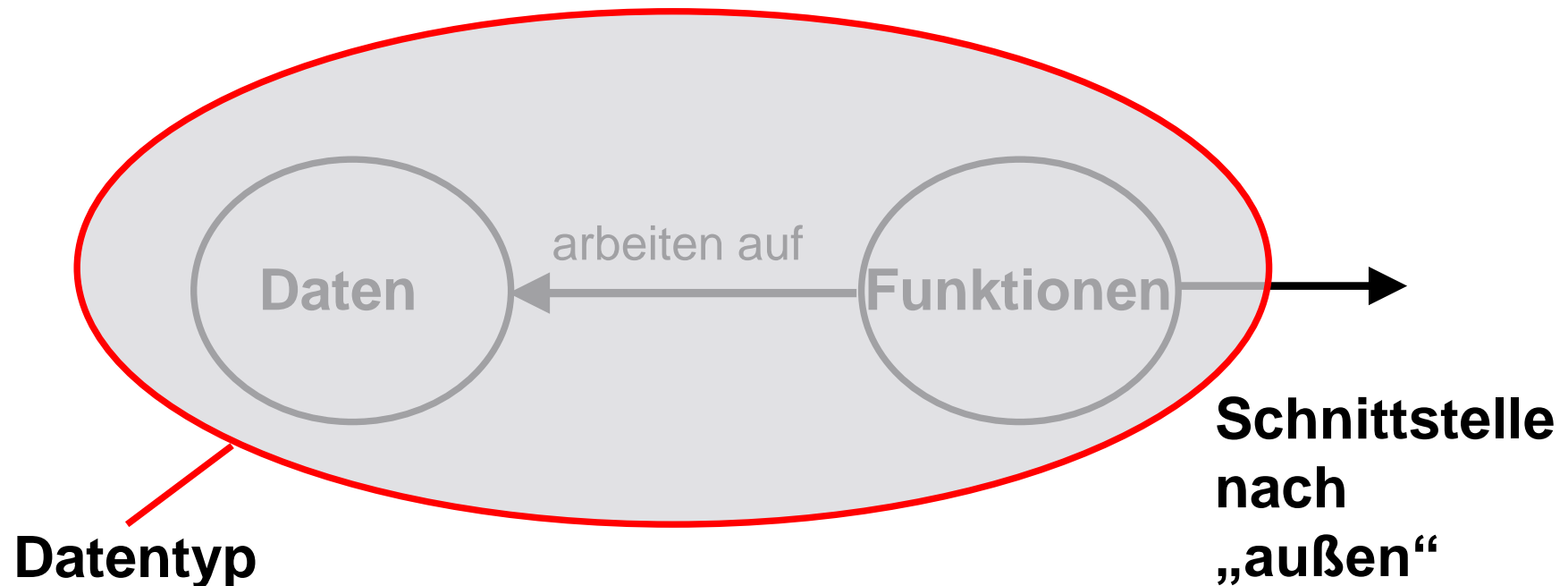
- Was passiert, wenn nur noch die Schnittstelle nach „außen“ bekannt ist ?



# Abstrakter Datentyp

## Motivation

- Was muss ich dann kennen, wenn ich mit den Funktionen des Datentyps arbeiten möchte?



# Abstrakter Datentyp

## Motivation

- **Wiederverwendung**
  - ➔ **Trennung von Schnittstelle und Implementierung**
  - ➔ **Wiederverwender kennt nur die Schnittstelle eines Datentyps\*)**
  - ➔ **unabhängig von der Implementierung, falls Schnittstelle konstant**

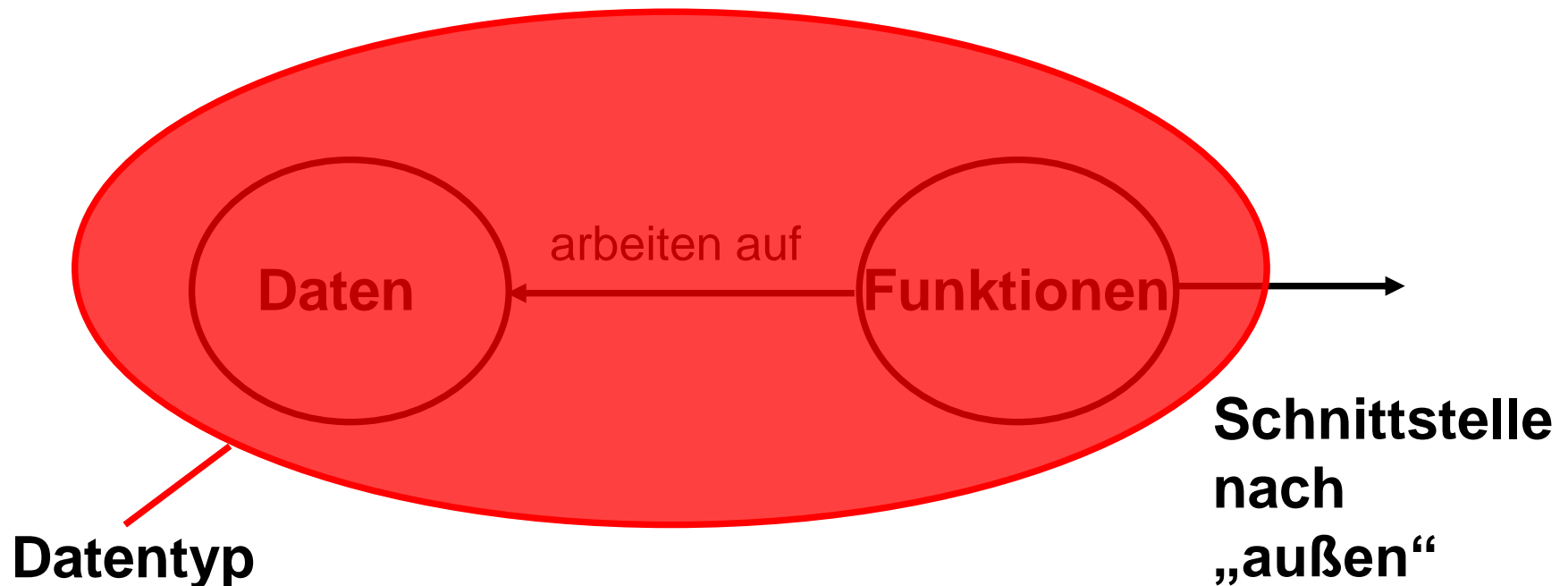
\*) Der Implementierer des Datentyps muss sich natürlich über die interne Struktur des Datentyps Gedanken machen. Dies kann abhängig von der Umgebung sein.



# Abstrakter Datentyp

## Motivation

- Sichtweise des Nutzers eines Datentyps



# Übung Abstrakter Datentyp

## Spezifikation Abstrakter Datentyp

- was ist ein abstrakter Datentyp ?
  - wofür wird er eingesetzt ?
  - was hat er mit Objektorientierter Programmierung zu tun?
  - Beispiele ?
- 
- Bearbeiten Sie das Thema gruppenweise!
  - Bereiten Sie eine kurze Präsentation zu Ihrem Thema vor!



## Abstrakter Datentyp

### Spezifikation Abstrakter Datentyp

- Name, ggf. mit Parametern
- Signatur, d.h. Schnittstellen der Funktionen des Datentyps
- Eigenschaften der Funktionen sog. Axiome (Gesetze),
- ggf. Vorbedingungen, unter denen partielle Funktionen aufgerufen werden können

# Abstrakter Datentyp

## Beispiel Datentyp PUNKT:

**Typ:** Punkt

**Funktionen:**

erzeuge:  $\text{real} \times \text{real} \rightarrow \text{Punkt}$

get\_x:  $\text{Punkt} \rightarrow \text{real}$

get\_y:  $\text{Punkt} \rightarrow \text{real}$

ist\_ursprung:  $\text{Punkt} \rightarrow \text{bool}$

verschiebe:  $\text{Punkt} \times \text{real} \times \text{real} \rightarrow \text{Punkt}$

skaliere:  $\text{Punkt} \times \text{real} \rightarrow \text{Punkt}$

abstand:  $\text{Punkt} \times \text{Punkt} \rightarrow \text{real}$



Name der Funktion



Signatur (Abbildungsverhalten) der Funktion, hier Eingabe 2 Punkte,  
Resultat vom Typ real

# Abstrakter Datentyp

## Beispiel Datentyp PUNKT:

Typ: Punkt

### Axiome:

$\text{get\_x}(\text{erzeuge}(x, y)) = x$

$\text{get\_y}(\text{erzeuge}(x, y)) = y$

$\text{ist\_ursprung}(\text{erzeuge}(x, y)) \Leftrightarrow x = 0 \text{ und } y = 0$

$\text{verschiebe}(\text{erzeuge}(x, y), a, b) = \text{erzeuge}(x+a, y+b)$

$\text{skaliere}(\text{erzeuge}(x, y), a) = \text{erzeuge}(x*a, y*a)$

$\text{abstand}(\text{erzeuge}(x, y), \text{erzeuge}(z, w)) = \sqrt{(x - z)^2 + (y - w)^2}$

### Vorbedingungen:

keine

## Abstrakter Datentyp

### Anmerkungen

- Funktionen, die ein Element des Datentyps erzeugen, heißen Konstruktoren (im Beispiel erzeuge)
- Jedes Element des Datentyps wird ausschließlich mittels eines Konstruktors erzeugt

## Abstrakter Datentyp

### Anmerkungen

- **Selektorfunktion** zerlegen ein Element in seine Bestandteile (das Element bleibt dabei erhalten)  
(im Beispiel `get_x`, `get_y`)

# Übung

## Abstrakter Datentyp

### Beschreibung:

Abstrakte Datentypen dienen dazu, die Schnittstelle und das Verhalten eines Datentyps von seiner Implementierung zu abstrahieren

### Aufgabe:

1. Definieren Sie einen Abstrakten Datentyp **Natürliche Zahl**, mit folgenden Funktionen:

- Null
- Nachfolger
- Vorgänger
- Addiere
- Multipliziere
- kleiner

Legen Sie zunächst die Signatur des Datentyps fest und anschließend die Axiome, die das Verhalten beschreiben.

2. Definieren Sie einen Abstrakten Datentyp Ihrer Wahl

### Zeit:

30 min, arbeiten Sie ggf. mit einem Partner





## Abstrakter Datentyp

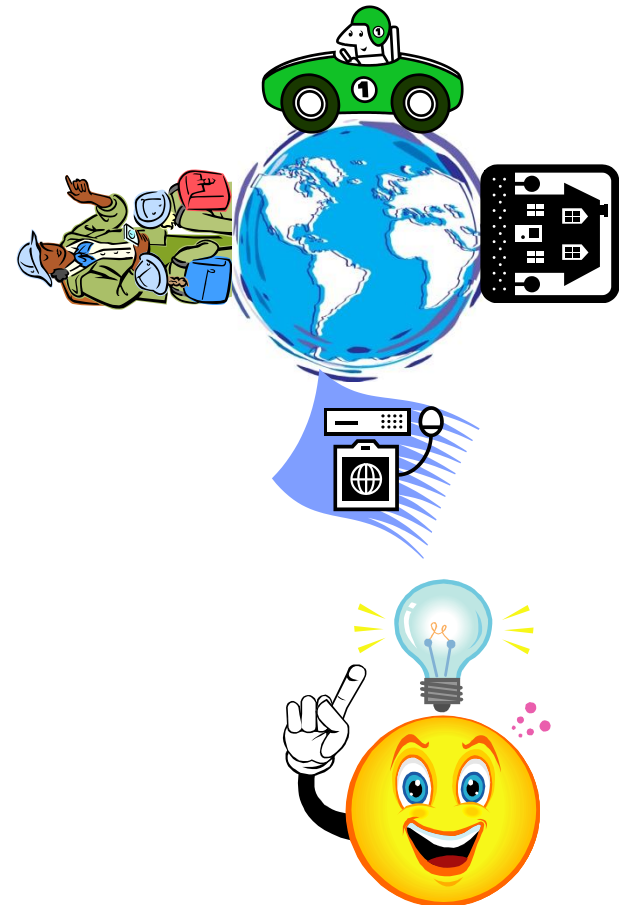
**Zum Schluss dieses Abschnitts ...**

**Noch Fragen ??**

# Objekt

## Motivation

- was ist ein Objekt ?
- welche Objekte kennen Sie ?



