

# Objektorientierte Programmierung

## **Grundbegriffe Teil 2**

Prof. Dr. Peter Jüttner

## Grundbegriffe – Vererbung

### Motivation

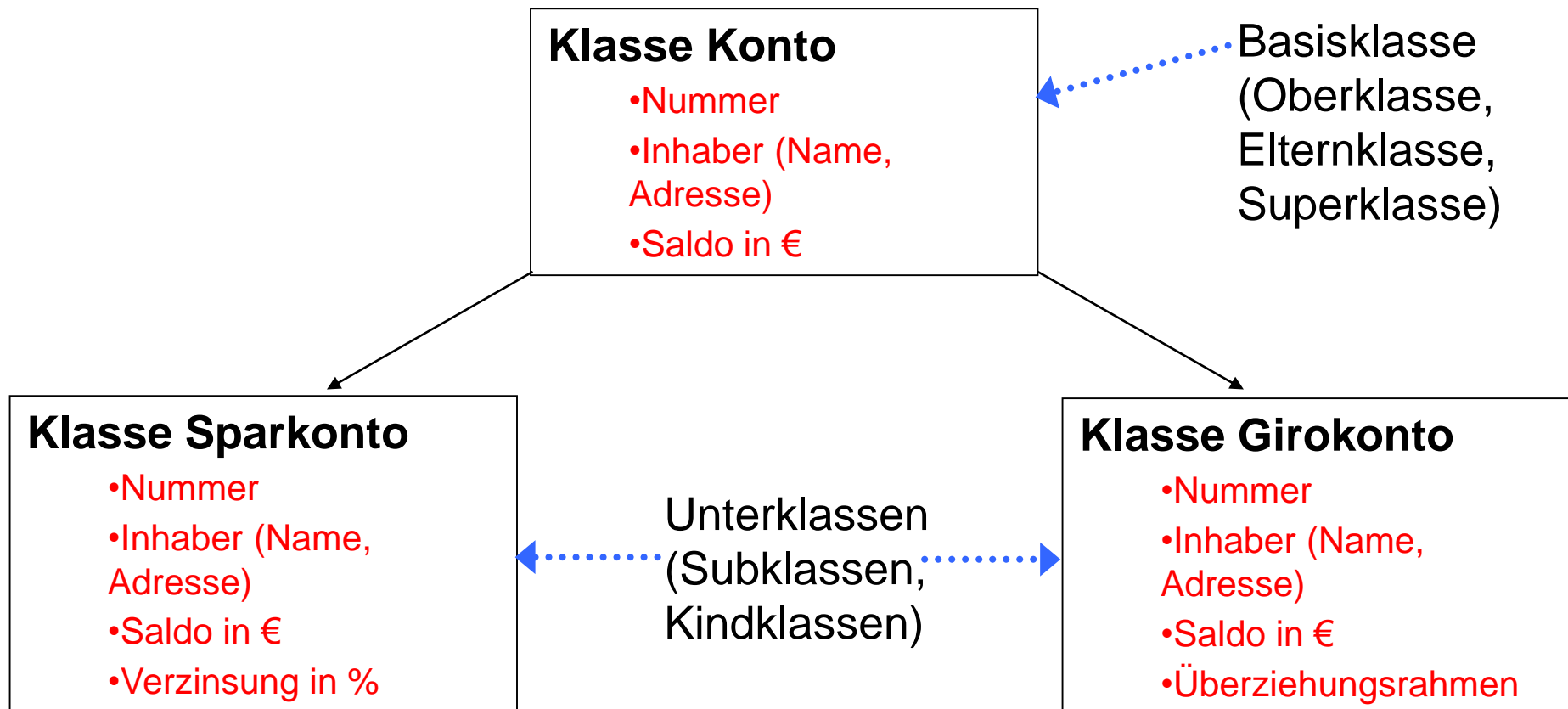
**Eine bereits existierende Klasse soll erweitert werden, ohne dass die Klasse selbst geändert werden muss.**

- **Bei der Erweiterung sollen alle bisherigen Eigenschaften der existierenden (Basis-) Klasse erhalten bleiben und neue in der (Unter-) Klasse definiert werden.**
- **Je nach Anwendung können dabei auch mehrere neue Klassen entstehen.**

# Grundbegriffe – Vererbung

## Beispiele

- Die Klasse „Konto“ soll erweitert werden, so dass zwei neue Klassen Sparkonto und Girokonto entstehen



## Grundbegriffe – Vererbung

### Motivation

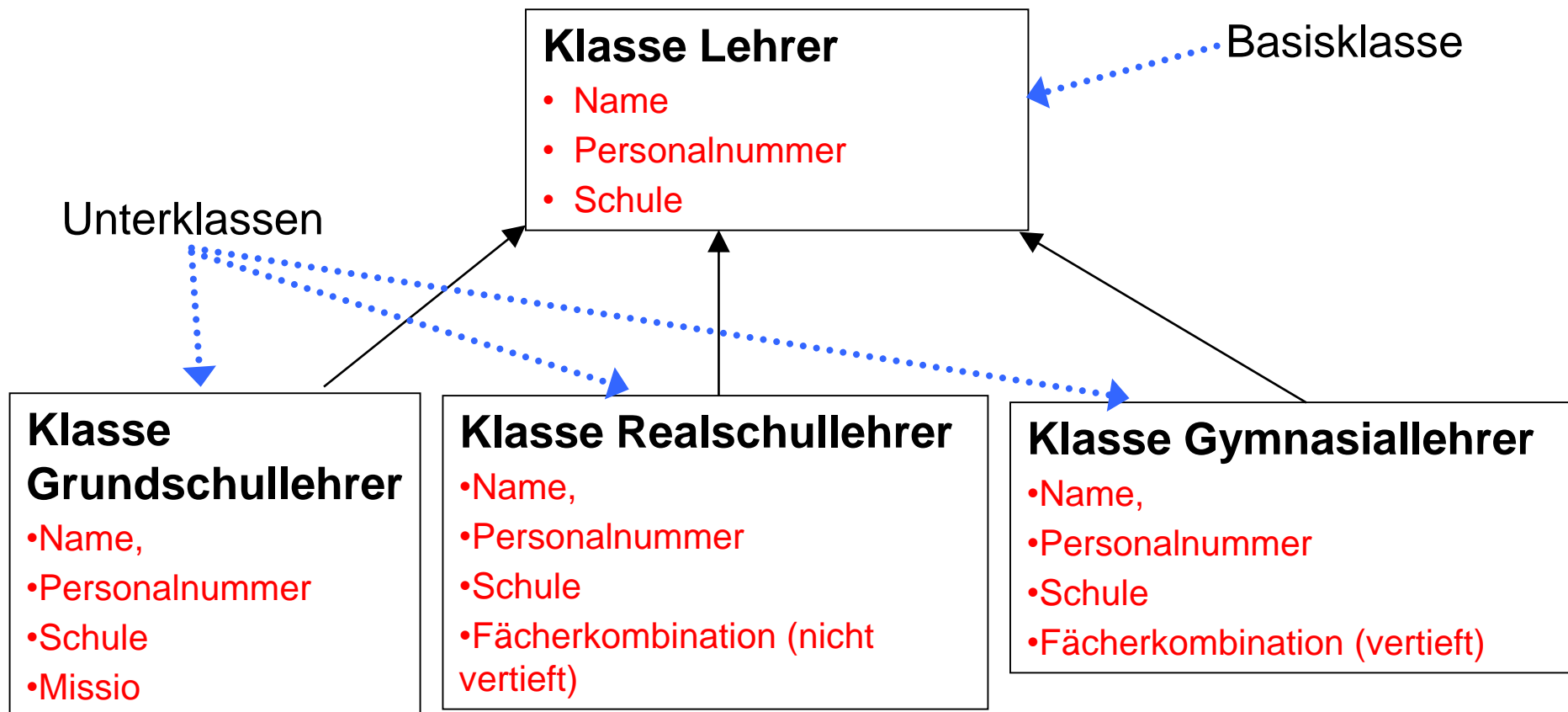
**Gemeinsamkeiten mehrerer Klassen sollen in einer gemeinsamen Oberklasse zusammengefasst werden. Dabei sollen die Unterklassen gemeinsame Eigenschaften an die neue Oberklasse „abgeben“ aber nach außen ihre Gesamteigenschaften behalten.**

**Anmerkung: Dieser Fall ist eher selten**

# Grundbegriffe – Vererbung

## Beispiele

- Die Klasse „Lehrer“ soll die Gemeinsamkeiten von Grundschullehrer, Realschullehrer und Gymnasiallehrer zusammenfassen.



