



Wintersemester 2006/07

**Einführung in die Informatik für  
Naturwissenschaftler und Ingenieure**  
(alias **Einführung in die Programmierung**)  
(Vorlesung)

Prof. Dr. Günter Rudolph  
Fachbereich Informatik  
Lehrstuhl für Algorithm Engineering

**Ziele von Klassen**➤ *Schon besprochen:*

- Kapselung von Attributen (wie `struct` in Programmiersprache C)
- Kapselung von klassenspezifischen Funktionen / Methoden
- Erweiterte Möglichkeiten gegenüber `struct`
  - Konstruktoren / Destruktoren
  - Überladen von Funktionen (Methoden) und Konstruktoren
  - Überladen von Operatoren

➤ *Neu:*

- Effiziente Wiederverwendbarkeit
  - dazu: → **Vererbung**

**Modellierung von Objekten mit geringen Unterschieden****Bisherige Mittel** zum Modellieren von „ähnlichen“ Objekten:

- Sei Klasse A bereits definiert (Beispiel: Sparkonten, Kap. 9).
- Wir wollen jetzt Girokonten modellieren → Klasse B.

**Ansatz:**

- Kopiere Code von Klasse A
- Umbenennung in Klasse B
- Hinzufügen, ändern, entfernen von Attributen und Methoden

**Probleme:**

- Aufwändig bei Änderungen (z.B. zusätzlich Freistellungsbetrag für alle Konten)
- Fehleranfällig ... und langweilig!

**Alternative: Vererbung**Seien **A** und **B** Klassen:

- **A** ist Oberklasse von **B** bzw. **B** ist Unterklasse von **A**
- Wir sagen: **B** erbt Attribute und Methoden von **A**  
d.h. **B** „kennt“ Attribute und Methoden von **A**  
(Grad der Bekanntschaft wird gleich noch detailliert angegeben)
- **B** fügt neue Attribute und Methoden zu denen von **A** hinzu  
→ ggf. werden (alte) Methoden neu definiert
- Jede Instanz von **B** ist auch eine Instanz von **A**

Beispiel: Klassen KString und KVersalien

➤ Definiere Klasse KString

- Attribute
  - Zeichenkette
  - Länge
- Methoden
  - Konstruktor / Destruktor
  - SetValue, GetValue, Length, Print

extrem  
„abgespeckte“ Version der  
Klasse std::string

➤ Betrachte Unterklasse KVersalien

⇒ Für Zeichenketten, die Buchstaben nur als Großbuchstaben aufweisen!

Klasse KString: 1. Versuch

```
class KString {
private:
    char* mValue;
    int mLength;
public:
    KString(char *s);
    bool SetValue(char *s);
    char *GetValue();
    int Length();
    void Print();
    ~KString();
};
```

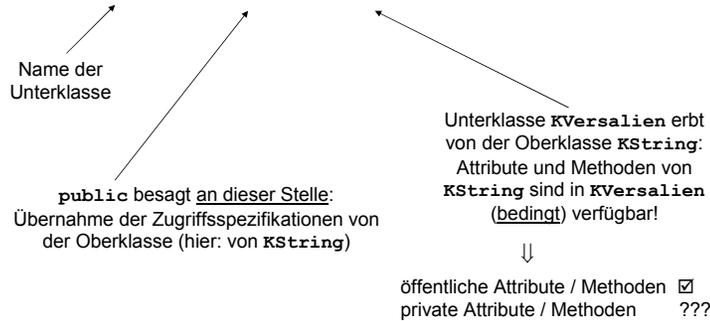
... schon ganz gut, aber ...

Zugriffsspezifikation private wird Probleme machen!

→ siehe später!

Grundansatz Vererbung

```
class KVersalien : public KString { ... };
```



Grundansatz Vererbung

```
class KVersalien : public KString {
public:
    KVersalien(char *s); // Eigener Konstruktor
    bool SetValue(char *s); // Überschriebene Methode
    void Print(); // Überschriebene Methode
};
```

KVersalien möchte von KString erben:

- Attribute mValue und mLength → private! ❌
- Methoden GetValue und Length → public! ✅
- Destruktor → public! ✅

**Verfeinerung des Grundansatzes**

Zwei Arten des Verbergens:

1. Geheimnis (auch) vor Kindern  
 Klasse möchte Attribute und Methoden exklusiv für sich behalten und nicht **beim Vererben** weitergeben  
 ⇒ Wahl der Zugriffsspezifikation **private**
2. „Familiengeheimnisse“  
 Attribute und Methoden werden nur den Erben (und deren Erben usw.) bekannt gemacht, nicht aber Außenstehenden  
 ⇒ Wahl der Zugriffsspezifikation **protected**