# Objekt

## Motivation

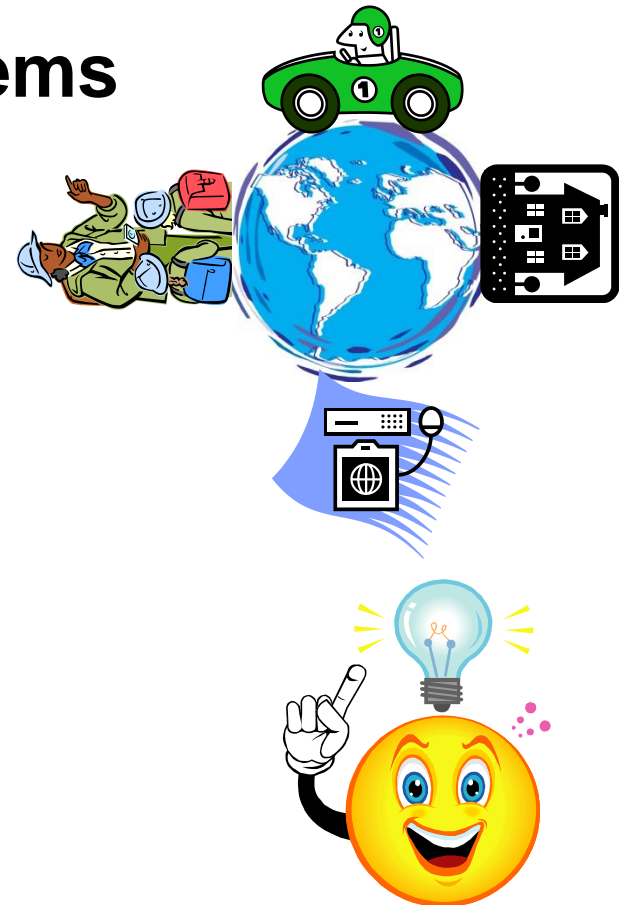
- unsere Alltagswelt besteht aus **Objekten** (Menschen, Autos, Häuser, Computer, Hörsäle)
- diese Alltagsobjekte sind z.T. aktiv handelnd (Personen), z.T. passiv (Häuser)
- Alltagsobjekte kommunizieren miteinander



# Objekt

## Motivation

➔ **Idee: Zur Lösung eines Problems mittels Software werden die Objekte, die in der Problemwelt eine Rolle spielen in der Software als Objekte dargestellt.**



## Objekt

- **zentrales, grundlegendes Element der Objektorientierten Software Entwicklung**
- **bildet ein Objekt der „realen“ Welt abstrahiert in die Software ab**
- **Abstraktion heißt Weglassen nicht benötigter Details**
- **Abstraktion erhält nur die für die Lösung relevanten Eigenschaften des Objekts**

## Objekt

- **ein Objekt hat in der Regel einen (eindeutigen) Namen**
- **ein Objekt hat ein definiertes Verhalten (Schnittstelle)**
- **Objekte können aus anderen Objekten aufgebaut sein**
- **Objekte können temporär oder permanent vorhanden sein**

## Objekt

- **das Objekt der „realen“ Welt kann gegenständlich sein, z.B. Auto, Mensch, Baum, Motor**
- **das Objekt der „realen“ Welt kann virtuell sein, z.B. Lebensversicherung, Konto, Gehalt**

## Objekt

- **Objekte werden in allen Phasen der SW Entwicklung verwendet**
  - **zur Darstellung der Anforderungen (Requirements)**
  - **zur Analyse der Problemstellung (Analyse)**
  - **zum Design der Lösung (Software-Design)**
  - **zur Implementierung der Lösung (Codierung)**



## Objekt

- „gleichartige“ Objekte werden zu Klassen zusammengefasst, d.h. ihre Eigenschaften werden durch Klassen beschrieben
- ➔ siehe Abschnitt „Grundbegriffe - Klassen“
- Analogie Variable in funktionaler Programmierung

# Objekt

## Beispiele:

### 1. Programm zur Verwaltung von Patientendaten in Arztpraxen

- Programm soll relevante Daten der Patienten erfassen und verwalten



Person Hans Meier bzw.  
Patient (Objekt) in der realen Welt

Abstraktion



Person Hans Meier bzw.  
Patient (Objekt) in der SW

# Objekt

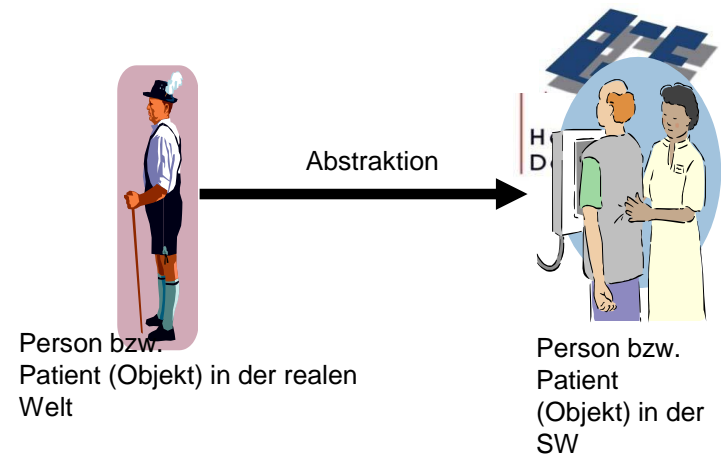
## Beispiele 1: Person

- Hans Meier
- geboren am 25.1.1958
- Häherweg 3, München
- graue Haare
- Schuhgröße 45
- Schulabschluss Abitur
- Körpergröße 180
- Konfektionsgröße 52
- Krankenkasse  
Versicherungskammer
- verheiratet, 3 Kinder
- Hobbies Sport
- Beruf Informatiker
- ...

Abstraktion

## Patient in der SW

- Hans Meier
- geboren am 25.1.1958
- Häherweg 3, München
- Krankenkasse  
Versicherungskammer



# Objekt

## Beispiele:

- ### 2. Programm zur Steuerung eines Verbrennungsmotors
- Programm soll Zündung und Kraftstoffeinspritzung steuern



zu entwickelnder Motor (Objekt) in der realen Welt

Abstraktion



zu entwickelnder Motor (Objekt) in der SW

# Objekt

## Beispiele 2:

### Motor

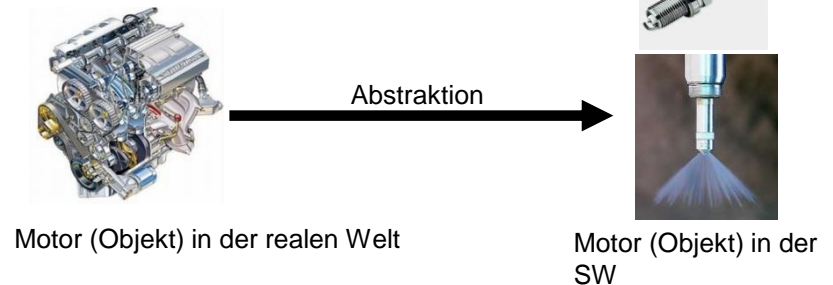
- neuer Motor für Fzg. X
- Zylinderzahl 8
- Anzahl Zündkerzen pro Zylinder 2
- Anzahl Einspritzdüsen 1
- Hubraum 4000cm<sup>3</sup>
- Ventilanzahl pro Zylinder 4
- Zylinderwanddicke 15mm
- Ölsorte 10W-40
- max. Leistung 250KW
- max. Drehmoment 550NM
- ...

Abstraktion



### Motor in der SW

- neuer Motor für Fzg. X
- Zylinderzahl 8
- Anzahl Zündkerzen pro Zylinder 2
- Anzahl Einspritzdüsen 1



# Objekt

## Beispiele:

### 3. Bankgirokonto

- Programm soll Kontodaten verwalten



Abstraktion



Art	Name	Datum	Kontostand
Kasse	Haushaltskasse	29.12.2006	823,09
Bank	Bank	31.12.2006	4.745,52
Bank	Postbank	22.12.2006	9.248,17

Konto (Objekt) in der realen Welt

Konto in der SW

# Objekt

## Beispiele 3:



Abstraktion



Konto (Objekt) in der realen Welt

Konto in der SW

## Konto

- Nummer 120123
- Inhaber (Hans Meier, Häherweg 3, München)
- Saldo 1500 €
- Überziehungslimit 5000€
- Verzinsung 1,5%
- ...

Abstraktion

## Konto in der SW

- Nummer 120123
- Inhaber (Hans Meier, Häherweg 3, München)
- Saldo 1500 €
- Überziehungslimit 5000€
- Verzinsung 1,5%
- ...

## Objekt

- **„komplexe“ Objekte der realen Welt erfordern eine „hohe“ Abstraktion**
  - **viele Details sind für die zu entwickelnde Software nicht relevant**
    - **können bzw. müssen daher weggelassen werden (Beispiele Patient, Motor)**
- **im Beispiel Konto müssen viele (alle) Details in der Software modelliert werden (das Konto ist bereits abstrakt)**



## Objekt

# Implementierung in C++

**→ siehe Abschnitt „Grundbegriffe - Klassen“**

# Übung

## Objekt

### Beschreibung:

Objekte im Sinn der Objektorientierten Programmierung dienen dazu, Objekte der realen Welt in Software zu modellieren. Dazu werden die Eigenschaften realer Objekte geeignet abstrahiert.

### Aufgabe:

In einem objektorientiert entwickelten Programm sollen Immobilien als Objekte verwaltet werden.

Überlegen Sie generell Daten, die zu einer Immobilie gehören und definieren Sie, welche Daten auch in der Software Verwendung finden sollen.  
Begründen Sie Ihre Wahl.

### Zeit:

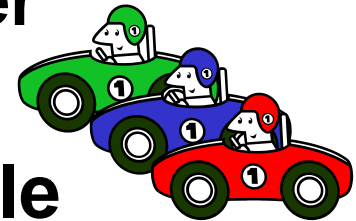
10 Minuten, arbeiten Sie ggf. zusammen mit einem Partner



# Klasse

## Motivation

- in der Regel werden in einem Programm mehrere oder viele gleichartige oder sehr ähnlich Objekte behandelt
- diese müssen oder können nicht alle individuell beschrieben werden



# Klasse

## Motivation

- ➔ Idee: „Ähnliche“ oder „gleichartige“ Objekte werden durch eine Klasse nur einmal beschrieben
- ➔ Alle Objekte einer Klasse werden nach dieser Beschreibung aufgebaut



## Klasse

- **Individuelle Beschreibung aller Objekte ist in der Regel nicht möglich bzw. sinnvoll**
- **Einmalige Beschreibung gleichartiger Objekte durch Klassen**
- **Beschreibung definiert, wie Objekte der Klasse aufgebaut sind und welches Verhalten sie haben**

## Klasse

- **Aufbau (Struktur) einer Klasse wird durch Attribute beschrieben**
- **Das Verhalten der Objekte wird durch Funktionen (in der OOP = Methoden) festgelegt, die auf den Attributen (und ggf. Parametern) arbeiten**
- **In einer Klasse bilden Attribute und Methoden eine Einheit.**
- **Analogie Typ in funktionaler Programmierung**

# Klasse

## Klasse

Attribut $a_1$	Methode $m_1$
Attribut $a_2$	Methode $m_2$
Attribut $a_3$	Methode $m_3$
...	...
Attribut $a_m$	Methode $m_n$

## Datentyp

Komponente  $k_1$   
Komponente  $k_2$   
Komponente  $k_3$   
...  
Komponente  $k_l$

Funktion  $f_1$   
Funktion  $f_2$   
Funktion  $f_3$   
...  
Funktion  $f_k$

## Klasse

- **Nach der „reinen Lehre“ der Objektorientierung sind die Attribute und die Implementierungen der Methoden gekapselt, d.h. außerhalb der Klasse nicht bekannt.**
  - **Das Verhalten der Klasse wird durch die (axiomatische) Spezifikation der Methoden festgelegt.**
- Klasse kann als Implementierung eines abstrakten Datentyps betrachtet werden.**



## Klasse

- **Attribute haben den Charakter von objektlokalen Variablen, d.h.**
  - **Attribute haben einen Standardtyp (z.B. int, char, ...) oder gehören zu einer Klasse**
  - **Pointer können auch als Attribute verwendet werden**

## Klasse

- **Attribute haben den Charakter von objektlokalen Variablen, d.h.**
- **Ein Pointerattribut kann auch auf ein Objekt der eigenen Klasse zeigen (rekursive Klassen, analog zu rekursiven Datentypen in C)**

## Klasse

- Aus Klassen werden Objekte durch Instantiierung erzeugt, wobei Objekte meist einen Namen haben oder durch Referenzierung via Pointer eindeutig identifizierbar sind
- Analogie Variable in funktionaler Programmierung

## Klasse

- **Attribute haben den Charakter von objektlokalen Variablen, d.h.**
  - **jedes Objekt (als Instanz) einer Klasse hat seine eigenen Attributinstanzen\*), damit hat jedes Objekt individuelle Attributwerte**

\*) C++, erlaubt Klassenattribute, die nur einmal instanziiert werden unabhängig von der Zahl der instanziierten Objekte. Diese Klassenattribute existieren zur Laufzeit nur einmal und sind allen Objekten gemeinsam.

## Klasse

- **Methoden sind klassenspezifisch, d.h. für alle Objekte der Klasse gleich**
- **Methoden haben den Charakter von Funktionen bzw. Prozeduren, d.h.**
  - **Methoden können Parameter haben**
  - **Methoden können ein Ergebnis zurückliefern**

## Klasse

- **Methoden werden immer(!) an einem Objekt aufgerufen (d.h. das gerufene Objekt ist immer eine Art „impliziter“ Parameter der Methode)**
- **Das gerufene Objekt hat i.a. eine „Sonderstellung“ im Vergleich zu den Methodenparametern:  
Methode hat Zugriff auf gekapselte Daten und kann daher den Zustand des Objekts verändern (durch Veränderung der Attributwerte)**

## Klasse

- **Ein Methodenaufruf wird auch als Botschaft oder Auftrag an ein Objekt interpretiert, eine bestimmten Dienst (Methode) auszuführen.**
  - ➔ **„Erzeuge dich“ (Konstruktor)**
  - ➔ **„Beende dich“ (Destruktor)**
  - ➔ **„Ändere dich“**
  - ➔ **„Führe etwas aus“ (was nur du kannst)**
  - ➔ **„Gib mir eine Information über dich“**
  - ➔ **...**

# Klasse

## Beispiele:

### 1. Programm zur Verwaltung von Patientendaten in Arztpraxen

- Programm soll relevante Daten der Patienten erfassen und verwalten



(beliebige) Person bzw.  
Patient (Klasse) in der realen Welt

Abstraktion



Klasse Patient in der SW

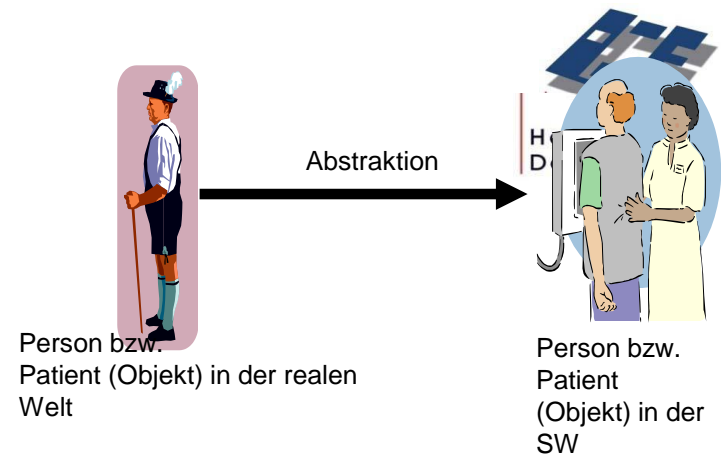


# Klasse

## Beispiele 1: Person

- Name
- Geburtsdatum
- Anschrift
- Haarefarbe
- Schuhgröße
- Schulabschluss
- Körpergröße
- Konfektionsgröße
- Krankenkasse
- Familienstand
- Hobbies
- Beruf
- ...