## Grundbegriffe – Vererbung

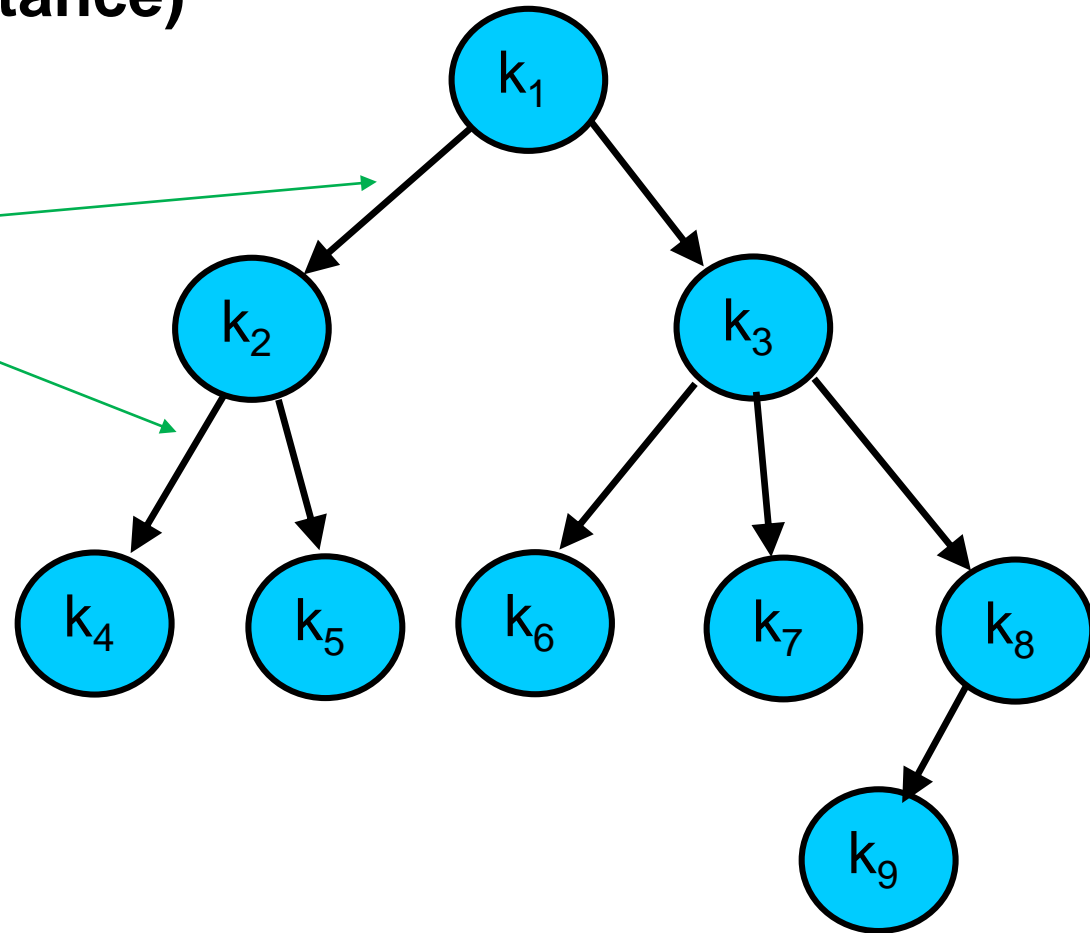
### Vererbung (Inheritance)

- **grundlegendes Konzept der Objektorientierung**
- **basierend auf existierenden Klassen werden neue Klassen definiert. Die Beziehung zwischen Ausgangsklassen und den neuen ist dabei dauerhaft.**
- **meist eine 1-zu-n Beziehung, d.h. aus einer Klasse werden mehrere andere Klassen abgeleitet.**
- **Mehrstufige Vererbung kann zu baumartigen Vererbungsstrukturen führen.**

# Grundbegriffe – Vererbung

## Vererbung (Inheritance)

Vererbungsbe-  
ziehung



## Grundbegriffe – Vererbung

### Vererbung (Inheritance)

- Eine neue Klasse kann dabei eine Erweiterung oder eine Einschränkung der ursprünglichen Klasse sein.
- Die vererbende Klasse wird Basisklasse (auch Super-, Ober- oder Elternklasse, Superclass, Baseclass) genannt
- Die (erbende) abgeleitete Klasse wird als Unter- (auch Sub-, oder Kind-) Klasse bezeichnet.



## Grundbegriffe – Vererbung

### Vererbung (Inheritance)

- **Die Ableitung einer Unterklasse aus einer Oberklasse**  
**Vererbung stellt eine Spezialisierung dar (ein Girokonto ist eine spezielle Form eines Kontos).**
- **Wird aus Basisklassen eine neue Oberklasse abgeleitet, so wird generalisiert oder verallgemeinert (ein Konto ist allgemeiner als ein Girokonto).**

## Grundbegriffe – Vererbung

### Vererbung (Inheritance)

- **Abgeleitete Klasse und Basisklasse stehen typischerweise in einer „ist-ein“-Beziehung zueinander, d.h. ein Element der Unterklasse ist immer auch ein Element der Oberklasse (aber nicht umgekehrt!)**  
**z.B. ist ein Grundschullehrer immer auch ein Lehrer, umgekehrt gilt das nicht, da ein Lehrer auch ein Gymnasial- oder Realschullehrer sein kann.**

## Vererbung

### Vererbung (Inheritance)

- **Die Unterklasse erbt „alles“ von der Oberklasse, d.h. alle Attribute und Methoden der Oberklasse sind in der Unterklasse ebenfalls vorhanden.**
- **Die Unterklasse erbt auch alle Änderungen der Oberklasse, d.h. wird die Oberklasse geändert, ändert sich automatisch auch die Unterklasse\*)**
  - ➔ **Basisklassen nur ändern wenn unvermeidlich, z.B. Fehlerbehebung**
  - ➔ **Vererbungsbeziehungen dokumentieren**

\*) technisch durch erneutes Compilieren und Linken

## Vererbung

### Vererbung (Inheritance)

- **Die Oberklasse „kennt“ ihre Unterklassen nicht  
→ Vererbungsbeziehungen dokumentieren**
- **Der Zugriffsschutz der Oberklasse bleibt immer erhalten, d.h. gekapselte Elemente der Oberklasse sind auch in der Unterklasse gekapselt.**

## Vererbung

### Vererbung (Inheritance) – Nutzen / Vorteile

- **definierte, systematische Erweiterung bestehender Software-Bausteine (Klassen), ohne die bestehende Software verändern zu müssen**
  - **Eigenschaften können zu einer Klasse hinzugefügt werden (eine neue Klasse entsteht), ohne, dass die Ursprungs-klasse und ihre Anwender von der Änderung betroffen sind**
  - **definierte, systematische Möglichkeit, Gemeinsamkeiten von Software-Bausteinen gemeinsam zu verwalten**
- ➔ „einfache“ Wiederverwendung bestehender Software**

## Vererbung

### Vererbung (Inheritance) – Implementierung in C++

```
class oberklasse
{ protected:
    ... /* dieser Teil ist auch in der Unterklasse bekannt */
    public:
    ... /* dieser Teil ist „überall“ bekannt */
};
```

Schlüsselwort **protected** bedeutet, dass die nachstehenden Identifier in einer Unterklasse bekannt sind.

```
class unterklasse : public oberklasse
{ protected: ...
    public: ...
    private: ...
};
```

Das Schlüsselwort **public** bedeutet, dass die public Elemente der Oberklasse auch in der Unterklasse public sind und dass die protected Elemente der Oberklasse auch in der Unterklasse protected, sind.

