

Einführung in die Programmierung

Wintersemester 2020/21

Kapitel 10: Vererbung

M.Sc. Roman Kalkreuth
Lehrstuhl für Algorithm Engineering (LS11)
Fakultät für Informatik

Modellierung von Objekten mit geringen Unterschieden

Bisherige Mittel zum Modellieren von „ähnlichen“ Objekten:

- Sei Klasse A bereits definiert (Beispiel: Sparkonten).
- 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!

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**

Alternative: Vererbung bzw. Erben

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 const *s);
    bool setValue(char const *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 { ... };
```

Name der
Unterklasse

Unterklasse **KVersalien** **erbt**
von der Oberklasse **KString**:
Attribute und Methoden von
KString sind in **KVersalien**
(bedingt) verfügbar.

⇓

öffentliche Attribute / Methoden
private Attribute / Methoden

public besagt **an dieser Stelle**:
Übernahme der Zugriffsspezifikationen von
der Oberklasse (hier: von **KString**)

Grundansatz Vererbung

```
class KVersalien : public KString {
public:
    KVersalien(char const *s); // eigener Konstruktor
    bool setValue(char const *s); // überschriebene Methode
    void print(); // überschriebene Methode
};
```

KVersalien möchte von **KString** erben:

- Attribute **mValue** und **mLength** → **private**
- Methoden **GetValue** und **Length** → **public**
- Destruktor → **public**

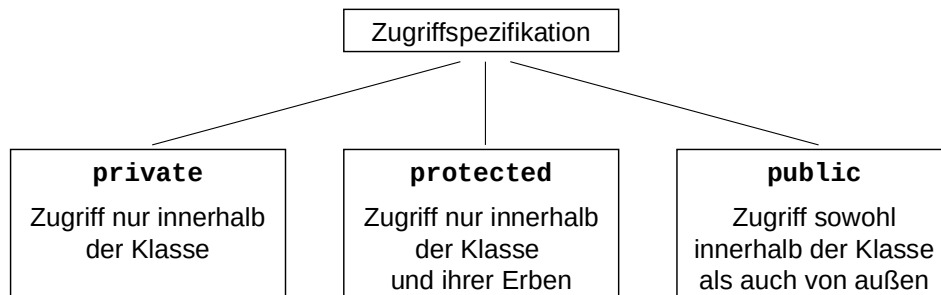
Verfeinerung des Grundansatzes

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.) **bekanntgemacht**, nicht aber Außenstehenden
⇒ Wahl der Zugriffsspezifikation **protected**

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.



Klasse KString: 2. Versuch

```

class KString {
protected:
    char *mValue;
    int mLength;
public:
    KString(char const *s);
    bool setValue(char const *s);
    char *getValue();
    int length();
    void print();
    ~KString();
};
  
```

einzig
Veränderung



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

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.

Dieses **public** sorgt für die soeben zusammengefassten Zugriffsregeln beim Vererben:

```

class KVersalien : public KString {
public:
    KVersalien(char const *s); // eigener Konstruktor
    bool setValue(char const *s); // überschriebene Methode
    void print(); // überschriebene Methode
};
  
```

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

Weitere Formen der Ableitung:

• **public-Ableitung**

Oberklasse: public → Unterklasse: public
 Oberklasse: protected → Unterklasse: protected
 Oberklasse: private → Unterklasse: nicht verfügbar

} weiteres Ableiten
 ermöglichen:
der Normalfall

• **protected-Ableitung**

Oberklasse: public → Unterklasse: **protected**
 Oberklasse: protected → Unterklasse: protected
 Oberklasse: private → Unterklasse: nicht verfügbar

} **Spezialfall**

• **private-Ableitung**

Oberklasse: public → Unterklasse: **private**
 Oberklasse: protected → Unterklasse: **private**
 Oberklasse: private → Unterklasse: nicht verfügbar

} weiteres Ableiten
 unterbinden:
selten

Fortsetzung ...

```
int KString::length() {
    return mLength;
}
void KString::print() {
    cout << mValue << endl;
}
char *KString::getValue() {
    return mValue;
}
bool KString::setValue(char const *s) {
    int length = strlen(s);
    if (length > mLength) return false;
    strcpy(mValue, s);
    mLength = length;
    return true;
}
```

Implementierung der Klasse KString

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

using namespace std;

KString::KString(char const *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;
}
```

Fortsetzung auf nächster Folie ...