Abstraktion



## Patient in der SW

Attribute:

- Name
- Geburtsdatum
- Anschrift
- Krankenkasse

Methoden

- `set_name(char patientenname[30])`
- `lese_daten(&name, &gebdatum, ... )`

# Klasse

## Beispiele:

2. Programm zur Steuerung eines Verbrennungsmotors
  - Programm soll Zündung und Kraftstoffeinspritzung eines beliebigen Ottomotors steuern



zu entwickelnder Motor (Klasse) in der realen Welt

Abstraktion



zu entwickelnder Motor (Klasse) in der SW

# Klasse

## Beispiele 2:

### Motor

- Name
- Zylinderzahl
- Anzahl Zündkerzen pro Zylinder
- Anzahl Einspritzdüsen
- Hubraum
- Ventilanzahl pro Zylinder
- Zylinderwanddicke
- Ölsorte
- max. Leistung
- max. Drehmoment
- ...

Abstraktion



Motor (Objekt) in der realen Welt



Motor (Objekt) in der SW

### Motor in der SW

Attribute:

- Name
- Zylinderzahl
- Anzahl Zündkerzen pro Zylinder
- Anzahl Einspritzdüsen

Methoden:

- zünde(Zylinder)
- einspritzung(zyylinder, menge, dauer)

# Klasse

## Beispiele:

### 3. Bankkonto

- Programm soll Kontodaten verwalten



Abstraktion



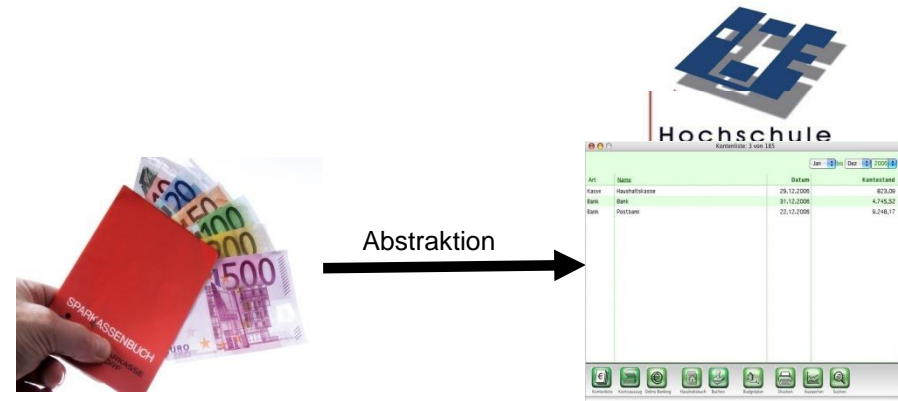
Art	Name	Datum	Kontostand
Kasse	Haushaltskasse	29.12.2006	823,09
Bank	Bank	31.12.2006	4.745,52
Bank	Postbank	22.12.2006	9.248,17

Konto (Objekt) in der realen Welt

Konto in der SW

# Klasse

## Beispiele 3:



Konto (Objekt) in der realen Welt

Konto in der SW

## Konto

- Nummer
- Inhaber (Name, Adresse)
- Saldo in €

Abstraktion

## Konto in der SW

- Nummer
- Inhaber (Name, Adresse)
- Saldo in €
- Eröffne\_Konto()
- Einzahlung(Betrag)
- Auszahlung(Betrag)

## Klasse

### Implementierung in C++ (Klassen und Objekte)

- **Grundsätzlicher Aufbau eines C++ Programms:**  
„wie ein C Programm“, d.h.
  - es gibt eine **main()** Funktion, die beim Programmstart ausgeführt wird
  - Präprozessorfunktionen wie bisher
  - alle C-Spachelemente stehen „in gewohnter Form“ zur Verfügung (sollten aber nur da, wo sinnvoll verwendet werden!)

# Klasse

## Implementierung in C++ (Klassen und Objekte)

- Grundsätzlicher Aufbau eines C++ Programms
- zusätzlich:
  - Klassen, Methoden, Objekte (klassische Elemente der Objektorientierten Programmierung)
  - Vererbung, Redefinition, Polymorphismus (klassische Elemente der Objektorientierten Programmierung)
  - Besonderheiten von C++

# Klasse

## Implementierung in C++ (Klassen und Objekte)

- Durch das Schlüsselwort „class“ wird eine Klasse definiert:

```
class c  
{  
    /* Rumpf der Klasse bestehend aus Attributen  
und */  
    /* Methoden  
*/  
};
```



## Klasse

### **Implementierung in C++ (Klassen und Objekte)**

- **Innerhalb des Klassenrumpfs können keine weiteren Klassen definiert werden**
- Klassen können nicht geschachtelt werden**
- Alle Klassen eines Programms sind auf dem gleichen Sichtbarkeitsniveau**

# Klasse

## Implementierung in C++ (Klassen und Objekte)

- **Attribute werden syntaktisch definiert wie Variable in C**

```
class c
{
    int attribut1;          /* Attribut eines Standardtyps */
    char attribut2[10];    /* Arrayattribut */
    float *fpointer;       /* Pointerattribut */
    class2 obj;            /* Attribut vom Typ einer anderen Klasse */
    class3 *cpointer       /* Attribut vom Typ Pointer auf Klasse */
};
```

- **Attribute werden in C++ auch als Membervariable bezeichnet**

# Klasse

## Implementierung in C++ (Klassen und Objekte)

- **Objekte werden angelegt wie Variable in C**

```
class c  
{  
    ...  
};
```

```
c objekt;
```

```
c *c_pointer = &objekt;
```

- **Jedes Objekt der Klasse c hat diese Attribute (mit ggf. unterschiedlichen Werten zur Laufzeit)**

# Klasse

## Implementierung in C++ (Klassen und Objekte)

- **Objekte können global oder lokal sein:**

```
class c { ... };
```

```
c globales_objekt;
```

```
class d  
{ int m(c objekt_als_parameter, ... )  
  { c lokales_objekt_einer_Methode;  
    ...  
  };  
};
```

```
int main ()  
{ c lokales_objekt_im_hauptprogramm;  
}
```

## Klasse

### Implementierung in C++ (Klassen und Objekte)

- **Lebensdauer und Sichtbarkeit von Objekten wie bei C Variablen**
  - **globale Objekte existieren während des gesamten Programmablaufs**
  - **methoden-lokale Objekte existieren während des Methodenaufrufs**
  - **Objekte auf Attributposition existieren, solange das umfassende Objekt existiert**
  - **lokale Objekte verschatten namensgleiche globale Objekte**

## Klasse

### Implementierung in C++ (Klassen und Objekte)

- **Attribute von Standardtypen werden in den Methoden der Klasse wie Variable verwendet, d.h. über ihren Namen angesprochen**

```
methode1 (... )  
{  
  ...  
  attribut1 = 5;  
  attribut2[5] = 'x';  
  fpointer = &x;  
  ...  
};
```

Zugriff auf Attribute des Objekts  
an dem die Methode aufgerufen  
wurde (nur Attributnamen)

# Klasse

## Implementierung in C++ (Klassen und Objekte)

- **Attribute von Standardtypen werden in den Methoden der Klasse wie Variable verwendet, d.h. über ihren Namen angesprochen**

```
methode1 (... , c o, ... )  
{  
  ...  
  o.attribut1 = 5;  
  o.attribut2[5] = 'x';  
  o.fpointer = &x;  
  ...  
};
```

Zugriff auf andere Attribute der Klasse c, zu der die Methode gehört. Objekts  
(Objektname.Attributnamen)

## Klasse

### Implementierung in C++ (Klassen und Objekte)

- **Methoden werden syntaktisch definiert wie**
  - **Funktionen (mit einem Ergebnistyp)**
  - **Prozeduren (ohne Ergebnistyp)**
- **Wie in C Funktionen können in C++ Methoden nicht geschachtelt werden**



## Klasse

### Implementierung in C++ (Klassen und Objekte)

- **Unterschied zu C-Funktionen / C-Prozeduren:**
  - **Methoden werden immer an einem Objekt aufgerufen**
- **Methoden werden in C++ auch als Memberfunktionen bezeichnet**

# Klasse

## Implementierung in C++ (Klassen und Objekte)

```
class c
{ int attribut1;          /* Attribut eines Standardtyps */

    public:
    void methode (int p) /* Methode */
    { attribut1 = p;
      };
};

c o;
c *c_pointer;
c_pointer = &o;

o.methode(5) /* Methodenaufruf am Objekt */
c_pointer->methode(5) /* Methodenaufruf am Pointer */
```

# Klasse

## Implementierung in C++ (Klassen und Objekte)

```
class patient
{
    char name[31]; // Name des Patienten max. Länge 30 + \0
    char adresse[51]; //Adresse des Patienten max. Länge 50 + \0
    int krankenkasse; // jede Kasse hat eine eindeutige Nummer

    public:

    void set_name(char patientenname[30])
    { strcpy(name,patientenname);
    };

    char* get_name()
    { return name;
    };

    // weitere Methoden werden hier nicht beschrieben
};
```

Schlüsselwort für Klasse **class**,  
gefolgt vom Klassennamen,

Attribute mit entsprechendem Typ

**public** kennzeichnet außerhalb der  
Klasse bekannte Bezeichner (s.  
Abschnitt Datenkapselung)

Methoden entsprechen der C-  
Syntax für Funktionen.

Achtung: Methoden arbeiten auf  
Parametern und Attributen!

Methoden werden in C++ als  
Memberfunktionen  
(Memberfunctions) bezeichnet!

# Klasse

## Implementierung in C++ (Klassen und Objekte)

```
main()
{
    patient p;
    patient *p_pointer;

    p_pointer = &p;

    p.set_name("Hans_Meier");

    p_pointer-> set_name("Hugo_Mueller");
};
```

Vereinbarung eines Objekts, Objekt wird instanziiert.

Vereinbarung eines Pointers auf ein Objekt, keine Objektinstanziierung

Aufruf einer Methode an einem Objekt

Aufruf einer Methode an einem Objekt, in diesem Fall ein dereferenzierter Pointer

# Klasse

## Klassen und Datenkapselung

- **Klassen kapseln Attribute und ggf. Methoden (solche, die nach außen nicht bekannt sein sollen / dürfen)**
- **Nach außen bekannt sind (nach der reinen Lehre) nur Methoden, mit denen die Anwender der Klasse arbeiten**

## Klasse

### **Implementierung der Datenkapselung in C++:**

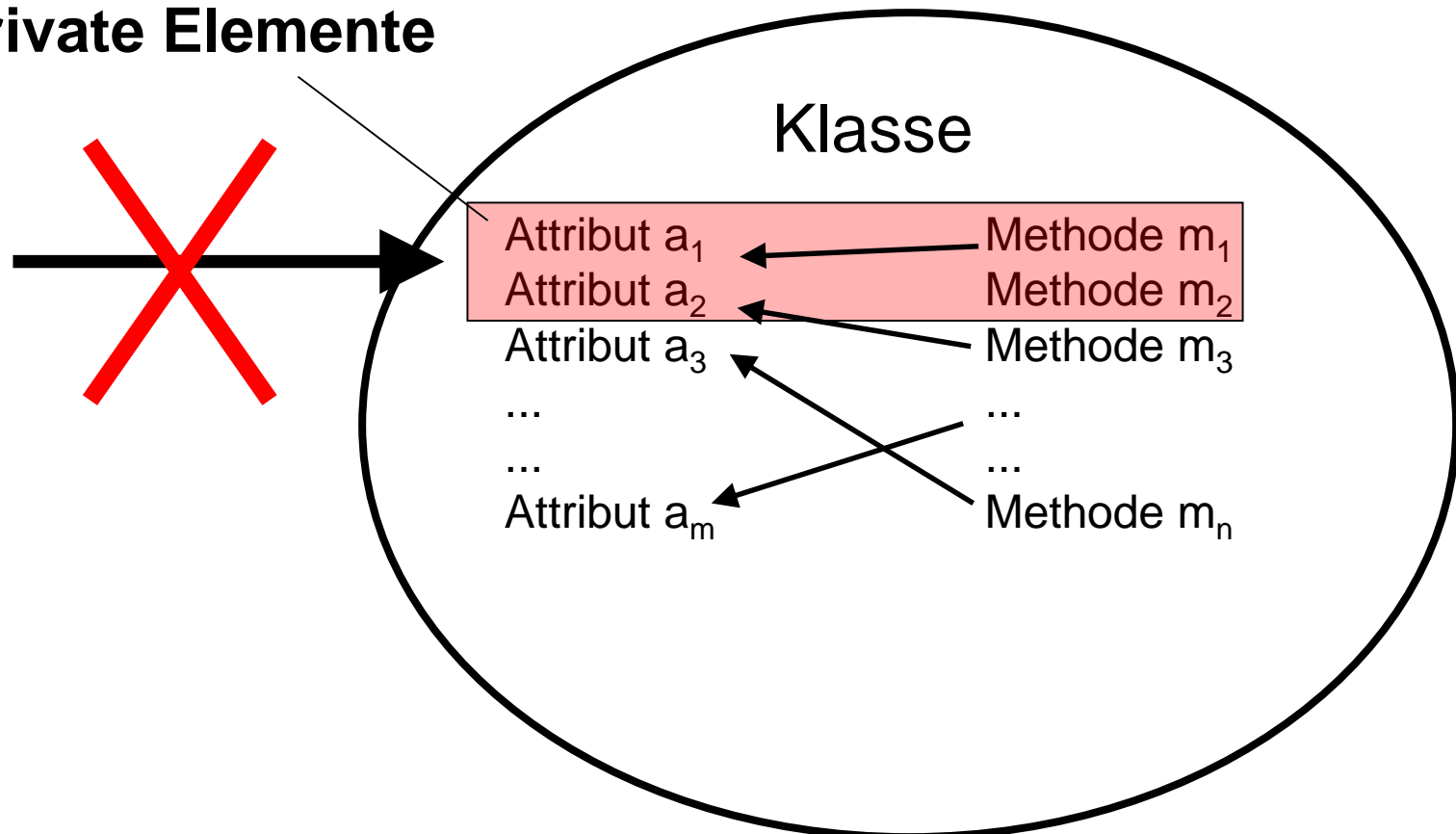
#### **C++ verfügt über eine dreistufige Datenkapselung**