Neue Elemente der Unterklasse, hier als protected bzw. public bzw. private definiert.

## Vererbung

### Implementierung in C++

```
class lehrer
{ protected:
  char name[31]; // Name des Lehrers max. Länge 30
  char schule[31]; // Name der Schule max. Länge 30
  char personalnummer[6];
```

Schlüsselwort **protected** bedeutet, dass die nachstehenden Identifier in einer Unterklasse bekannt sind.

**public:**

```
void set_name(char lehrername[31])
{ strcpy(name,lehrername);
};
```

```
char* get_name()
{ return name;
};
```

```
// weitere Methoden zum Schreiben und Lesen anderer Attribute werden hier nicht beschrieben
};
```

# Vererbung

## Implementierung in C++

```
class gymnasiallehrer : public lehrer
```

```
{ protected:
```

```
    char fach1[20];
```

```
    char fach2[20];
```

```
public:
```

```
    void set_fach1(char fach[20])
```

```
    { int i;
```

```
      for (i=0; i<20; i++)
```

```
        fach1[i] = fach[i];
```

```
    };
```

```
    char* get_fach1()
```

```
    { return fach1;
```

```
    };
```

```
// weitere Methoden zum Schreiben und Lesen anderer Attribute werden hier nicht beschrieben
```

```
};
```

Klasse Gymnasiallehrer ist Unterklasse der Klasse Lehrer. Das Schlüsselwort **public** bedeutet, dass die public Elemente der Klasse lehrer auch in der Unterklasse public sind und dass die protected Elemente der Klasse lehrer auch in der Unterklasse protected, sind.

Neue Elemente der Unterklasse, hier als protected bzw. public definiert.



## Vererbung

- **C++ bietet spezielle Möglichkeiten, die Vererbung einzuschränken**
- **Einschränkungen können dabei sowohl von der Basisklasse als auch von der Unterklasse definiert werden.**
- **Diese Einschränkungen gehen über die ursprüngliche Intention der Objektorientierung (Oberklasse vererbt alles) hinaus.**

# Vererbung

## Implementierung in C++ (Datenkapselung)

```
class basisklasse
{
    private:
        int privates_attribut;
        void private_methode()
        { ... };

    protected:
        int protected_attribut;
        void protected_methode()
        { ... };

    public:
        int public_attribut;
        void public_methode()
        { ... };
};
```

Schlüsselwort **private** bedeutet, dass die folgenden Bezeichner nur innerhalb der Klasse bekannt sind. Die Bezeichner werden zwar vererbt, sind aber nicht zugänglich.

Schlüsselwort **protected** bedeutet, dass die nachstehenden Bezeichner in der Klasse selbst und ggf. in einer Unterklasse bekannt sind.

Schlüsselwort **public** bedeutet, dass die nachstehenden Bezeichner in der Klasse selbst und auch ausserhalb der Klasse bekannt sind.

## Vererbung

### Implementierung in C++ (Art der Vererbung)

```
class basisklasse
```

```
{  
    // wie zuvor  
};
```

```
class public_unterklasse : public basisklasse
```

```
{  
    ...  
};
```

Das Schlüsselwort **public** bedeutet, dass die public Elemente der Oberklasse auch in der Unterklasse public sind und dass die protected Elemente der Klasse lehrer auch in der Unterklasse protected, sind.

Private Elemente der Oberklasse bleiben privat.

## Vererbung

### Implementierung in C++ (Art der Vererbung)

```
class basisklasse
```

```
{  
    // wie zuvor  
};
```

```
class private_unterklasse : private basisklasse
```

```
{  
    ...  
};
```

```
class protected_unterklasse : protected basisklasse
```

```
{  
    ...  
};
```

Das Schlüsselwort **private** bedeutet, dass die public und protected Elemente der Oberklasse in der Unterklasse private werden.

Private Elemente der Oberklasse bleiben privat.

Das Schlüsselwort **protected** bedeutet, dass die public und protected Elemente der Oberklasse in der Unterklasse protected werden.

Private Elemente der Oberklasse bleiben privat.

## Vererbung in C++

### Zusammenfassung

<div> <div>Ableitung</div> <div>Kapselung in Basisklasse</div> </div>			
	private	protected	public
	Kein Zugriff	Kein Zugriff	Kein Zugriff
	protected	protected	protected
public	private	protected	public

## Vererbung in C++

**Zum Schluss dieses Abschnitts ...**

**Noch Fragen ??**

## Vererbung und Redefinition

- **Redefinition ist die Möglichkeit, in einer Unterklasse ererbte Elemente mit identischem Namen in anderer Form neu zu definieren.**
- **Sowohl Attribute als auch Methoden können redefiniert werden (müssen aber nicht !)**
- **Attribute werden meist mit einem anderen Typ/Struktur redefiniert**
- **Redefinierte Methoden haben meist einen anderen Algorithmus, z.B. um neue oder auch redefinierte Attribute zu berücksichtigen**

## Vererbung und Redefinition

- **Abhängig davon, zu welcher Klasse ein Objekt gehört, (Ober oder Unterklasse) wird über den Namen der Element und die Klassenzugehörigkeit das richtige Element ausgewählt.**
- **Die Auswahl der richtigen Methode bei Redefinition geschieht statisch bei Aufruf einer Methoden an einem Objekt. D.h. Am Objekt der Oberklasse wird die Oberklassenmethode aufgerufen, am Objekt der Unterklasse die redefinierte Unterklassenmethode**



## Vererbung und Redefinition

- **Beispiel:**
  - **in einem Lohnabrechnungsprogramm wird das Gehalt der Mitarbeiter einer Firma berechnet. Dabei wird zwischen fest angestellten Mitarbeitern und Werkstudenten unterschieden. Die Klasse Werkstudent wird als Unterklasse der Klasse Mitarbeiter implementiert.**

# Vererbung und Redefinition

## Implementierung in C++

```
const float steuersatz = 20.0/100;
```

```
const float sozialabgaben = 15.0/100;
```

```
class Mitarbeiter
```

```
{ protected:
```

```
    char name[30];
```

```
    unsigned long bruttogehalt;
```

```
    public:
```

```
    Mitarbeiter(const char n[30], unsigned long b)
```

```
    { strcpy(name, n);
```

```
      bruttogehalt = b;
```

```
    };
```

```
    float berechne_netto Gehalt(void)
```

```
    { return ((bruttogehalt * (1-sozialabgaben)) * (1-steuersatz));
```

```
    };
```

```
};
```

```
class Werkstudent : public Mitarbeiter
```

```
{ public:
```

```
    Werkstudent(const char* n, unsigned long b) :
```

```
        Mitarbeiter(n,b)
```


```
    { };
```

```
    float berechne_netto Gehalt(void)
```

```
    { return (bruttogehalt * (1-steuersatz)); // nur Steuer
```

```
    };
```

```
};
```



Redefinition der ererbten Funktion  
berechne\_netto Gehalt (...).