**Klasse KString: 2. Versuch**

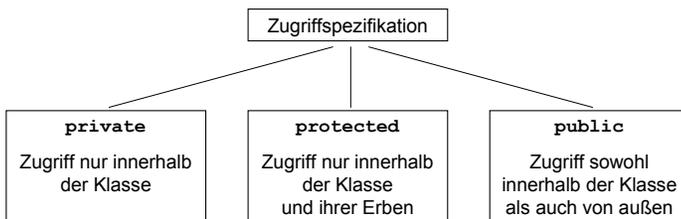
```
class KString {
protected:
    char* mValue;
    int mLength;
public:
    KString(char *s);
    bool SetValue(char *s);
    char *GetValue();
    int Length();
    void Print();
    ~KString();
};
```

einzigste Veränderung

mValue und mLength sind allen Unterklassen von KString bekannt!  
 Objekte anderer Klassen können nicht darauf zugreifen!

**Erste Zusammenfassung**

1. Alle als **public** oder **protected** zugreifbaren Komponenten sind für Erben sichtbar.
2. Die als **private** charakterisierten Komponenten sind in ihrer Sichtbarkeit auf die Klasse selbst beschränkt.



**Sprachliche Regelung:**

Der Vorgang des Erzeugens einer Unterklasse aus einer Oberklasse durch Vererbung nennen wir ableiten.

Hier: Klasse **KVersalien** wird von der Klasse **KString** abgeleitet.

sorgt für die soeben zusammengefassten Zugriffsregeln beim Vererben

```
class KVersalien : public KString {
public:
    KVersalien(char *s); // Eigener Konstruktor
    bool SetValue(char *s); // Überschriebene Methode
    void Print(); // Überschriebene Methode
};
```

Man sagt auch: **public**-Ableitung (zur Unterscheidung ...)

Weitere Formen der Ableitung:

- **public**-Ableitung
 

Oberklasse: public	→ Unterklasse: public	}	weiteres Ableiten ermöglichen: <u>der Normalfall</u>
Oberklasse: protected	→ Unterklasse: protected		
Oberklasse: private	→ Unterklasse: nicht verfügbar		
  
- **protected**-Ableitung
 

Oberklasse: public	→ Unterklasse: <b>protected</b>	}	<u>Spezialfall</u>
Oberklasse: protected	→ Unterklasse: protected		
Oberklasse: private	→ Unterklasse: nicht verfügbar		
  
- **private**-Ableitung
 

Oberklasse: public	→ Unterklasse: <b>private</b>	}	weiteres Ableiten unterbinden: <u>selten</u>
Oberklasse: protected	→ Unterklasse: <b>private</b>		
Oberklasse: private	→ Unterklasse: nicht verfügbar		

Implementierung der Klasse KString

```
#include <iostream>
#include <cstring>
#include "KString.h"

using namespace std;

KString::KString(char *s) {
    mLength = strlen(s); // Länge ohne terminale '\0'
    mValue = new char[mLength+1]; // +1 für '\0'-Zeichen!
    strcpy(mValue, s); // kopiert auch terminale '\0'
}

KString::~KString() {
    delete[] mValue;
    mLength = -1;
}
```

Fortsetzung auf nächster Folie ...

Fortsetzung ...

```
int KString::Length() {
    return mLength;
}

void KString::Print() {
    cout << mValue << endl;
}

char *KString::GetValue() {
    return mValue;
}

bool KString::SetValue(char *s) {
    int length = strlen(s);
    if (length > mLength) return false;
    strcpy(mValue, s);
    mLength = length;
    return true;
}
```

Implementierung der abgeleiteten Klasse KVersalien

```
#include <iostream>
#include <ctype.h>
#include "KVersalien.h"

using namespace std;

KVersalien::KVersalien(char *s) : KString(s) {
    for (int i = 0; i < mLength; i++)
        if (islower(mValue[i]))
            mValue[i] = toupper(mValue[i]);
}
```

Zuerst wird der Konstruktor der Oberklasse KString aufgerufen

Konstruktor der Klasse KVersalien

Ablauf:

1. Zuerst wird Konstruktor von KString aufgerufen, d.h. nach Speicherallokation wird Zeichenkette nach mValue kopiert und mLength wird gesetzt.
2. Danach wird Code im Konstruktor von KVersalien ausgeführt.

Implementierung der abgeleiteten Klasse KVersalien (Fortsetzung)

```
void KVersalien::Print() {
    cout << "KVersalien::Print -> " << endl;
    KString::Print();
}
```

expliziter Aufruf der Methode der Oberklasse

```
bool KVersalien::SetValue(char *s) {
    if (!KString::SetValue(s)) return false;
    for (int i = 0; i < mLength; i++)
        if (islower(mValue[i]))
            mValue[i] = toupper(mValue[i]);
    return true;
}
```

Zeichenkette mit Elternmethode kopieren, falls genug Platz. Dann Versalien erzeugen.