- **private Elemente (Attribute, Methoden) sind nur innerhalb der Klasse bekannt, aber nicht nach außen und nicht in abgeleiteten Klassen (Unterklassen, → Abschnitt Vererbung). Private ist Voreinstellung**
- **nur die Methoden der Klasse haben Zugriff auf private Attribute und Methoden**

# Klasse

## Implementierung der Datenkapselung in C++: C++ verfügt über eine dreistufige Datenkapselung

- **private Elemente**



## Klasse

### **Implementierung der Datenkapselung in C++:**

#### **C++ verfügt über eine dreistufige Datenkapselung**

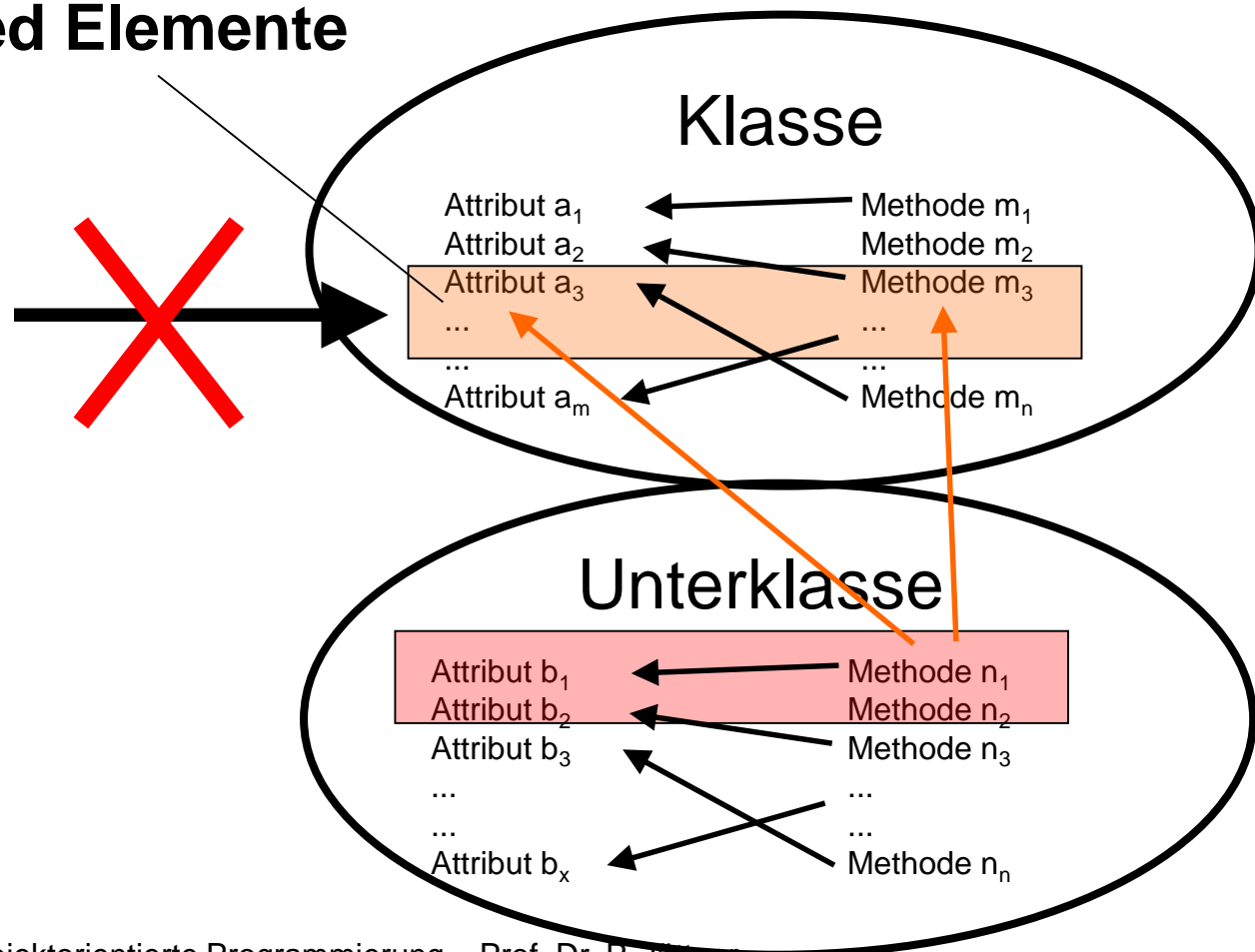
- **protected Elemente sind nur innerhalb der Klasse und in abgeleiteten Klassen (Unterklassen, → Abschnitt Vererbung) bekannt, aber nicht außerhalb der Klassenhierarchie**
- **nur die Methoden der Klasse und Unterklasse haben Zugriff**



# Klasse

## Implementierung der Datenkapselung in C++: C++ verfügt über eine dreistufige Datenkapselung

- protected Elemente



## Klasse

### **Implementierung der Datenkapselung in C++:**

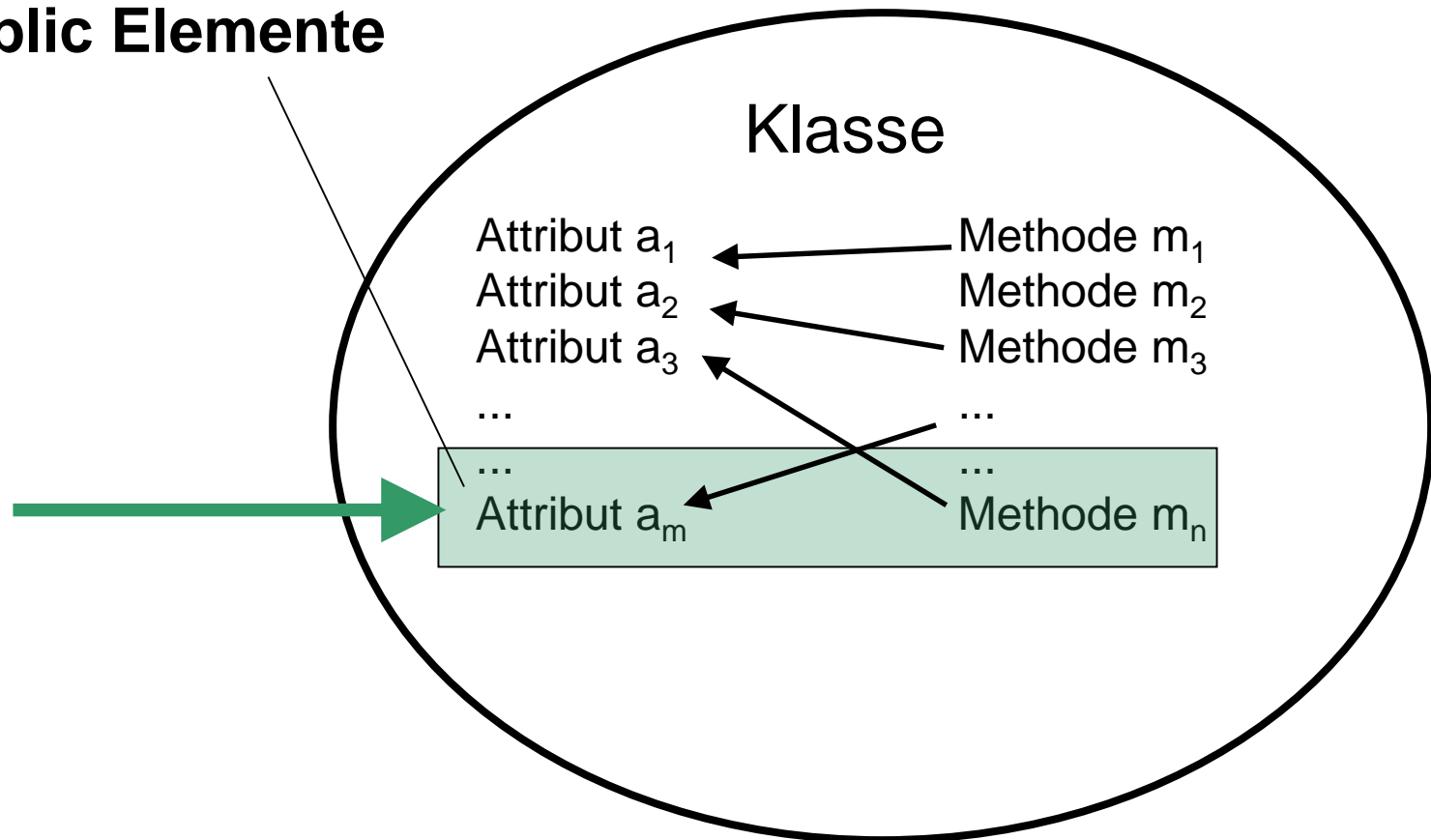
#### **C++ verfügt über eine dreistufige Datenkapselung**

- **public Elemente sind innerhalb der Klasse, außerhalb und in abgeleiteten Klassen (Unterklassen , → Abschnitt Vererbung) bekannt.**
- **„jeder“ hat Zugriff auf public Elemente**

# Klasse

## Implementierung der Datenkapselung in C++: C++ verfügt über eine dreistufige Datenkapselung

- **public Elemente**



# Klasse

## Implementierung der Datenkapselung in C++:

```
class kapselung
{
    private:
        int private_element;

    protected:
        int protected_element;

    public:
        int public_element;
};

main()
{
    kapselung objekt;

    objekt.private_element = 10;
    objekt.protected_element = 100;
    objekt.public_element = 1000;
}
```

# Übung

## Klassen & Datenkapselung

### Beschreibung:

Datenkapselung dient dazu, Implementierungsdetails einer Klasse vor dem Anwender der Klasse zu verbergen.

### Aufgabe:

1. Ergänzen Sie die Definition, aus der vorherigen Übung zu einer Klasse, d.h. definieren Sie gekapselte Attribute und (öffentliche) Methoden. Implementieren Sie diese in C++. Überlegen Sie sich alternative Implementierungen für ein oder mehrere Attribute.
2. Definieren Sie eine Klasse complex, die Komplexe Zahlen darstellt.  
Definieren Sie Methoden für die Addition, Subtraktion, Multiplikation, Division und Betrag
3. Definieren Sie eine Klasse Fahrzeug. Legen Sie relevante Attribute und Methoden fest. Beschreiben Sie Ihr Ergebnis in C++ Syntax.



### Zeit:

60 min.