# Vererbung und Redefinition

## Implementierung in C++

```
int main(void)
{
    Mitarbeiter m("Hans Meier", 10000);
    Werkstudent s("Haenschen Mueller", 2500);
    ...
    ... = m.berechne_netto Gehalt();
    ... = s.berechne_netto Gehalt();
    ...
}
```

Aufruf der Oberklassenmethode

Aufruf der Unterklassenmethode

## Vererbung und Redefinition

- **Sowohl Attribute als auch Methoden können redefiniert werden.**
- **Bei der Redefinition einer Methode wird der Name und die Signatur, d.h. die Schnittstelle der Methode (Ergebnistyp und Parameter) beibehalten.**
- **Klassen können namensgleiche Methoden unterschiedlicher Signatur haben.**

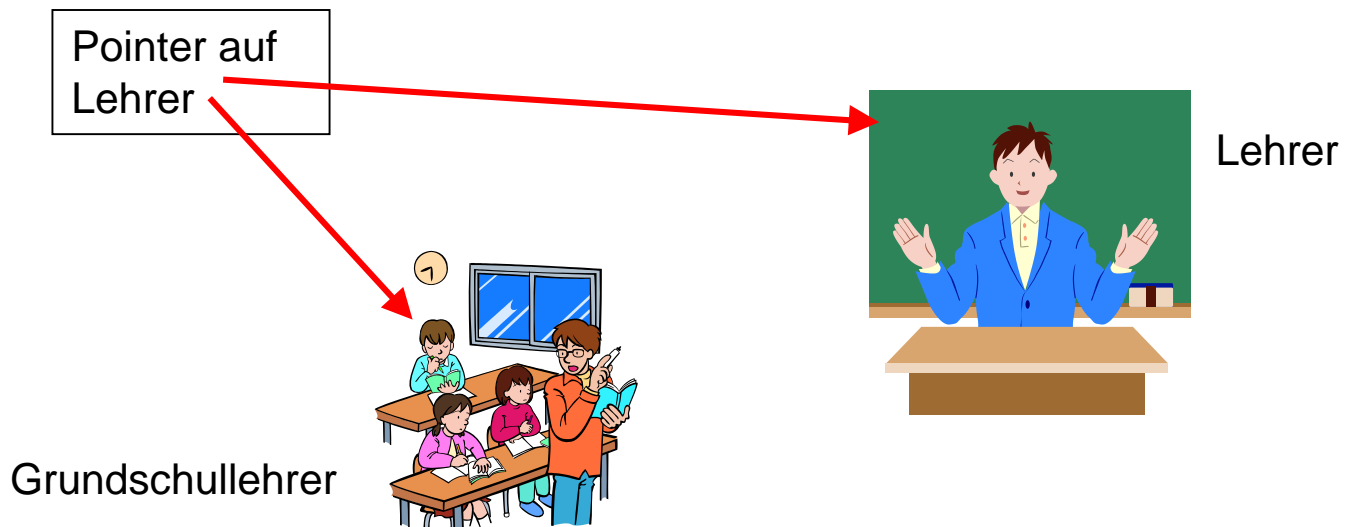
## Vererbung und Redefinition

**Zum Schluss dieses Abschnitts ...**

**Noch Fragen ??**

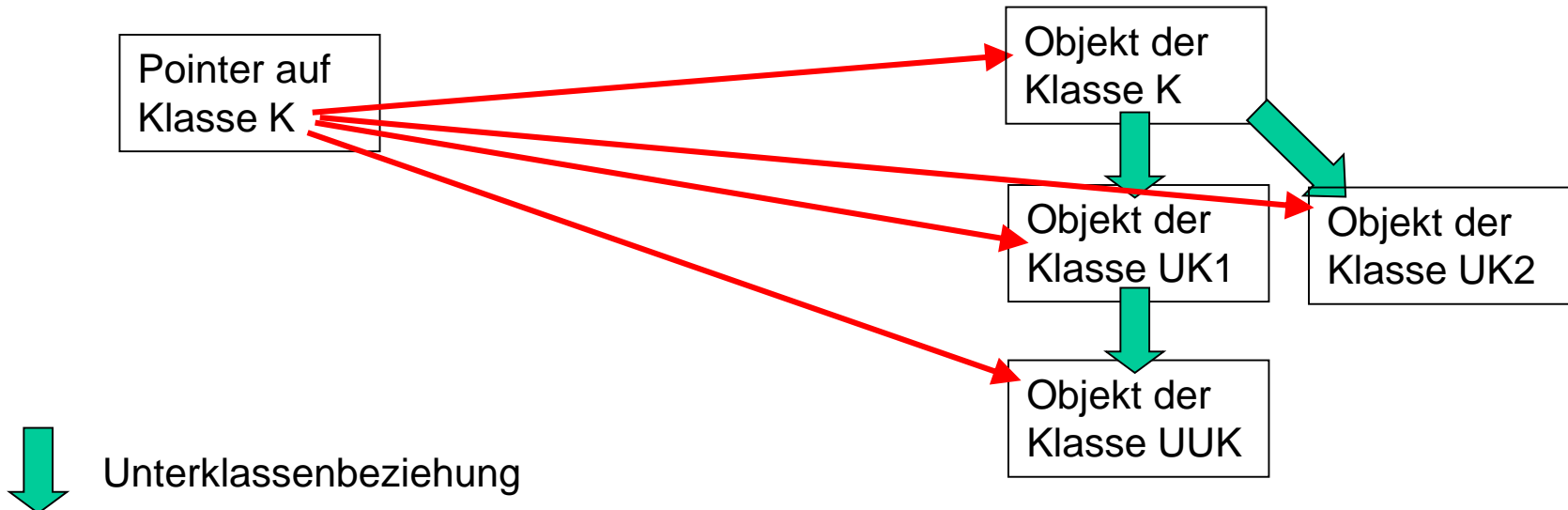
## Polymorphismus und dynamisches Binden

- **Unterlassenobjekte sind per Definition auch Elemente ihrer Oberklassen**  
z.B. ein Grundschullehrer ist immer auch ein Lehrer.
- **Analog dazu kann ein Zeiger auf ein Oberlassenobjekt auch auf ein Unterlassenobjekt zeigen.**  
(Umgekehrt gilt das nicht!)



## Polymorphismus und dynamisches Binden

- Zeiger sind in diesem Sinne „vielgestaltig“, d.h. polymorph.
- Polymorphismus im Sinne der Objektorientierung ist die Tatsache, dass ein Zeiger auf ein Objekt einer Klasse auch auf Objekte aller Unterklassen zeigen kann.



## Polymorphismus und dynamisches Binden

- Methode ist in einer Unterklasse redefiniert
  - Die redefinierte Methode wird an einem Oberklassenpointer aufgerufen
  - Zur Laufzeit wird entschieden, welche Methode aufgerufen wird (das „Original“ der Oberklasse oder die redefinierte Methode in der Unterklasse).
  - Die Entscheidung, welche Methode ausgeführt wird, hängt davon ab, auf welche Klasse der Pointer zum Aufrufzeitpunkt zeigt
- ➔ Dieser Vorgang wird als dynamisches Binden bezeichnet



# Polymorphismus und dynamisches Binden

```
class Oberklasse
```

```
{ ... ;
```

```
  public:
```

```
    virtual void methode() { ... }; ←
```

```
    ...
```

```
};
```

```
class Unterklasse : public Oberklasse
```

```
{ ... ;
```

```
  public:
```

```
    void methode() { ... }; ←
```

```
    /* Redefinition der Oberklassenmethode */
```

```
    { ... };
```

```
    ...
```

```
};
```

```
Oberklasse* OK_Pointer;
```

```
...
```

```
OK_Pointer = ... ;
```

```
...
```