Implementierung der abgeleiteten Klasse KVersalien

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

using namespace std;

KVersalien::KVersalien(char const *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();
}

bool KVersalien::setValue(char const *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;
}

```

expliziter Aufruf
der Methode der
Oberklasse

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

Methoden `length()`, `getValue()` und der Destruktor werden von der Eltern- / Oberklasse geerbt.

⇒ Implementierung fertig!

Ausgabe:

```

C:\> Kapitel 11
E:\EINI>erben
aBraCaDaBra
KVersalien::Print ->
ABRACADABRA
CUl8er
KVersalien::Print ->
CUL8ER
E:\EINI>_

```

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;
}

```

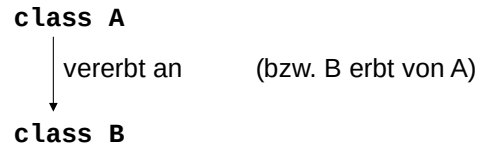
Sprachregelungen:

- 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.

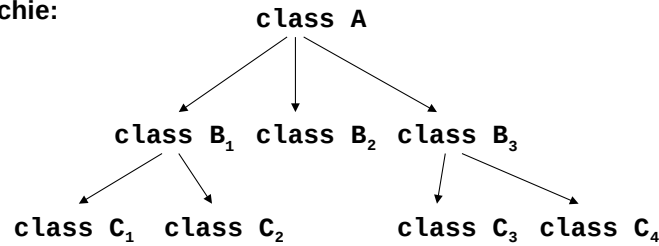
Zweite Zusammenfassung

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

Vererbung bisher:

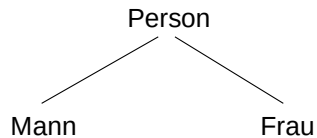


Klassenhierarchie:



hier: einfaches Erben (nur **eine** Oberklasse)

Beispiel Klassenhierarchie:



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

```

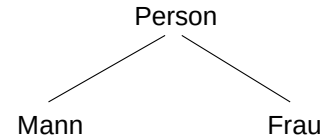
class Mann : public Person {
private:
    Frau *Ehefrau;
public:
    Mann(char const *vn);
    Mann(Person *p);
    void NimmZurFrau(Frau *f);
    Frau *EhemannVon();
};
    
```

```

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

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 const *vorname);
    Person(KString *vorname);

    char *Name();
    void SetzeVater(Mann *m);
    void SetzeMutter(Frau *f);
    void Druck(char const *s);
    ~Person();
};
    
```

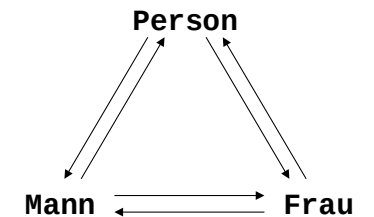
Annahme: Rechtslage in Deutschland vor Oktober 2017 ⇒
 • keine „Ehe für alle“
 • kein 3. Geschlecht

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 { ... };
```

```
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 für sie verwehrt!

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

Zwei Konstruktoren:

```
Person::Person(KString *vn) : Vater(nullptr), Mutter(nullptr) {
    Vorname = new KString(vn->getValue());
}
Person::Person(char const *vn) : Vater(nullptr), Mutter(nullptr)
{
    Vorname = new KString(vn);
}
```

Destruktor notwendig wegen Allokation von dynamischem Speicher

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

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

von **Person**
geerbte Methode

```

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

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

```

Implementierung der
Klasse Mann

```

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

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

```

Implementierung der
Klasse Frau

Hilfsroutinen

```

void Verheirate(Mann *m, Frau *f) {
    if (m != nullptr && f != nullptr){           // Existenz?
        if (m->EhemannVon() != nullptr) return; // ledig?
        if (f->EhefrauVon() != nullptr) return; // ledig?
        m->NimmZurFrau(f);
        f->NimmZumMann(m);
    }
}

void Scheide(Mann *m, Frau *f) {
    if (m != nullptr && f != nullptr){           // Existenz?
        if (m->EhemannVon() == f) {              // verheiratet?
            m->NimmZurFrau(nullptr);
            f->NimmZumMann(nullptr);
        }
    }
}

```

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;
}

```


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 instanziiert werden!
Hier wäre es jedoch möglich: **Person p("Fred")**;

→ Man kann auf Sprachebene **verbieten**, dass abstrakte Klassen instanziiert werden können.

→ nächstes Kapitel ... (u.a.)