## Klasse

**Zum Schluss dieses Abschnitts ...**

**Noch Fragen ??**

## Konstruktoren & Destruktoren

- **Konstruktoren und Destruktoren sind ...**
- **Finden Sie in Gruppenarbeit heraus,**
  - was in der Objektorientierten Programmierung Konstruktoren sind und für was Sie gut sind. (Gruppen 1)
  - was in der Objektorientierten Programmierung Destruktoren sind und für was Sie gut sind (Gruppen 2)
  - wie die Syntax von Konstruktoren und Destruktoren in C++ ist (Gruppe 3)
  - wann Konstruktoren und Destruktoren existieren (Gruppe 4)
  - wann Konstruktoren und Destruktoren aufgerufen werden (implizit, explizit) (Gruppe 5)
  - wie ein parameterloser und ein parametrierter Konstruktor für die Klasse Complex aussehen könnten, die Attribute geeignet initialisieren (Gruppe 6)

## Konstruktoren & Destruktoren

- **Konstruktoren und Destruktoren sind Methoden, die in jeder Klasse (automatisch) existieren**
- **„Standard-Aufgabe“ des Konstruktors:**
  - **Anlegen eines Objekts im Speicher (mindestens Reservierung des Speicherplatzes)**
  - **Aufrufen von Konstruktoren der Attribute (sofern diese Klassen sind)**
  - **ggf. Initialisieren von Attributen von C-Typen**



## Konstruktoren & Destruktoren

- **„Standard-Aufgabe“ des Destruktors:**
  - **Löschen des Objekts aus dem Speicher (mindestens Freigeben des Speicherplatzes)**
  - **Aufruf von Destruktoren der Attribute, sofern diese Klassen sind**
  - **ggf. „Aufräumarbeiten“**

## Konstruktoren & Destruktoren

- **Anlegen oder Löschen des Speicherplatzes für ein Objekt**
  - **auf dem Stack (automatische Speicherverwaltung durch Compiler)**
  - **auf dem Heap (dynamische Speicherverwaltung durch den Programmierer)**
- **analog zur Reservierung und Freigabe von Speicherplatz in der herkömmlichen Programmierung**

## Konstruktor & Destruktoren

- **jede Klasse hat mindestens einen Konstruktor und genau einen Destruktor (automatisch generiert durch den Compiler)**
- **der vom Compiler generierte Konstruktor ist parameterlos**

## Konstruktoren & Destruktoren

- **Weitere Aufgabe eines Konstruktors**
  - **Initialisierung von Attributen (in der Regel sind Attribute gekapselt und somit bei der Deklaration eines Objekts von außen nicht zugänglich)**
    - ➔ **Initialisierung wird aber nicht vom compilergenerierten Konstruktor durchgeführt!**
    - ➔ **Konstruktor muss explizit definiert werden**

## Konstruktoren & Destruktoren

- **Konstruktoren haben den gleichen Namen wie ihre Klasse.**
- **Der Rumpf des Konstruktors entspricht einem Methodenrumpf**

```
class c
{ ...
  c()
  { ... };
  ...
};
```

## Konstruktoren & Destruktoren

- **Konstruktoren können Parameter haben.  
(wie andere Methoden der Klasse)**

```
class c
{ ...
  c(int i_parameter, c2 o_parameter)
  { ... };
  ...
};
```

- **Konstruktoren haben keinen Rückgabetyp  
(auch nicht void).**

## Konstruktoren & Destruktoren

- **Eine Klasse kann mehrere Konstruktoren haben.**

```
class c
{ ...
  c()
  { ... };

  c(int i_parameter, c2 o_parameter)
  { ... };

  c(short s_parameter, c3 o_parameter2)
  { ... };
  ...
};
```

## Konstruktoren & Destruktoren

- **Konstruktoren können public, protected und private sein. Entsprechend ist ihre Sichtbarkeit.**

```
class c
{ private:
    c()
    { ... };
protected:
    c(int i_parameter, c2 o_parameter)
    { ... };
public:
    c(short s_parameter, c3 o_parameter2)
    { ... };
    ...
};
```



## Konstruktoren & Destruktoren

- **Unterschiedliche Konstruktoren werden beim Aufruf unterschieden durch ihre Signatur (Schnittstelle des Konstruktors)**
- **Aufgerufen werden Konstruktoren beim Anlegen eines Objekts**

# Konstruktor & Destruktoren

## Konstruktor Implementierung in C++

```
class konstruktor_test
{ int i;
public:
    konstruktor_test()
    { i = 0;
      printf("%s\n", "Konstruktor ohne Parameter wurde aufgerufen");
    };
    konstruktor_test (int j)
    { i=j;
      printf("%s%d\n", "Konstruktor mit Parameter j wurde aufgerufen aufgerufen mit j=",j);

    };
};
```

Definition eines parameterlosen Konstruktors, das Attribut i wird mit 0 initialisiert.

Definition eines zweiten Konstruktors mit einem int Parameter, das Attribut wird mit dem Wert des Parameters j initialisiert

# Konstruktor & Destruktoren

## Konstruktor Implementierung in C++

```
main ()  
{ konstruktor_test objekt1;  
  konstruktor_test objekt2(7);  
  ...  
};
```

Aufruf des parameterlosen Konstruktors (ohne“()“ !)

Aufruf des parametrisierten Konstruktors mit dem aktuellen Parameterwert 7

Das Programm erzeugt folgende Ausgabe:

*Konstruktor ohne Parameter wurde aufgerufen  
Konstruktor mit Parameter j wurde aufgerufen  
aufgerufen mit j=7*

## Konstruktoren & Destruktoren

- **Konstruktoren werden implizit aufgerufen, sobald ein Objekt erzeugt wird.**
- **Die Auswahl des richtigen Konstruktors geschieht dabei über die Parameter, sofern mehrere Konstruktoren vorhanden.**

```
konstruktor_test objekt1;  
/* Aufruf des parameterlosen Konstruktors */
```

```
konstruktor_test objekt2(7);  
/* Aufruf eines Konstruktors mit einem Parameter */
```

## Konstruktoren & Destruktoren

- **Gibt es einen Konstruktor mit einem oder mehreren Parametern, so wird dieser Konstruktor an den Stellen aufgerufen, an denen ein Objekt erwartet wird und ein Identifier vom passenden Parametertyp des Konstruktors steht**
- ➔ **Implizite Konstruktoraufrufe können somit auch eine Fehlerquelle sein!**

# Konstruktor & Destruktoren

## Konstruktor Implementierung in C++

```
class c1
{ public:
  c1(int i)
  { ... }
  ...
}

class c
{ public:
  void m(c1 p)
  { ... }
};
```

- Klasse c1 besitzt u.a. einen öffentlichen Konstruktor mit einem Parameter vom Typ int
- Klasse c besitzt u.a. eine öffentliche Methode mit einem Parameter vom Typ c1

# Konstrukturen & Destruktoren

## Konstruktor Implementierung in C++

```
main()
```

```
{ c o;
```

```
  o.m (5);
```

```
};
```

- Objekt o der Klasse c wird angelegt
- Methode m wird aufgerufen mit einem Integerparameter  
(„ungewöhnlich“, da m einen Parameter der Klasse c1 erwartet)
- Der für ein Objekt der Klasse c1 passende Konstruktor mit int Parameter wird **implizit** (!) aufgerufen. Ein Objekt der Klasse c1 wird erzeugt und als Parameter an m übergeben. m wird ausgeführt  
**Das ist kein Syntaxfehler!**

## Konstruktoren & Destruktoren

- **Memberinitialisierung**
  - **Klassen können Attribute vom Typ anderer Klassen enthalten**
  - **Informationen dieser Klassen, die als Attribut verwendet werden, sind i.d.R. gekapselt, d.h. in der verwendenden Klasse nicht zugreifbar**
  - **Der Konstruktor einer Klasse ruft implizit Konstruktoren der Attributklassen auf**



## Konstruktor & Destruktoren

# Memberinitialisierung in C++

```
class Person
{ char name [100];
  char adresse[100];
public:
  Person(char n[100], char a[100]) // Konstruktor der Klasse Person
  {
    strcpy(name,n);

    strcpy(adresse, a); // Kopieren der Parameter in die Attribute

  };
};
```

# Konstruktor & Destruktoren

## Memberinitialisierung in C++

```
class Mitarbeiter  
{ Person ma;  
  unsigned int personalnummer;  
public:  
  Mitarbeiter (char n[100], char a[100], unsigned int pn) : ma(n,a)  
  {  
    personalnummer = pn;  
  };  
};
```

Expliziter Aufruf des parametrisierten Konstruktors der Klasse Person für das Attribut ma, setzt die Attribute name und adresse des Mitarbeiters

## Konstruktoren & Destruktoren

- **Destruktoren werden verwendet um Objekte gezielt zu löschen bzw. notwendige „Aufräumarbeiten“ durchzuführen.**
- **Ein Destruktor hat keine Parameter und keinen Rückgabetyp. Daher gibt es maximal einen Destruktor pro Klasse**
- **Der Name des Destruktors in C++ setzt sich zusammen aus dem „~“ Zeichen und dem Klassennamen (z.B. ~Konto())**

## Konstruktor & Destruktoren

### Aufrufzeitpunkte von Konstruktoren und Destruktoren

- **Globale Objekte**
  - **werden beim Programmstart in der Reihenfolge ihrer Definition angelegt (unabhängig, ob sie innerhalb der Quelldatei oder global im gesamten Programm Gültigkeit besitzen.**
  - **werden beim Programmende freigegeben, die Freigabe erfolgt in umgekehrter Reihenfolge des Anlegens.**

## Konstruktoren & Destruktoren

### Aufrufzeitpunkte von Konstruktoren und Destruktoren

- **Statische Objekte innerhalb von Funktionen (Objekte die lokal in einer Funktion sind, nach Beendigung der Funktion erhalten bleiben)**
  - **werden beim ersten Aufruf der Funktion per Konstruktor angelegt.**
  - **Destruktoren werden beim Programmende in umgekehrter Reihenfolge der Konstruktoraufrufe aktiviert.**

## Konstruktor & Destruktoren

### Aufrufzeitpunkte von Konstruktoren und Destruktoren

- **Automatische Objekte (Objekt, das auf dem Programmstack angelegt wird, z.B. Parameter, lokales Objekt einer Funktion / Methode)**
- **Konstruktor automatischer Objekte werden aufgerufen, wenn ein Objekt angelegt wird.**

## Konstruktoren & Destruktoren

### Aufrufzeitpunkte von Konstruktoren und Destruktoren

- **Automatische Objekte (Objekt, das auf dem Programmstack angelegt wird, z.B. Parameter, lokales Objekt einer Funktion / Methode)**
- **Die Destruktoren werden beim Verlassen des Gültigkeitsbereichs aufgerufen, direkt bevor das Objekt auf dem Stapel ungültig wird.**

## Konstruktoren & Destruktoren

### Aufrufzeitpunkte von Konstruktoren und Destruktoren

- **Dynamische Objekte**
  - ➔ **beim Aufruf von new, nachdem der Speicherplatz reserviert ist**
  - ➔ **beim Aufruf von delete bevor der Speicherplatz zurückgegeben wird**



# Konstruktor & Destruktoren

## Konstruktoraufruf - Implementierung in C++

```
class konstruktor_test
{ int i;
  public:
    konstruktor_test()
    { i = 0;
      printf("%s\n", "Konstruktor wurde aufgerufen");
    };
    ~konstruktor_test()
    {
      printf("%s\n", "Destruktor wurde aufgerufen");
    };
};

void impliziter_konstruktor_und_destruktoraufruf()
{ konstruktor_test t;

main ()
{ impliziter_konstruktor_und_destruktoraufruf();
  ...
};
```

Prozedur hat ein lokales Objekt vom Typ „konstruktor\_test“

Bei der Ausführung wird das lokale Objekt instanziiert und zuerst der Konstruktor, dann bei Beendigung der Prozedur der Destruktor aufgerufen.

Das Programm erzeugt folgende Ausgabe:

*Konstruktor wurde aufgerufen wurde aufgerufen*

*Destruktor wurde aufgerufen*

## Datenkapselung

### Datenkapselung und Konstruktoren

- **Klassen können als gekapselte Attribute von anderen Klassen vorkommen.**
- **In der Klasse, die die Klassenattribute enthält, können die Attribute der Attributklassen wegen der Datenkapselung nicht initialisiert werden.**
- **Dieses Problem kann durch Memberinitialisierungslisten gelöst werden.**

# Datenkapselung

## Datenkapselung und Konstruktoren, Beispiel:

```
class date
{ short tag, monat, jahr;
...
public:
    date (short t, short m, short j)
    // Konstruktor
    { tag = t; monat = m, jahr = j; } ;
...
};
```

```
class person
{ char name[30];
  date geburtsdatum;
public:
    person(char name[30]; short t, short m,
    short j)
        : geburtsdatum(t,m,j)
    { name = strcpy (...);
    };
...
};
```

Aufruf des Konstruktors der Klasse date in der Memberinitialisierungsliste ermöglicht ein definiertes Initialisieren des Attributs geburtsdatum

## Konstruktor & Destruktoren

# Konstruktor, Destruktoren und Dynamische Speicherverwaltung

- **Mittels des Operators „new“ können zur Laufzeit dynamisch Objekte angelegt werden.**

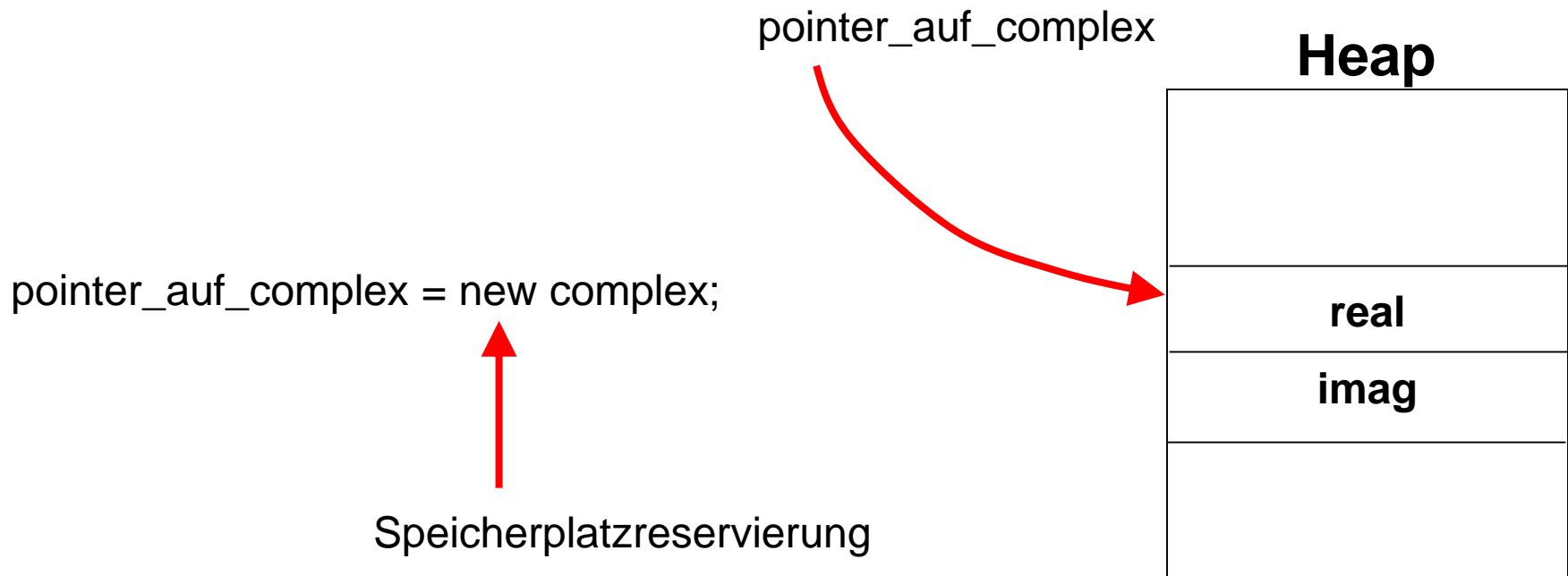
```
class complex
{ float real;
  float imag;
  ...
  complex()
  { real = 0.0; imag = 0.0; }
};
complex *pointer_auf_complex;

pointer_auf_complex = new complex;
```