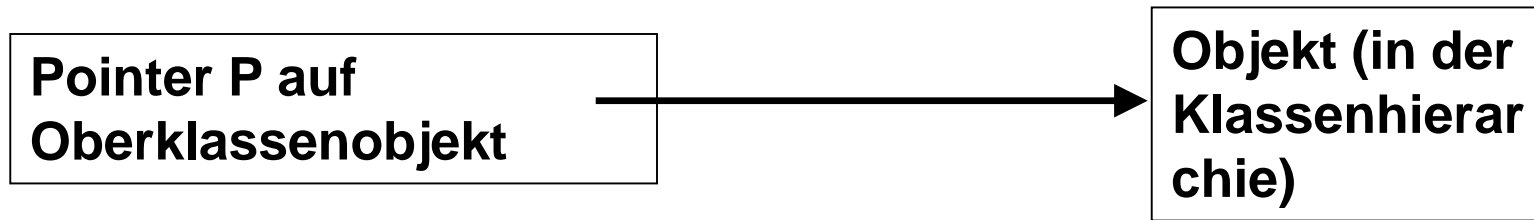
```
OK_Pointer->methode();
```

Aufruf einer redefinierten Methode an einem Oberklassenpointer

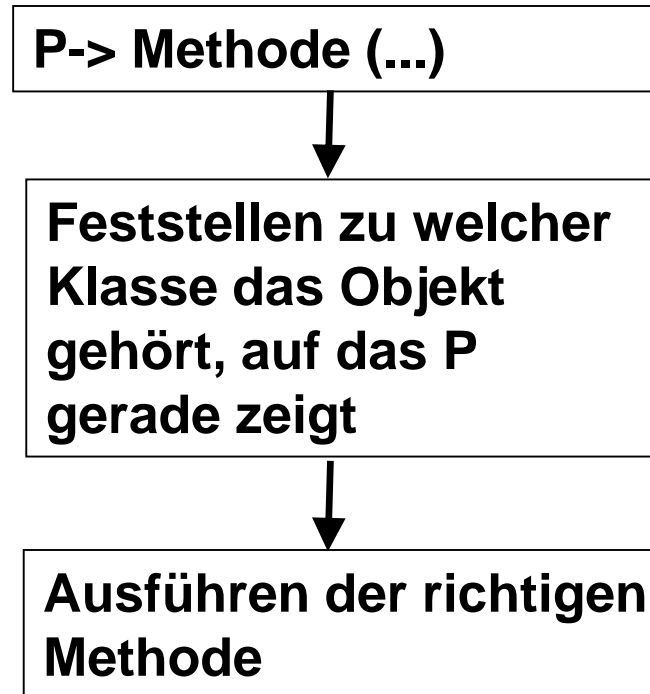
→ Zur Laufzeit (!) des Programms wird beim Aufruf der Methode festgestellt, auf welche Klasse OK\_Pointer zeigt. Zeigt OK\_Pointer auf ein Objekt der Oberklasse wird methode() der Oberklasse gerufen.

Zeigt OK\_Pointer auf ein Objekt der Unterklasse wird die redefinierte Methode der Unterklasse ausgeführt.

# Polymorphismus und dynamisches Binden



## Ablauf des Dynamischen Bindens



## Polymorphismus und dynamisches Binden

- **Zur Entscheidung, welche Methode beim dynamischen Binden ausgeführt wird, führt der Compiler Buch, auf welche Klasse ein Pointer zeigt und Sprungtabellen an, die die Anfangsadressen der dynamisch zu bindenden Methoden enthält.**

Klasse OK	Addr. m(..)
Klasse UK1	Addr. m(..)
Klasse UK2	Addr. m(..)
Klasse UK3	Addr. m(..)
Klasse UK4	Addr. m(..)

## Polymorphismus und dynamisches Binden

- **Dynamisches Binden findet immer an einem Pointer statt. `p_variable -> methode(...)`; (ein Objekt ist nicht polymorph)**
- **Dynamisches Binden findet nur bei Methoden statt (in C++ Programmen definierte C-Funktionen sind nicht geeignet)**
- **Dynamisches Binden überlässt die Entscheidung, welche Funktion aufgerufen wird, dem Laufzeitsystem und erspart damit explizite Typumwandlung**

## Polymorphismus und dynamisches Binden



- **Spätere Erweiterungen der Klassenhierarchie werden (nach Neucompilierung) automatisch berücksichtigt.**
- **Der „Gegensatz“ zu Dynamischen Binden ist statisches Binden, das bereits zur Compilezeit stattfindet.**

## Polymorphismus und dynamisches Binden

- **Nachteile:**
  - **zusätzlicher Speicherplatzverbrauch (jedes Objekt muss wissen, zu welcher Klasse es gehört)**
  - **zusätzliche Laufzeit (bei einem Methodenaufruf an einem Pointer muss zuerst die Klassenzugehörigkeit festgestellt werden, dann noch die richtige Methode ausgewählt werden)**
- ➔ **für Anwendungen mit begrenztem Speicher und laufzeitkritischen Anwendungen nicht oder nur mit Vorsicht zu verwenden!**

## Polymorphismus und dynamisches Binden

- **Implementierung in C++**
  - **in C++ muss eine Methode, die redefiniert wird und die dynamisch gebunden werden soll, mit dem Schlüsselwort virtual gekennzeichnet werden.**
  - **Fehlt das Schlüsselwort virtual, wird immer statisch gebunden, d.h. es wird die Methode der Klasse ausgeführt, auf die der Pointer statisch zeigt (Typ des Pointers).**

# Polymorphismus und dynamisches Binden

## Implementierung in C++

```
const float steuersatz = 20.0/100;
```

```
const float sozialabgaben = 15.0/100;
```

```
class Mitarbeiter
```

```
{ protected:
```

```
    char name[30];
```

```
    unsigned long bruttogehalt;
```

```
public:
```

```
    Mitarbeiter(const char n[30], unsigned long b)
```

```
    { strcpy(name, n);
```

```
      bruttogehalt = b;
```

```
};
```

```
virtual float berechne_netto Gehalt(void)
```

```
{ return ((bruttogehalt * (1-sozialabgaben)) * (1-steuersatz));
```

```
};
```

```
};
```

Kennzeichnung der Methode durch  
Schlüsselwort **virtual**

```
class Werkstudent : public Mitarbeiter
```

```
{ public:
```

```
    Werkstudent(const char* n, unsigned long b) :
```

```
        Mitarbeiter(n,b)
```

```
    { };
```

```
    float berechne_netto Gehalt(void)
```

```
    { return (bruttogehalt * (1-steuersatz)); // nur Steuer
```

```
    };
```

```
};
```

Redefinition der ererbten Funktion  
berechne\_netto Gehalt (...).



## Vererbung und Redefinition

### Implementierung in C++

```
int main(void)
```

```
{  
    Mitarbeiter m1("Hans Meier", 10000);  
    Mitarbeiter m2("Sepp Huber", 20000);  
    Werkstudent s("Haenschen Mueller", 2500);  
    Mitarbeiter* p_m = &m;  
    p_m = &m1;  
    ... = p_m->berechne_netto Gehalt();  
    p_m = &s;  
    ... = p_m->berechne_netto Gehalt();  
    p_m = &m2;  
    ... = p_m->berechne_netto Gehalt();  
}
```

p\_m zeigt auf Oberklassenobjekt →  
Aufruf der Oberklassenmethode

p\_m zeigt jetzt auf  
Unterklassenobjekt → Aufruf der  
Unterklassenmethode

p\_m zeigt jetzt wieder auf  
Oberklassenobjekt → Aufruf der  
Oberklassenmethode

## Vererbung und Redefinition

**Zum Schluss dieses Abschnitts ...**

**Noch Fragen ??**

## Pure Virtual Methoden und Abstrakte Klassen

### Motivation: Beispiel Grafische Elemente

- **Es soll eine Klassenhierarchie definiert werden, die die Darstellung einfacher grafischer Elemente (Linie, Kreis, Quadrat, Rechteck, ... ) am Bildschirm ermöglicht.**
- **Für jedes Element soll eine eigene Klasse, abgeleitet von der Basisklasse Grafikelement, definiert werden.**
- **Jede Unterklasse der Klasse Grafikelement soll eine Ausgabefunktion zeichne() implementieren, die die Ausgabe am Bildschirm durchführt.**