Methoden Length(), GetValue() und der Destruktor werden von der Eltern- / Oberklasse geerbt!

⇒ Implementierung fertig!

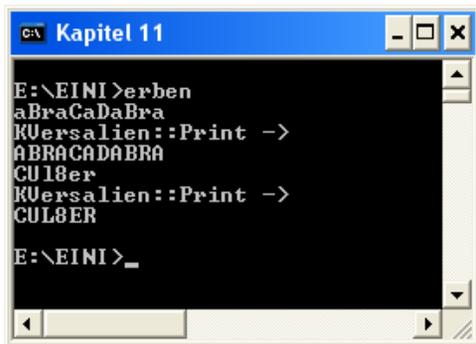
Testumgebung

```
#include <iostream>
#include "KString.h"
#include "KVersalien.h"

using namespace std;

int main() {
    KString *s = new KString("aBraCaDaBra");
    s->Print();
    KVersalien *v = new KVersalien(s->GetValue());
    v->Print();
    s->SetValue("CUl8er");
    s->Print();
    v->SetValue(s->GetValue());
    v->Print();
    delete s;
    delete v;
}
```

Ausgabe:



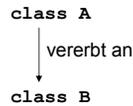
Sprachregelungen:

- Oberklassen werden Elternklassen, manchmal auch Vaterklassen genannt.
- Unterklassen sind von Elternklassen abgeleitete Klassen.
- Abgeleitete Klassen werden manchmal auch Tochterklassen genannt.
- Die Methoden aus Elternklassen können in den abgeleiteten Klassen überschrieben oder redefiniert werden.

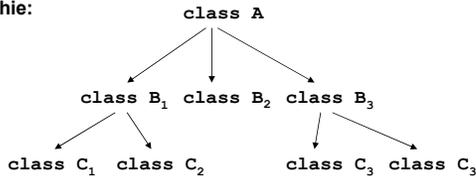
Zweite Zusammenfassung

1. Häufigste Form der Ableitung ist die **public**-Ableitung: `class B:public A { }`
2. Methoden der Elternklassen können benutzt oder überschrieben werden, sofern sie in der Elternklasse **public** bzw. **protected** sind.
3. Überschriebene Methoden der Elternklasse können explizit durch Angabe der Elternklasse aufgerufen werden (Bsp: `KString::SetValue`).

Vererbung bisher:



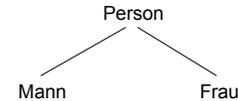
Klassenhierarchie:



hier: einfache Vererbung (nur eine Oberklasse)

Beispiel:

Einfache Klassenhierarchie

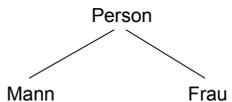


Klasse **Person** enthält alle Attribute und Methoden, die geschlechtsunspezifisch sind.

```

class Person {
private:
    KString *Vorname;
    Frau *Mutter;
    Mann *Vater;
public:
    Person(char *vorname);
    Person(KString *vorname);
    char *Name();
    void SetzeVater(Mann *m);
    void SetzeMutter(Frau *f);
    void Druck();
    ~Person();
};
    
```

Beispiel Klassenhierarchie:



Die abgeleiteten Klassen **Mann** und **Frau** enthalten alle Attribute und Methoden, die geschlechtsspezifisch sind.

```

class Mann : public Person {
private:
    Frau *Ehefrau;
public:
    Mann(char *vn);
    Mann(Person *p);
    void NimmZurFrau(Frau *f);
    Frau *EhemannVon();
};
    
```

```

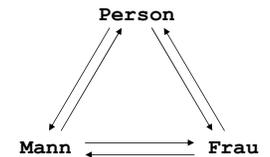
class Frau : public Person {
private:
    Mann *Ehemann;
public:
    Frau(char *vn);
    Frau(Person *p);
    void NimmZumMann(Mann *m);
    Mann *EhefrauVon();
};
    
```

Problem: Zirkularität

Für Klasse **Mann** müssen die Klassen **Person** und **Frau** bekannt sein!

Für Klasse **Frau** müssen die Klassen **Person** und **Mann** bekannt sein!

Für Klasse **Person** müssen die Klassen **Mann** und **Frau** bekannt sein!