## Konstruktor & Destruktoren

### Konstruktor, Destruktoren und Dynamische Speicherverwaltung

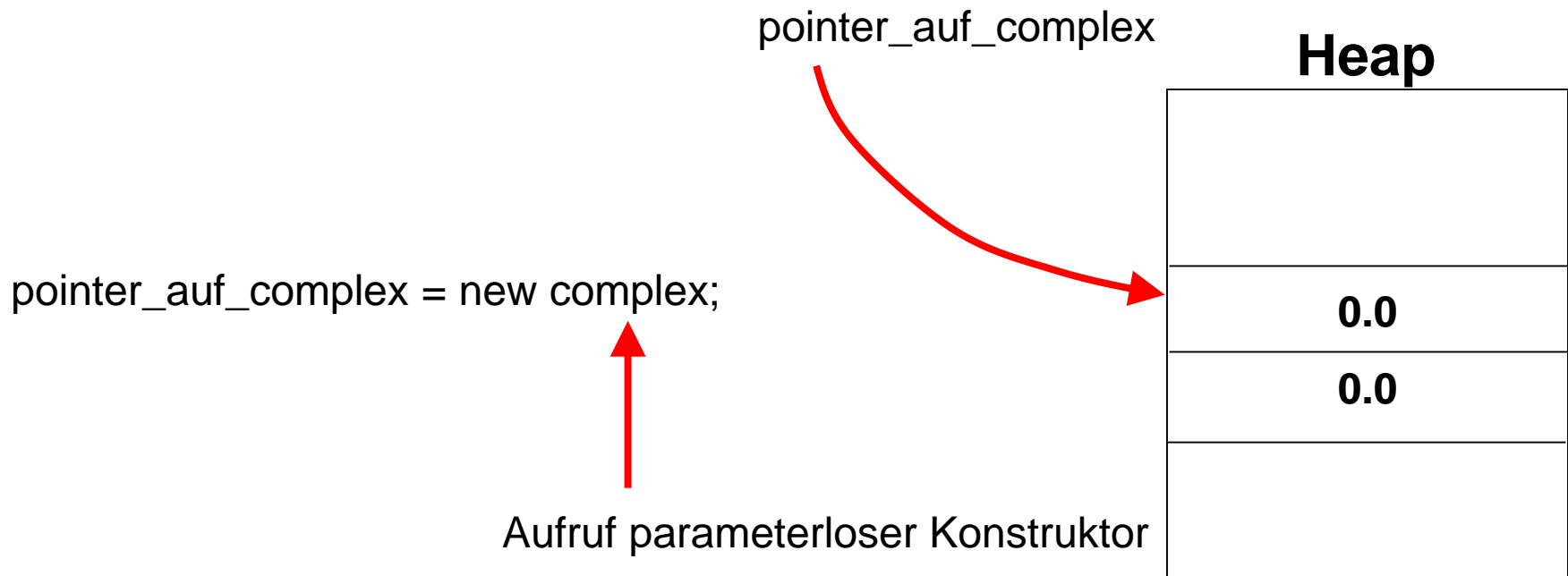
- **New reserviert zunächst Speicherplatz für ein neues Objekt auf dem Heap (ähnlich zu malloc() in C)**



## Konstruktor & Destruktoren

### Konstruktor, Destruktoren und Dynamische Speicherverwaltung

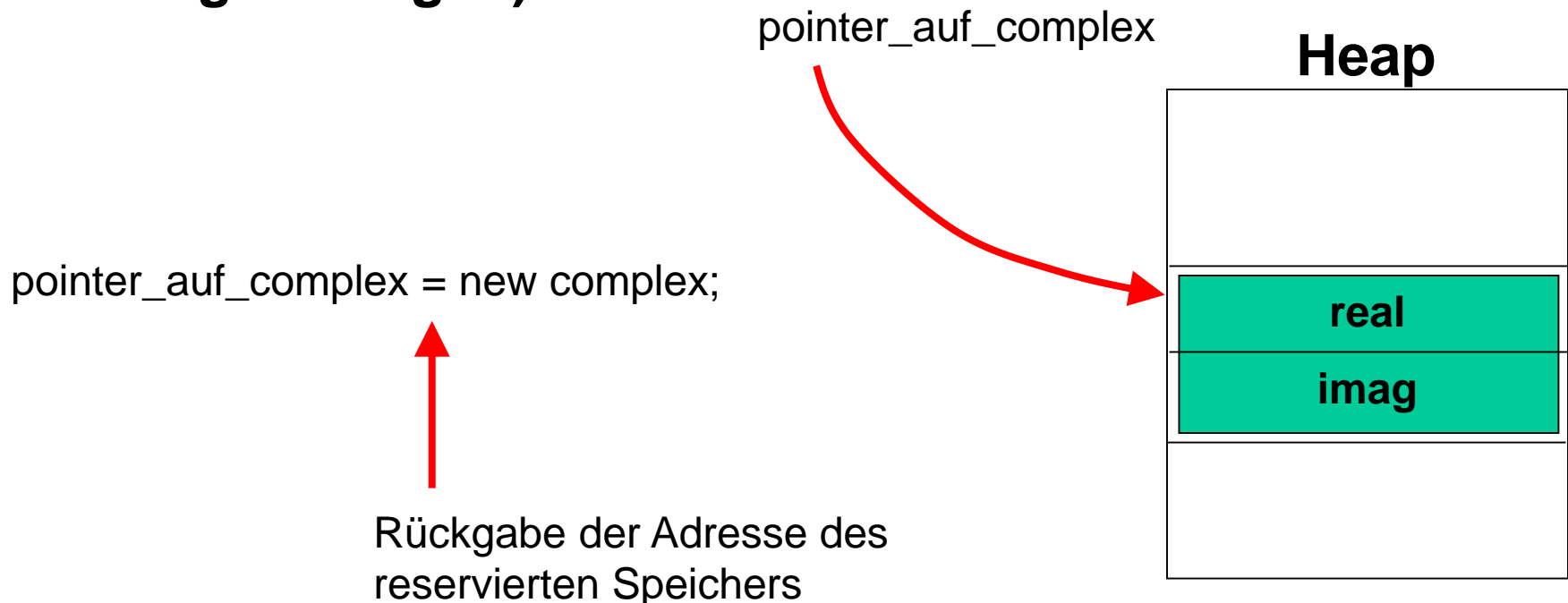
- **New ruft dann implizit den passenden Konstruktor (im Beispiel parameterlos) auf.**



## Konstruktor & Destruktoren

### Konstruktor, Destruktoren und Dynamische Speicherverwaltung

- Als Rückgabewert liefert `new` die Adresse des neuen Objekts (oder `NULL` falls Speicherplatzreservierung fehlgeschlagen)



## Konstruktoren & Destruktoren

### Konstruktoren, Destruktoren und Dynamische Speicherverwaltung

- **Syntax von new mit parameterlosem Konstruktor**

**Pointer\_auf\_Objekt = new Klassenname;**

- **Syntax von new mit parameteriertem Konstruktor**

**Pointer\_auf\_Objekt = new Klassenname  
(5,10,20)\*);**

\* ) Annahme der Konstruktor hat drei Parameter vom Typ int



## Konstruktor & Destruktoren

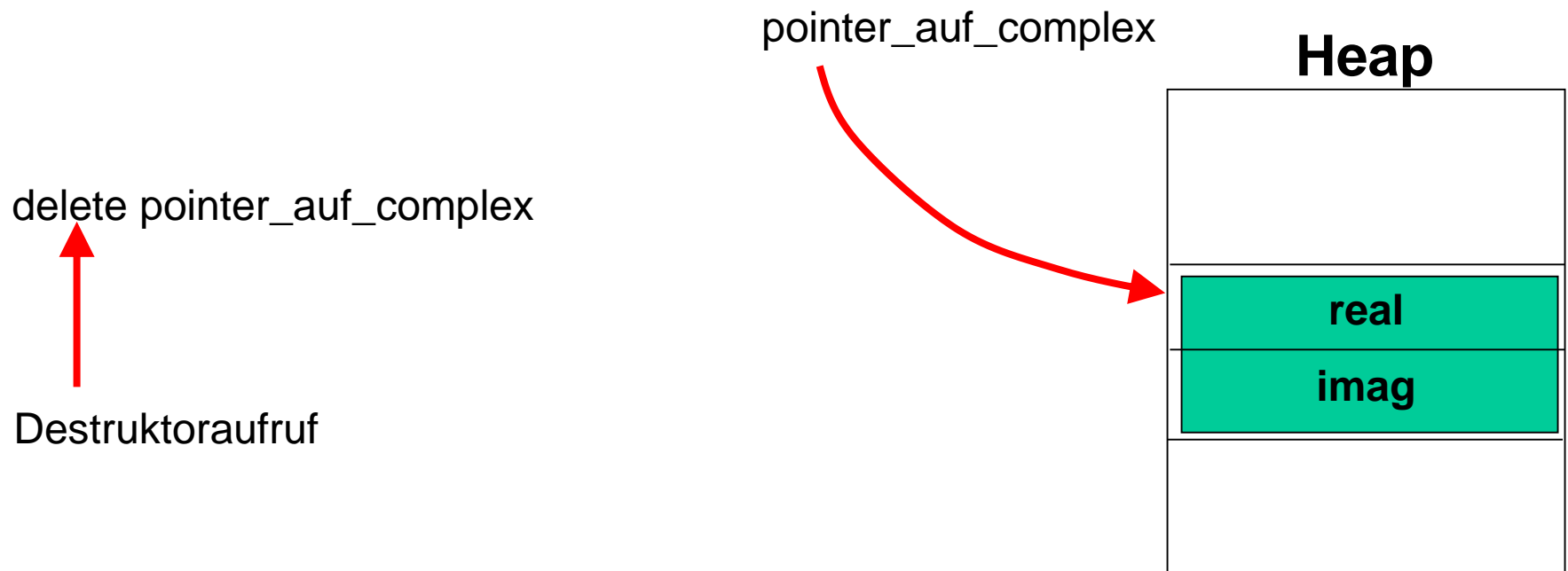
# Konstruktor, Destruktoren und Dynamische Speicherverwaltung

- **Syntax von delete**  
**delete Pointer\_auf\_Objekt;**

## Konstruktor & Destruktoren

# Konstruktor, Destruktoren und Dynamische Speicherverwaltung

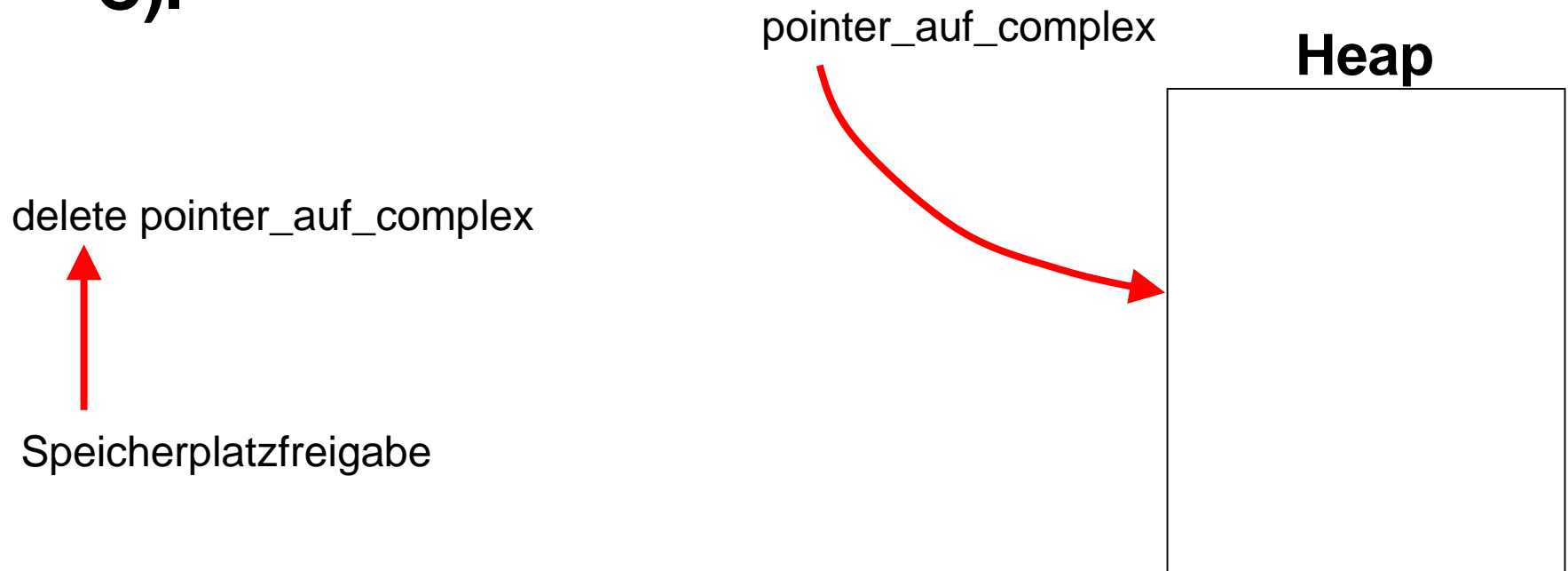
- **Delete ruft zunächst den Destruktor der Klasse auf, zu der das Objekt gehört**



## Konstruktor & Destruktoren

# Konstruktor, Destruktoren und Dynamische Speicherverwaltung

- **Delete** gibt nach dem Destruktoraufruf den Speicherplatz wieder frei (ähnlich zu `free()` in C).