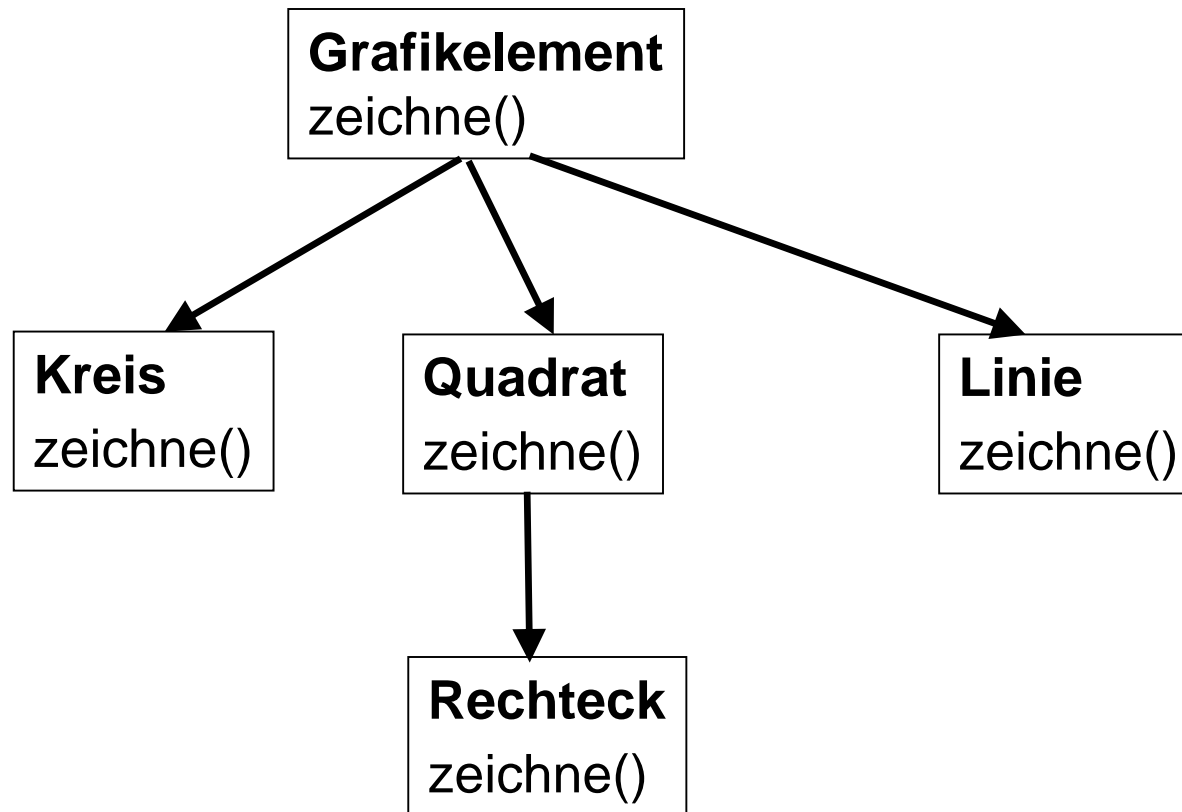
## Pure Virtual Methoden und Abstrakte Klassen

### Motivation: Beispiel Grafische Elemente

- Die Zuordnung der Ausgabefunktion zu einem über Pointer verwalteten Grafikelement soll zur Laufzeit erfolgen.
- Die Klasse Grafikelement soll/muss dabei nicht darstellbar sein, sondern nur ihre Unterklassen

## Pure Virtual Methoden und Abstrakte Klassen

### Motivation: Beispiel Grafische Elemente



## Pure Virtual Methoden und Abstrakte Klassen

### **Verallgemeinert (Motivation)**

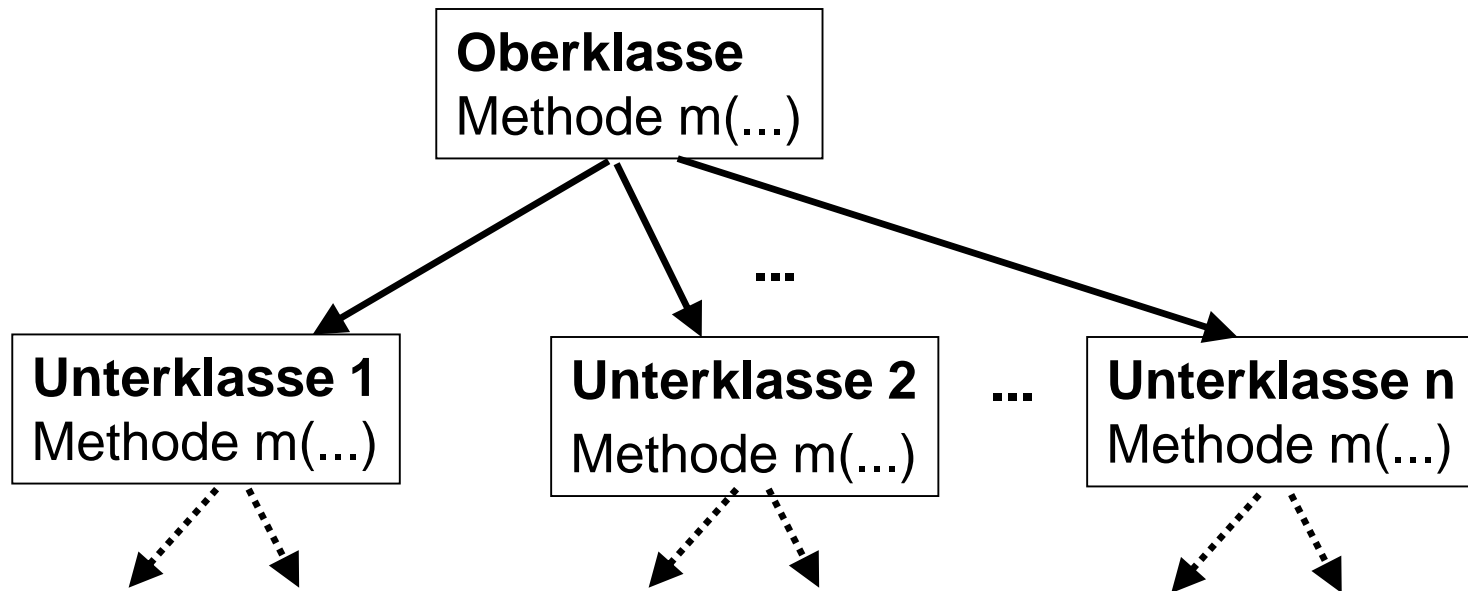
**Es soll eine Klassenhierarchie mittels Vererbung definiert werden, bei der in allen Unterklassen eine bestimmte Methode redefiniert werden muss.**

**Bei Bedarf soll durch dynamisches Binden zur Laufzeit entschieden werden, welche Implementierung der ausgeführt wird.**

**Die Oberklasse soll dabei möglichst abstrakt formuliert werden und Objekte der Oberklasse müssen nicht unbedingt instanziiert werden.**

## Pure Virtual Methoden und Abstrakte Klassen

### Verallgemeinert (Motivation)



### Lösung: Rein virtuelle (pure virtual) Methoden

## Pure Virtual Methoden und Abstrakte Klassen

- **Pure Virtual Methoden (rein virtuelle Methoden) definieren nur eine Schnittstelle und besitzen keinen Methodenrumpf**
- **Sie sind somit nicht ausführbar**
- **Klassen mit Pure Virtual Methoden werden auch als Abstrakte Klassen bezeichnet**
- **Abstrakte Klassen können nicht instanziiert werden, d.h. es können keine Objekte dieser Klasse erzeugt werden**
- **Pointer auf Abstrakte Klassen sind möglich**



# Polymorphismus und dynamisches Binden

## Implementierung in C++

```
class Abstrakte_Klasse
```

```
{ ...  
  public:  
    virtual Virtuelle_Methode(int i) = 0;  
  ...  
};
```

Kennzeichnung der rein virtuellen Methode durch Schlüsselwort **virtual** und Zuweisung des Werts 0

➔ von dieser Klasse können keine Objekte vereinbart werden.

```
class Konkrete_Klasse : public Abstrakte_Klasse
```

```
{ protected:  
  ...  
  public:  
    Virtuelle_Methode(int i)  
    { ...  
    };  
  ...  
};  
int main()  
{ Abstrakte_Klasse *p;  
  ...  
}
```

Hier wird die rein virtuelle Methode der Abstrakten Oberklasse implementiert.  
Von dieser Klasse können Objekte vereinbart werden.