A → B bedeutet:  
A wird von B benötigt

Lösung: Vorwärtsdeklaration (wie bei Funktionen)

- bei Funktionen: z.B. `void Funktionsname(int x);`
- bei Klassen: z.B. `class Klassenname;`

hier:

```
class Mann;
class Frau;

class Person { ... };
class Frau: public Person { ... };
class Mann: public Person { ... };
```

Zwei Konstruktoren:

```
Person::Person(KString *vn) : Vater(0), Mutter(0) {
    Vorname = new KString(vn->GetValue());
}

Person::Person(char *vn) : Vater(0), Mutter(0) {
    Vorname = new KString(vn);
}
```

Destruktor notwendig wegen Allokation von dynamischem Speicher

```
Person::~Person() {
    delete Vorname;
}
```

```
char *Person::Name() {
    return Vorname->GetValue();
}
```

Vorname ist private!  
Name() ist public!



Von `Person` abgeleitete Klassen dürfen `Name()` nicht überschreiben,  
sonst ist der Zugriff auf `Vorname` auch für sie verwehrt!

```
void Person::SetzeMutter(Frau *f) {
    Mutter = f;
}
void Person::SetzeVater(Mann *m) {
    Vater = m;
}
```

```
void Person::Druck(char *s) {
    cout << s << "Vorname: " << Vorname->GetValue() << endl;
    if (Mutter != 0) {
        cout << s << "Mutter: ";
        Mutter->Druck("");
    }
    if (Vater != 0) {
        cout << s << "Vater : ";
        Vater->Druck("");
    }
}
```

von Person  
geerbte Methode

## Kapitel 11: Vererbung

```
Mann::Mann(Person *p) : Person(p->Name()), Ehefrau(0) { }
Mann::Mann(char *vn) : Person(vn), Ehefrau(0) { }
```

```
void Mann::NimmZurFrau(Frau *f) {
    Ehefrau = f;
}
Frau *Mann::EhemannVon() {
    return Ehefrau;
}
```

Implementierung der  
Klasse Mann

```
Frau::Frau(Person *p) : Person(p->Name()), Ehemann(0) { }
Frau::Frau(char *vn) : Person(vn), Ehemann(0) { }
```

```
void Frau::NimmZumMann(Mann *m) {
    Ehemann = m;
}
Mann *Frau::EhefrauVon() {
    return Ehemann;
}
```

Implementierung der  
Klasse Frau

## Kapitel 11: Vererbung

### Hilfsroutinen

```
void Verheirate(Mann *m, Frau *f) {
    if (m != 0 && f != 0) {
        m->NimmZurFrau(f);
        f->NimmZumMann(m);
    }
}
```

### Bemerkung:

„Schlampige“ Programmierung,  
weil man vorher noch testen  
müsste, ob beide Personen  
ledig sind!

```
void Scheide(Mann *m, Frau *f) {
    if (m->EhemannVon() == f) {
        m->NimmZurFrau(0);
        f->NimmZumMann(0);
    }
}
```

### Bemerkung:

„Schlampige“ Programmierung,  
weil ... ja, warum?

## Kapitel 11: Vererbung

### Testprogramm

```
int main() {
    Mann *Anton = new Mann("Anton");
    Frau *Bertha = new Frau("Bertha");

    Mann *Carl = new Mann("Carl");
    Carl->SetzeMutter(Bertha); Carl->SetzeVater(Anton);

    Frau *Doris = new Frau("Doris");
    Doris->SetzeMutter(Bertha); Doris->SetzeVater(Anton);

    Anton->Druck("A: "); Bertha->Druck("B: ");
    Carl->Druck("\tC:"); Doris->Druck("\tD:");

    Verheirate(Anton, Bertha);
    Bertha->EhefrauVon()->Druck("B ist Frau von: ");
    Anton->EhemannVon()->Druck("A ist Mann von: ");

    delete Doris; delete Carl; delete Bertha; delete Anton;
}
```

## Kapitel 11: Vererbung

### Ausgabe:

```
A: Vorname: Anton
B: Vorname: Bertha
   C:Vorname: Carl
   C:Mutter: Vorname: Bertha
   C:Vater : Vorname: Anton
   D:Vorname: Doris
   D:Mutter: Vorname: Bertha
   D:Vater : Vorname: Anton
B ist Frau von: Vorname: Anton
A ist Mann von: Vorname: Bertha
```



## Abstrakte Klassen

*„ ... ein paar Bemerkungen vorab ...“*

hier:

- Klasse **Person** dient nur als „Behälter“ für Gemeinsamkeiten der abgeleiteten Klassen **Mann** und **Frau**
- Es sollen keine eigenständigen Objekte dieser Klassen instantiiert werden!  
Hier wäre es jedoch möglich: `Person p("Fred");`

→ Man kann erzwingen, dass abstrakte Klassen nicht instantiiert werden können!

→ nächstes Kapitel ... (u.a.)