## Konstruktor & Destruktoren

# Konstruktor, Destruktoren und Dynamische Speicherverwaltung

- **Achtung:**
  - **Der Pointer an dem delete aufgerufen wurde, behält seinen Wert**

```
pointer_auf_complex = new complex;  
printf(„%...“, pointer_auf_complex); /* Ausgabe 0x2345 */  
...  
delete pointer auf complex;  
printf(„%...“, pointer_auf_complex); /* Ausgabe 0x2345 */
```

## Konstruktor & Destruktoren

# Konstruktor, Destruktoren und Dynamische Speicherverwaltung

- **Achtung:**
  - **Delete darf nur auf eine Adresse angewendet werden, die per new erhalten wurde (delete NULL ist auch OK)**



```
complex c;  
pointer_auf_complex = &c;  
delete pointer_auf_complex;
```

## Konstruktor & Destruktoren

# Konstruktor, Destruktoren und Dynamische Speicherverwaltung

- **Achtung:**
  - **Delete darf niemals zweimal hintereinander am gleichen Objekt aufgerufen werden (ohne zwischenzeitliches new)**

```
complex c;  
pointer_auf_complex = new complex;  
delete pointer auf complex;  
delete pointer auf complex;
```



## Konstruktoren & Destruktoren

# Konstruktoren, Destruktoren und Dynamische Speicherverwaltung

- **Achtung:**
  - **Mittels new angelegte Objekte sind nur über einen Pointer zugänglich, sie besitzen keinen Namen**

# Konstruktor & Destruktoren

## Konstruktor-/Destruktoraufrufe - Implementierung in C++

```
class new_delete_test
{ int i;
  public:
    new_delete_test (int j)
    { i=j;
      printf("%s%d\n", "Konstruktor mit Parameter j wurde aufgerufen mit j=",j);
    };
    new_delete_test ()
    { i=0;
      printf("%s\n", "Konstruktor ohne Parameter wurde aufgerufen");
    };
    ~new_delete_test()
    {
      printf("%s\n", "Destruktor wurde aufgerufen");
    };
    void ausgabe()
    { printf("%s%d\n", "Objekt hat Attribut i mit Wert ",i);
    };
};
```



# Konstruktor & Destruktoren

## Konstruktoraufruf - Implementierung in C++

```
main ()  
{ new_delete_test* Pointer_auf_neues_Objekt1;  
  
  new_delete_test* Pointer_auf_neues_Objekt2;  
  
  Pointer_auf_neues_Objekt1 = new new_delete_test;  
  Pointer_auf_neues_Objekt1->ausgabe();  
  
    Pointer_auf_neues_Objekt2 = new new_delete_test(10);  
  Pointer_auf_neues_Objekt2->ausgabe();  
  
  delete Pointer_auf_neues_Objekt1;  
  
  delete Pointer_auf_neues_Objekt2;  
  
  delete Pointer_auf_neues_Objekt2;  
  
};
```

Pointer auf Testobjekte

Speicherplatzanforderung und Aufruf der  
Konstruktor (einmal ohne Parameter,  
einmal mit Parameter)

Aufrufe des Destruktors und Freigabe des  
Speicherplatzes

Illegaler 2. Aufruf des Destruktors am  
gleichen Objekt (Pointer zeigt noch immer  
auf den vorher angeforderten und schon  
freigegebenen Speicherplatz!  
(wird vom Compiler nicht abgelehnt)

## Konstruktor & Destruktoren

### Konstruktor und Destruktoren bei Arrays

- Bei Arrays von Objekten wird der Standardkonstruktor aufgerufen. Eine Initialisierung ist nur elementweise möglich.
- Beim Aufruf von delete ist „[ ]“ anzugeben, damit bei jedem Arrayelement der Destruktor aufgerufen wird.

# Konstruktoren & Destruktoren

## Beispiel

```
class string
{ char* p;
  unsigned short laenge;
  public:
  string ()
  { p = new char[1]; laenge = 0;
    *p = '\0';
    printf("%s\n", "Konstruktor ohne Parameter wurde aufgerufen");
  };
  ~string()
  { printf("%s\n", "Destruktor wurde aufgerufen"); } // Destruktor
};
```

```
main ()
{
  string* stringpointer = new string[20];
  delete[] stringpointer;
  system("PAUSE");
};
```

Aufruf des parameterlosen  
Konstruktors für jedes Arrayelement  
(insgesamt 20)

Aufruf des Destruktors für jedes  
Arrayelement (insgesamt 20)

## Konstruktoren & Destruktoren

- **Wiederholung**  
**„Konstruktoren und Destruktoren sind ...**

**Bearbeiten Sie in einer Gruppe eines der folgenden Themen und bereiten Sie eine entsprechende Tafelanschrift zu Ihrem Gruppenthema vor:**

- Gruppe 1: Was sind in der Objektorientierten Programmierung Konstruktoren , für was sind sie gut und warum werden sie benötigt?
- Gruppe 2: Was sind in der Objektorientierten Programmierung Destruktoren, für was sind sie gut und warum werden sie benötigt?
- Gruppe 3: Wie die Syntax von Konstruktoren und Destruktoren in C++ ?

## Konstruktoren & Destruktoren

- **Wiederholung**  
**„Konstruktoren und Destruktoren sind ...**

**Bearbeiten Sie in einer Gruppe eines der folgenden Themen und bereiten Sie eine entsprechende Tafelanschrift zu Ihrem Gruppenthema vor:**

- Gruppe 4: Wann existieren Konstruktoren und Destruktoren?
- Gruppe 5: Wann werden Konstruktoren und Destruktoren aufgerufen (implizit, explizit) ?
- Gruppe 6: Was ist eine Memberinitialisierung ?
- Gruppe 7: Was ist der new Operator? Wie hängt er mit Konstruktoren zusammen?

## Konstruktoren & Destruktoren

- **Wiederholung**  
**„Konstruktoren und Destruktoren sind ...**

**Bearbeiten Sie in einer Gruppe eines der folgenden Themen und bereiten Sie eine entsprechende Tafelanschrift zu Ihrem Gruppenthema vor:**

- Gruppe 8: Was ist der delete Operator? Wie hängt er mit Destruktoren zusammen?
- Gruppe 9: Was ist im Zusammenhang von Vektoren (Arrays) und Konstruktoren bzw. Destruktoren zu beachten?
- Gruppe 10: Was versteht man unter Memberinitialisierung?

## Konstruktoren & Destruktoren

- **Wiederholung**  
**„Konstruktoren und Destruktoren sind ...**

**Bearbeiten Sie in einer Gruppe eines der folgenden Themen und bereiten Sie eine entsprechende Tafelanschrift zu Ihrem Gruppenthema vor:**

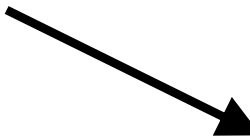
- Gruppe 11: Was ist ein Copy-Konstruktor (Kopierkonstruktor), wie sieht er aus und warum wird er benötigt?
- Gruppe 12: Warum muss i.d.R. der Zuweisungsoperator überladen werden, wenn es einen eigenen Copy-Konstruktor gibt? Wie sieht der überladene Zuweisungsoperator aus?

## Konstruktoren & Destruktoren

### Ein besonderer Konstruktor: Der Kopierkonstruktor (Copy Constructor)

- was passiert in folgenden Codebeispiel?

```
class string
{ char* p;
  short länge;
public:
  string(const char* s) // Konstruktor
  { p = new char[strlen(s)+1];
    strcpy(p,s);
    länge = strlen(s);
  };
  ~string() // Destruktor
  { ... };
};
```



```
...
string s1("Hallo Welt");
...
string s2 = s1;
...
```



## Konstruktoren & Destruktoren

### Ein besonderer Konstruktor: Der Kopierkonstruktor (Copy Constructor)

- was passiert in folgenden Codebeispiel?

...

string s1("Hallo Welt");

...

string s2 = s1;

...

Deklaration eines Objekts s1 und Aufruf des Konstruktors mit Parameter char \*

Deklaration eines Objekts s2 und Aufruf eines Kopierkonstruktors, **keine Zuweisung!**

Kopierkonstruktor (hier generiert vom Compiler) kopiert elementweise!

## Konstruktor & Destruktoren

### Ein besonderer Konstruktor: Der Kopierkonstruktor (Copy Constructor)

- was passiert in folgenden Codebeispiel?

...

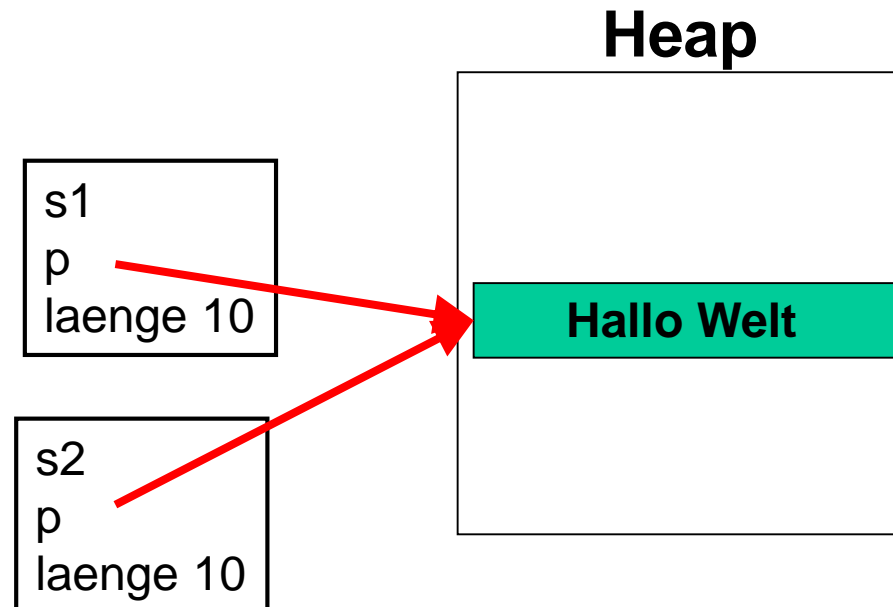
```
string s1("Hallo Welt");
```

...

```
string s2 = s1;
```

```
/* Das Attribut p von s1 zeigt auf die */  
/* gleiche Speicherstelle wie p von s2 */  
/* Immer wenn sich der Text von s1 */  
/* ändert, ändert sich auch s2 ! */
```

...



## Konstruktoren & Destruktoren

### Kopierkonstruktor

- Falls dieser „Effekt“ (Pointer zeigen auf gleichen Heap-Bereich) nicht gewünscht ist  
→ Definition eines eigenen Kopierkonstruktors:

```
klassenname(const klassenname& o)  
{ ...  
    /* Durchführung der gewünschten */  
    /* Kopieroperationen */  
};
```

## Konstruktoren & Destruktoren

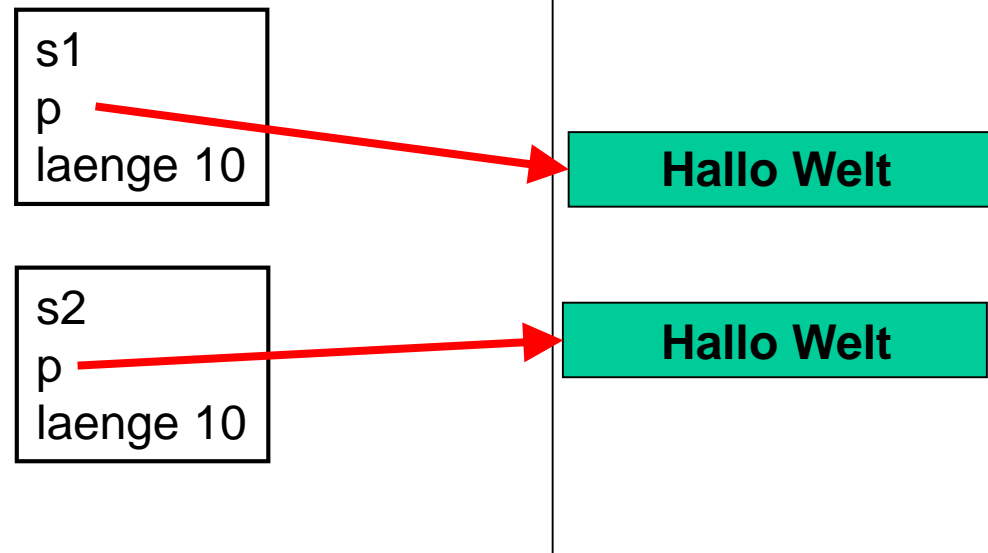
### Kopierkonstruktor (Copy Constructor)

- Im Beispiel string:

```
string(const string& s)
{ p = new char[s.laenge+1];
  strcpy(p,s.p);
  laenge = s.laenge;
};
```