Pointer auf Abstrakte\_Klasse können definiert werden

## Motivation

**Zum Schluss dieses Abschnitts ...**

**Noch Fragen ??**

## Vererbung und Konstruktoren

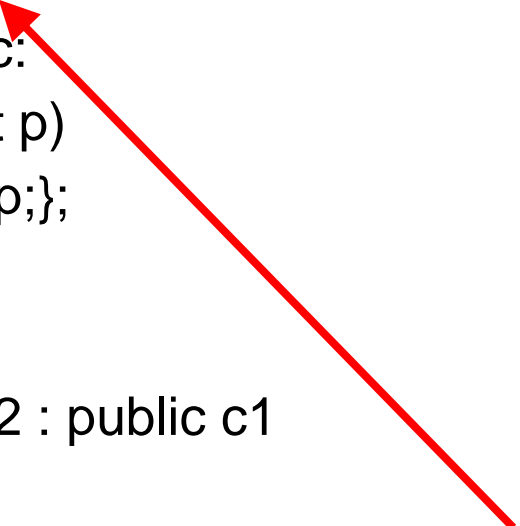
- **Konstruktoren werden beim Erzeugen eines Objekts implizit aufgerufen (ggf. unterschieden nach Parametern)**
- **Beim Aufruf des Konstruktors einer Unterklasse wird (wenn nicht anders angegeben und vorhanden) der parameterlose Konstruktor der Oberklasse implizit aufgerufen**
- **Gibt es keinen parameterlosen Konstruktor oder mehrere Konstruktoren in der Oberklasse, muss der vorhandene oder gewünschte Konstruktor der Oberklasse im Konstruktor der Unterklasse angegeben werden.**

## Vererbung und Konstruktoren

**ein weiteres (damit verwandtes) Problem ...**

```
class c1
{ private:
  int i;
  public:
    c1(int p)
      {i = p;};
};

class c2 : public c1
{ ...
  // c2 kann auf ererbte, aber private Oberklassenattribute nicht
  // zugreifen
```

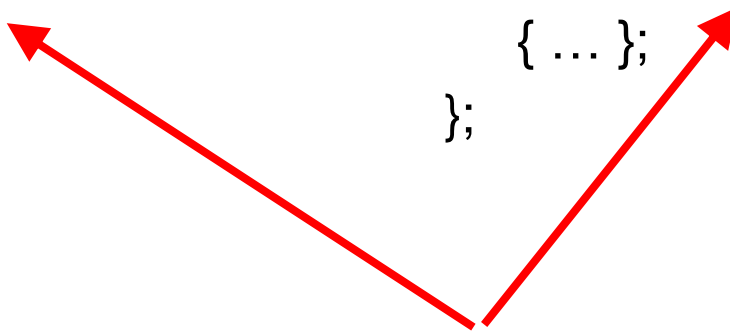


## Vererbung und Konstruktoren

### Lösung des Problems: Aufruf des Oberklassenkonstruktors im Konstruktor der Unterklasse

```
class c1
{ private:
  int i;
  public:
  c1(int p)
  { i = p; };
};
```

```
class c2 : public c1
{ ...
  public:
  c2(int p) : c1(p)
  { ... };
};
```



Aufruf des Oberklassenkonstruktors bei Ausführung des Konstruktors der Unterklasse durch ':' getrennt (ähnlich Memberinitialisierung)

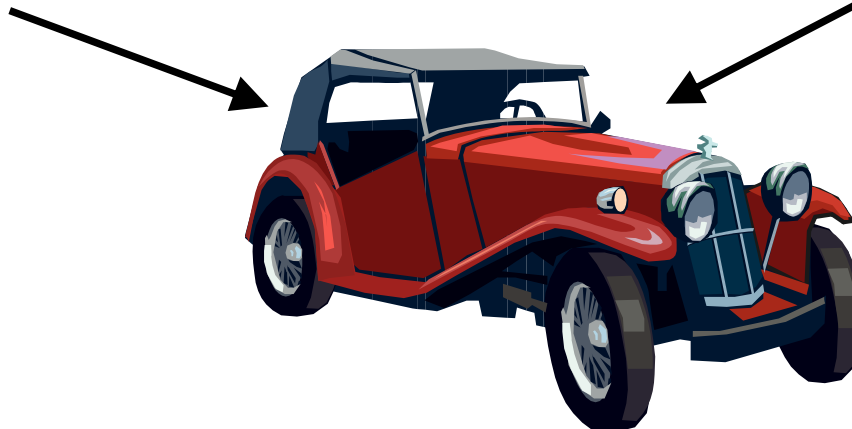
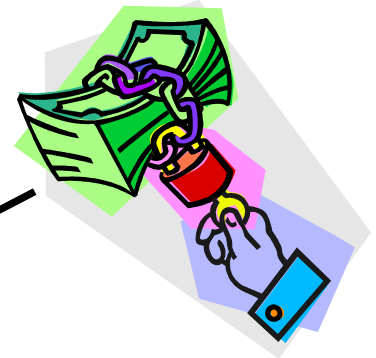
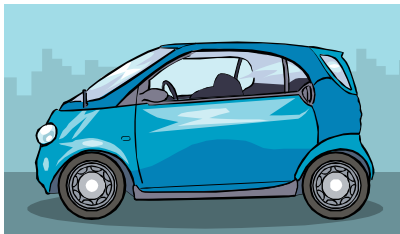
## Motivation

**Zum Schluss dieses Abschnitts ...**

**Noch Fragen ??**

## Mehrfachvererbung

- Was passiert, wenn ich nicht nur von einer Oberklasse erben möchte?
- Beispiel:  
Ein Oldtimer soll definiert werden zum einen als Auto, zum anderen als Geldanlage.



## Mehrfachvererbung

- **Lösung: Mehrfachvererbung, d.h. die Klasse Oldtimer erbt sowohl von der Klasse Auto als auch von der Klasse Geldanlage.**
- **Ein Oldtimer hat damit sowohl die Eigenschaften eines Autos als auch die Eigenschaften einer Geldanlage.**
- **Mehrfachvererbung wird nicht in allen Programmiersprachen unterstützt.**



## Mehrfachvererbung

### Implementierung in C++

```
class geldanlage
{ protected:
    short rendite_pro_jahr;
    unsigned long anlagebetrag;
public:
    geldanlage(unsigned long betrag, short rendite)
    { rendite_pro_jahr = rendite;
      anlagebetrag = betrag;
    };
    // ... weitere Memberfunktionen ...
};
```

```
class automobil
{ protected:
    unsigned int hubraum;
    unsigned int leistung;
    unsigned int geschwindigkeit;
public:
    automobil(unsigned int h, unsigned int l,
              unsigned int g)
    { hubraum = h;
      leistung = l;
      geschwindigkeit = g;
    };
    // ... weitere Memberfunktionen
};
```

## Mehrfachvererbung

### Implementierung in C++

```
class oldtimer: public automobil , public geldanlage
{
public:
    oldtimer (unsigned int h, unsigned int l, unsigned int g,
              unsigned long b, short r) : automobil(h,l,g), geldanlage(b,r)
    {
    };
    /// weitere Memberfunktionen
};
```

## Mehrfachvererbung

- **Einfachvererbung führt zu baumartigen Vererbungsstrukturen (d.h. Vererbungspfade laufen nicht zusammen, nur auseinander). Mehrfachvererbung bildet Vererbungsgraphen (d.h. Vererbungspfade können auch wieder zusammenlaufen)**
- **Namensgleichheit von Attributen oder Methoden in Oberklassen erzeugt Konflikte (auf welches Element soll zugegriffen werden?) → Mehrfachvererbung vermeiden oder Auflösung durch qualifizierten Zugriff (Klasse::Bezeichner)**