...

```
string s1(„Hallo Welt“);
string s2 = s1;
```



## Konstruktor & Destruktoren

### Kopierkonstruktor

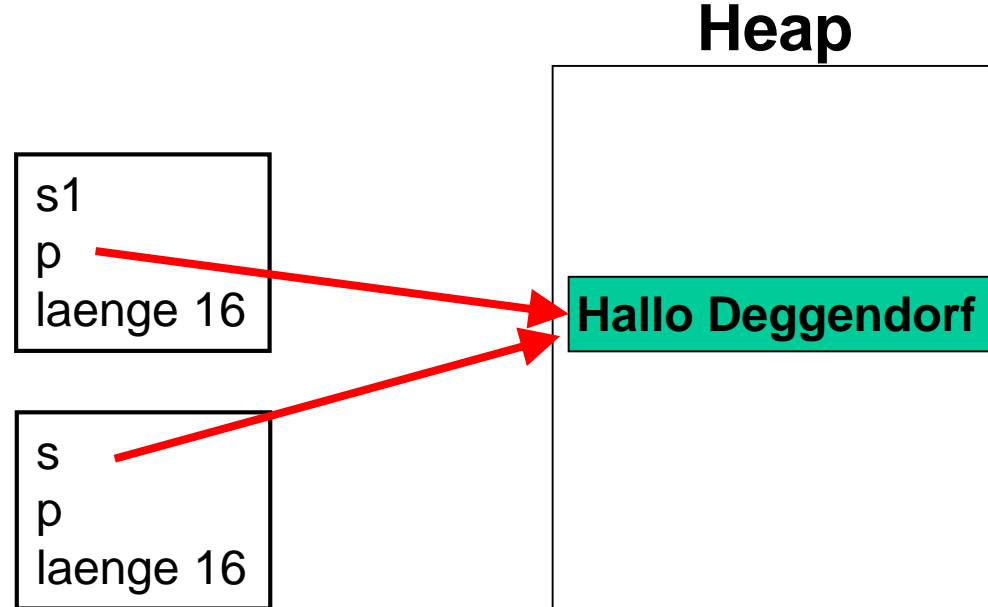
### Parameterübergabe als weitere Anwendung des Kopierkonstruktors

- **Objekt (keine Referenz) als Parameter einer Funktion / Methode → Kopierkonstruktor wird aufgerufen**

*/\* Ohne selbst definierten Kopierkonstruktor \*/*

```
class printer
{ ...
  druckestring(string s)
  { ... };
  ...
};

...
printer p;
string s1(„Hallo Deggendorf“);
p.druckestring(s1);
```



## Konstruktor & Destruktoren

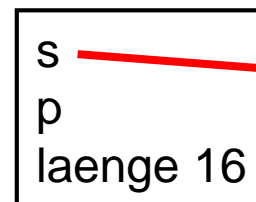
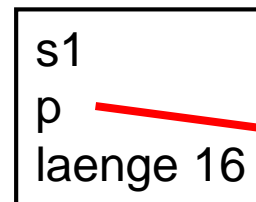
### Kopierkonstruktor

### Parameterübergabe als weitere Anwendung des Kopierkonstruktors

- **Objekt (keine Referenz) als Parameter einer Funktion / Methode → Kopierkonstruktor wird aufgerufen**

*/\* Mit selbst definierten Kopierkonstruktor \*/*

```
class printer  
{ ...  
  druckestring(string s)  
  { ... };  
  ...  
};  
...  
printer p;  
string s1(„Hallo Deggendorf“);  
p.druckestring(s1);
```



**Heap**



Hallo Deggendorf

Hallo Deggendorf

## Konstruktoren & Destruktoren

### Kopierkonstruktor

### Achtung:

**Nach Ablauf der Methode `druckestring(s1)` wird der Destruktor für die Kopie des Parameterobjekts aufgerufen!**

**Schon aus diesem Grund sollte ein Kopierkonstruktor definiert werden, um zu verhindern, dass Speicher des Parameterobjekts irrtümlich freigegeben wird.**

## Motivation

**Zum Schluss dieses Abschnitts ...**

**Noch Fragen ??**



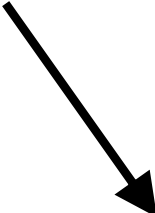
## Konstruktoren & Destruktoren

### Noch ein (verwandtes) Problem:

- was passiert in folgenden Codebeispiel?

```
class string
{ char* p;
  short länge;
public:
  string(const char* s) // Konstruktor
  { p = new char[strlen(s)+1];
    strcpy(p,s);
    länge = strlen(s);
  };
  ~string() // Destruktor
  { ... } ;
};
```

...



```
string s1("Hallo Welt");
string s2;
...
s2 = s1;
...
```

## Konstruktor & Destruktoren

### Noch ein (verwandtes) Problem:

- was passiert in folgenden Codebeispiel?

...

string s1("Hallo Welt");

string s2( "");

s2= s1;

...

Deklaration eines Objekts s1 und Aufruf des Konstruktors mit Parameter char \*

Deklaration eines Objekts s2 und Aufruf des Konstruktors mit Parameter char \*

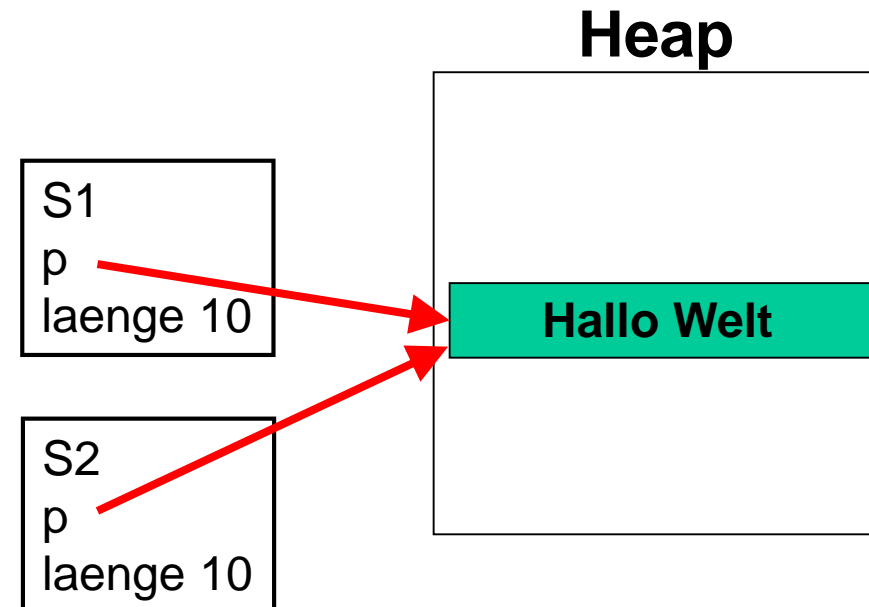
Zuweisung eines Objekts an ein anderes (komponentenweise)

## Konstruktor & Destruktoren

### Ein besonderer Konstruktor: Der Kopierkonstruktor (Copy Constructor)

- was passiert in folgenden Codebeispiel?

```
...  
string s1("Hallo Welt");  
string s2("");  
  
s2= s1;  
  
/* Das Attribut p von s1 zeigt auf die */  
/* gleiche Speicherstelle wie p von s2 */  
/* Immer wenn sich der Text von s1 */  
/* ändert, ändert sich auch s2 ! */  
...
```



## Konstruktor & Destruktoren

### Zuweisungsoperator

- Falls dieser „Effekt“ (Pointer zeigen auf gleichen Heap-Bereich) nicht gewünscht ist

➔ Definition eines eigenen  
Zuweisungsoperators

## Konstruktoren & Destruktoren

### Zuweisungsoperator

```
klassenname& operator=(const klassenname&  
o)  
{ ...  
    /* Durchführung der gewünschten */  
    /* Kopieroperationen                */  
    return *this; /* Zurückgeben eines  
Ergebnisses */  
};
```

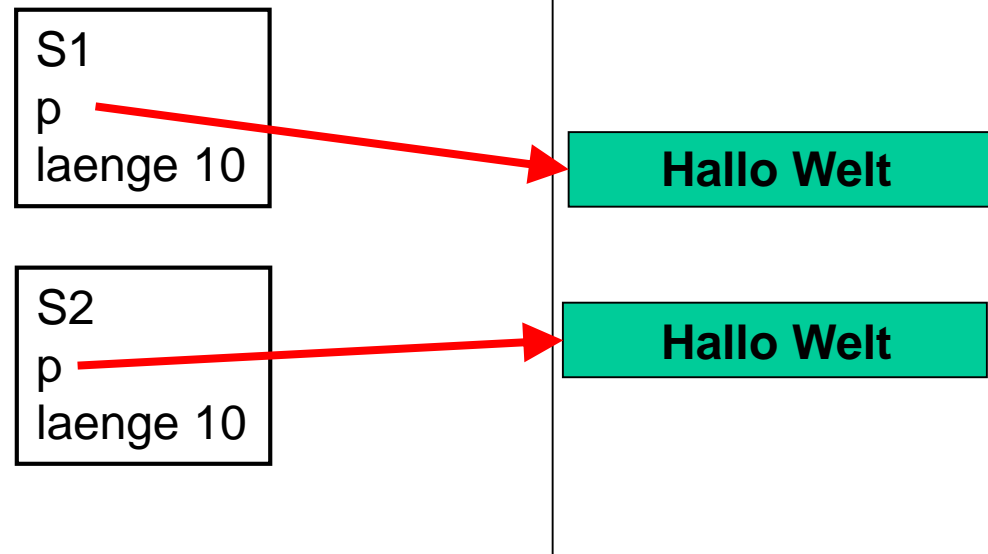
## Konstruktor & Destruktoren

### Zuweisungsoperator

- Im Beispiel string:

```
string& operator=(const string& s)
{ printf("Aufruf des Zuweisungsoperators\n");
  p = new char[s.laenge+1];
  strcpy(p,s.p);
  laenge = s.laenge;
  return *this;
};
```

```
...
string s1(„Hallo Welt“);
string s2(““);
...
s2 = s1;
```



## Klassen und dynamische Speicherverwaltung

### Zusammenfassung

**Eine Klasse K, die dynamisch (per new) Speicher reserviert, sollte über folgende Elemente verfügen:**

- einen selbst definierten Destruktor  $\sim K$ , der den angeforderten Speicherplatz, wo notwendig, wieder freigibt
- einen selbst definierten Zuweisungsoperator  $K\& \text{ operator}=(\text{const } K\&)$ , der Zuweisungsoperationen  $o1 = o2$  korrekt durchführt

## Klassen und dynamische Speicherverwaltung

### Zusammenfassung

**Eine Klasse K, die dynamisch (per new) Speicher reserviert, sollte über folgende Elemente verfügen:**

- einen selbst definierten Kopierkonstruktor  $K(const K\&)$ , der Initialisierungen  $K\ o1 = o2$  korrekt durchführt



## Motivation

**Zum Schluss dieses Abschnitts ...**

**Noch Fragen ??**

## This-Pointer

- **Eine Methode wird immer an einem Objekt aufgerufen (Objektname.methodenname(aktuelle Parameter))**
- **Eine Methode  $m_1$  kann die Methode  $m_2$  einer anderen Klasse aufrufen. Dabei kann es vorkommen, dass das Objekt, an dem  $m_1$  aufgerufen wurde, als Parameter an die Methode  $m_2$  übergeben werden muss.**

## This-Pointer

### Lösung: this-Pointer

- **Der this-Pointer zeigt innerhalb eines Methodenrumpfs immer auf das Objekt an dem eine Methode aufgerufen wurde.**
- **Er wird automatisch erzeugt**
- **Ist nur innerhalb des Methodenrumpfs bekannt.**

# This-Pointer

## this-Pointer

```
class c1
{ ...
  m1(c2 *p) { ... };
};
class c2
{ ...
  m2()
  { ...
    o1.m1(this);
    ...
  };
  ...
}
...
o2.m2();
...
}
```

Aufruf von m2 an o2 führt  
zu Aufruf von m1  
mit this-Pointer, der auf o2  
zeigt

# Konstruktor & Destruktoren

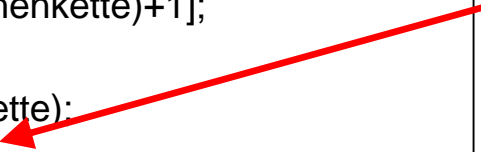
## Beispiel

```
class printer  
{ public:  
    void stringausgabe(stringklasse s)  
    { cout << "String lautet: " << s.p << endl; };  
};
```

```
class stringklasse  
{ friend class printer;  
    public: ... // Attribute, Konstruktor und Destruktor wie bisher
```

```
    void setzestring(char* zeichenkette)  
    { printer pr;  
      delete p;  
      p = new char [strlen(zeichenkette)+1];  
      strcpy(p, zeichenkette);  
      laenge = strlen(zeichenkette);  
      pr.stringausgabe(this);  
    };  
};
```

Übergabe des Objekts an dem  
setzestring aufgerufen wurde an die  
Methode stringausgabe via this-  
Pointer



## This-Pointer

### this-Pointer

- **Der this-Pointer kann auch verwendet werden um**
  - **auf Attribute des Aufrufobjekts zuzugreifen**  
**(this->attribut;)**
  - **Methoden der eigenen Klasse aufzurufen**  
**(this->methodenname(...);)**
- **Der this-Pointer sollte nicht per delete gelöscht werden!**

## This-Pointer

### this-Pointer

- **Dem this-Pointer sollte nicht die Adresse eines anderen Objekts zugewiesen werden**
- **Am this-Pointer sollte kein Speicherplatz angefordert werden.**

## Motivation

**Zum Schluss dieses Abschnitts ...**

**Noch Fragen ??**