**Zum Schluss dieses Abschnitts ...**

**Noch Fragen ??**

# Template Klassen

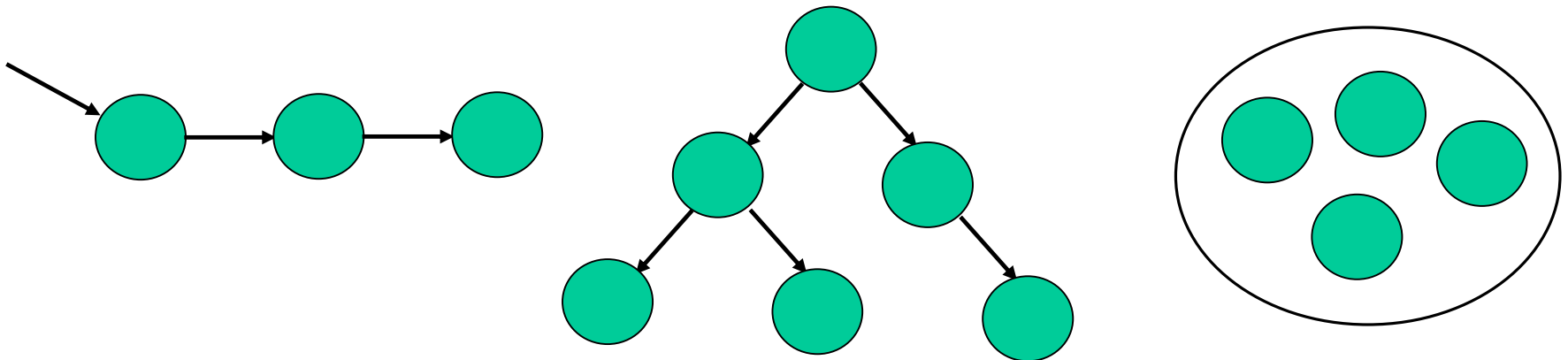
- **Motivation**
  - die Datenstruktur Array (Vektor) nimmt Elemente beliebiger Datentypen auf bzw. „speichert“ diese.
  - Zugriff auf Elemente erfolgt „standardisiert“ über den Index
  - ➔ Arrays bilden eine Art „Behälter“ („Container“) für Elemente anderer Datentypen
  - ➔ die Struktur Array ist dabei generisch, d.h. die Struktur und das Verhalten ist unabhängig vom Typ und vom Verhalten der gespeicherten Elemente

# Template Klassen

- **Motivation**
  - **Andere Datentypen wie Listen oder Bäume „speichern“ ebenfalls Elemente anderer Datentypen**
  - **Diese Datentypen sollen nur einmal definiert werden**
  - **Struktur und Verhalten dieser Datentypen ist ebenfalls unabhängig vom Datentyp der gespeichert wird**
  - **In der OOP werden Klassen verwendet**
- ➔ **Definition von generischen Klassen (Template Klassen) in der Objektorientierten Programmierung**

## Template Klassen

- **Motivation**
  - **Containerklassen sind Klassen, die Objekte anderer Klassen verwalten oder zu speichern.**
  - **Beispiele: Listen, Bäume, Mengen**



- **Verhalten der Containerklasse sollte möglichst unabhängig sein von den Klassen, deren Objekte verwaltet werden.**

## Template Klassen

- **Motivation**
  - **Konkrete Klasse, die in der Containerklasse verwaltet wird, wird nicht mehr angegeben.**
  - **Stattdessen soll ein Parameter für eine Klasse verwendet werden.**
    - ➔ **Parametrierten Containerklassen**
    - ➔ **Verallgemeinert: Parametrisierte Klassen (d.h. nicht nur Container)**
    - ➔ **Template Klassen (Schablonen) in C++**



## Template Klassen (Schablonen)

- ➔ **Konzept der Template Klassen in C++**
  - ➔ **Template Klassen sind Klassenschablonen, die formale Parameter („Platzhalter“) für eine Klasse oder für Konstante (ähnlich der formalen Parameter einer Methode) besitzen**
  - ➔ **Von einer Template Klasse können keine Objekte instanziiert werden.**
  - ➔ **Durch Ausprägen, d.h. Einsetzen von konkreten Klassen bzw. Konstanten für die formalen Klassenparameter einer Template Klasse entsteht eine neue Klasse, die instanziiert werden kann.**

# Template Klassen

- **Allgemeine Struktur:**

```
template<class Element> class Template_Klasse  
{
```

Parameter der Template Klasse.

Rumpf der Klasse, „wie gewohnt“  
Dabei darf der Klassenparameter  
**Element** wie eine reale Klasse  
verwendet werden, d.h. es dürfen  
z.B. Attribute mit dem Typ Element  
definiert werden

Schlüsselwort template  
kennzeichnet Template Klasse.

```
};
```

# Template Klassen

- **Beispiel:**

```
template<class Element> class Stack
```

Schlüsselwort template  
kennzeichnet Template Klasse.

Formaler Parameter der Template  
Klasse.

```
{
```

```
    Element a[100];
```

Attribut vom Typ des Template  
Parameters

```
    int stacksize; /* grÖÖe des Stacks */
```

```
    int elements; /* Anzahl der Elemente im Stack */
```

```
    public:
```

```
    Stack (); /* Konstuktor */ { ... };
```

```
    ~Stack (); { ... };
```

„Normales“ Attribut

```
    bool isempty() { return 0==elements; }
```

```
    Element top() { ... };
```

```
    Push (Element a) { ... };
```

```
    Pop() { ... };
```

Parameter einer Methode vom Typ  
des Template Parameters

```
};
```

## Template Klassen (Schablonen)

- **Konzept der Template Klassen in C++**
  - **Parametrisierte Klassenschablone (Parameter ist eine Klasse (ggf. auch mehrere))**
  - **Innerhalb einer Template Klasse dürfen Methoden und (überladene) Operatoren der Parameterklasse e aufgerufen werden.**
  - **Durch Ausprägen, d.h. Einsetzen einer existierenden Klasse für den Klassenparameter in die Template Klasse entsteht eine neue Klasse**
  - **Alle Methoden und Attribute der Template Klasse sind in der Ausprägung entsprechend vorhanden**

## Template Klassen (Schablonen)

- **Konzept der Template Klassen in C++**
  - **Innerhalb einer Template Klasse dürfen Methoden und (überladene) Operatoren der ausprägenden Klasse aufgerufen werden.**
  - ➔ **Die Template Klasse läßt sich dann nur mit aktuellen Parametern (Klassen) ausprägen, die diese Methoden und Operatoren auch besitzen.**
  - **Zum Ausprägen dürfen auch Standarddatentypen (int, float, long, usw.) verwendet werden, sofern die im Template verwendeten Operatoren für Klassenparameter existieren.**

## Template Klassen

- **Beispiel:**

```
template<class Element> class Stack  
{ /* wie zuvor */  
};
```

```
/* Ausprägung als Objekt */
```

```
Stack <int> i_S; /* Objekt i_S auf Basis der Template Klasse Stack */
```

```
Stack <int> *p_i_S; /* Pointer auf Objekt auf Basis der Template Klasse */
```

```
i_S.push(7); /* Aufruf einer Methode der Ausprägung */
```

## Template Klassen

- **Beispiel:**

```
template<class Element> class Stack  
{ /* wie zuvor */  
};
```

```
/* Ausprägung als Unterklasse */
```

```
class Int_Stack : public Stack<int> /* Definition der Klasse int_Stack als  
                                   Unterklasse Template Klasse Stack */  
{ /* hier leerer Klassenrumpf */};
```

## Template Klassen

- **Vorteile**
  - **Flexible, universell verwendbare Schablonen für Klassen lassen sich mittels Template Klassen erzeugen**
  - **Bei Bedarf können Methoden ausprägender Klassen aufgerufen werden (→ hohe Flexibilität)**
  - **Konzepte der Objektorientierung (Vererbung, Datenkapselung) gelten auch für Template Klassen**



## Template Klassen

- **Nachteile**
  - **Gewöhnungsbedürftige Schreibweise**
  - **Durch eine ausprägende Klasse können unerwartete Nebeneffekte auftreten (Grund: Methoden der ausprägenden Klasse können aufgerufen werden).**
  - ➔ **Daher genügt es unter Umständen nicht, nur eine Ausprägung exemplarisch zu testen.**

## Template Klassen

**Zum Schluss dieses Abschnitts ...**

**Noch Fragen ??**