Kapitel 14 - Vererbung

Peter Ulbrich

AG Systemsoftware

Veranstaltungswebseite

Einleitung

- Bisher: Wichtigste Eigenschaften von OOP
  - Sichtbarkeit/Kapselung (→ `private` / `public`)
  - Darauf aufbauend: Abstraktion (Verstecken der Implementierungsdetails)
- Für reale Anwendungsfälle fehlt aber noch etwas…
  - Vererbung (heute)
  - Polymorphismus

Initialisierung von Member-Objekten

- Im letzten Kapitel ausgeklammert:

  Wie werden Objekte, die Member sind, korrekt initialisiert?

- Objekte müssen als Member natürlich auch konsistent initialisiert werden
- Zwei Möglichkeiten
  - Automatische Default-Initialisierung (sofern möglich)
  - Expliziter Aufruf des Konstruktors in der Initialisierungsliste

```
1  class Foo {
2      int i;
3  public:
4      Foo() : i(0) {}
5      Foo(int num) : i(num) {}
6  };
7
8  class Bar {
9      Foo f;
10     double j;
11 public:
12     Bar(double d) : j(d) {} // f() automatisch
13     Bar(int num, double d) : f(num), j(d) {}
14 };
```

(Die Destruktoren von Membern werden übrigens auch automatisch bei Aufruf des Destruktors aufgerufen)

Beispiel - Initialisierung von Member-Objekten

cpp Run ▶

```cpp
#include <iostream>

using std::cout, std::endl;

class Foo {
    int i;
public:
    Foo() : i(0) { cout << "                endl; }
    Foo(int num) : i(num) {                 endl; }
    ~Foo() { cout << "Foo de
    int value() { return i; }
};

class Bar {
    Foo f;
    double j;
public:
    Bar(double d) : j(d) { cout << "Bar default constructor called" << endl; }
```

Und nun zur Vererbung 🥳

Einführungsbeispiel zur Vererbung

- Beispiel: Tiere
- Mit dem aktuellen Wissen wäre das bereits prinzipiell umsetzbar

```
1  class Dog {
2      std::string name;
3      std::string laut;
4  };
```

```
1  class Cat {
2      std::string name;
3      std::string laut;
4  };
```

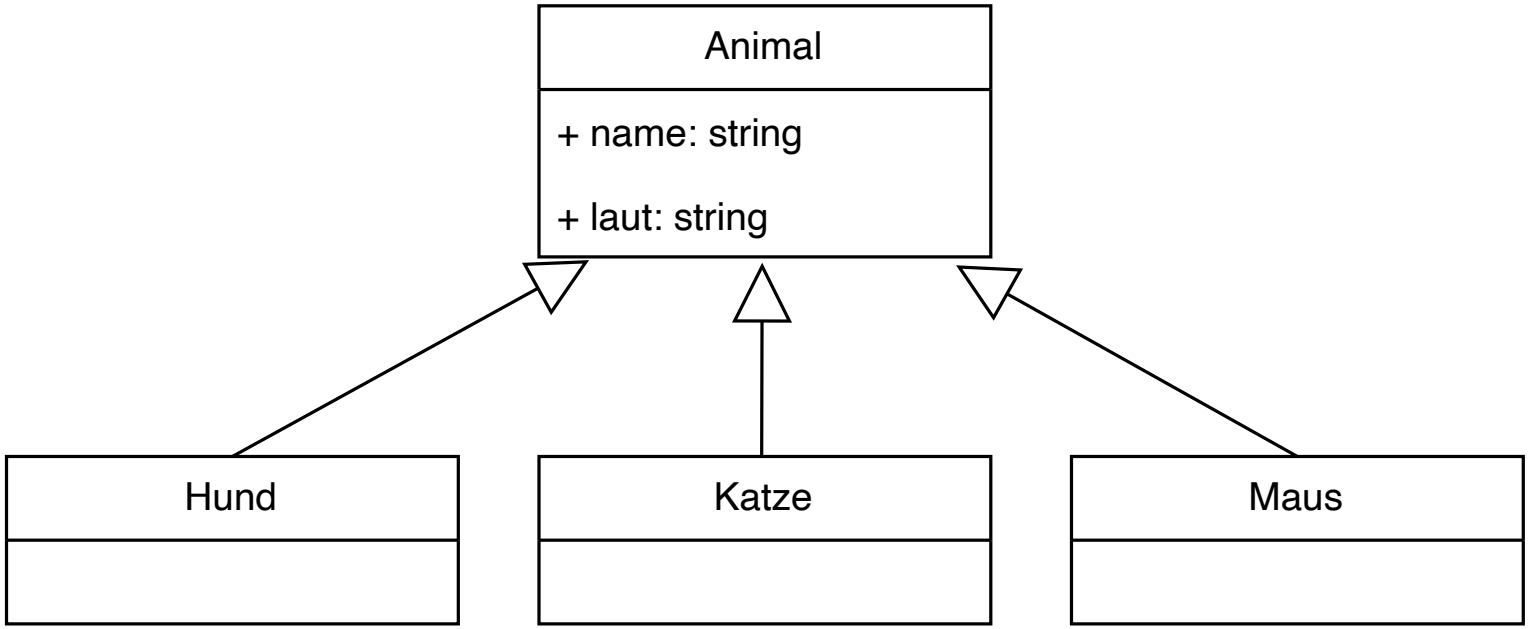
```
1  class Mouse {
2      std::string name;
3      std::string laut;
4  };
```

- Probleme
  - **Sehr** aufwendig, starke Duplizierung von Code
  - Fehleranfällig: Member vergessen beim Kopieren, falsche Initialisierung
  - ...und sehr stupide Arbeit

Vererbung

- Bessere Lösung: Vererbung
- Tiere haben gemeinsame Eigenschaften
  - Zusammenfassen in einer Oberklasse (häufig auch Elternklasse bzw. Basisklasse)
  - Oberklasse `Animal`
- Unterklassen `Dog` und `Cat` erben von `Animal`
  - Methoden und Member der Oberklasse stehen zur Verfügung
  - Hinzufügen weiterer Member in den Unterklassen möglich
  - Ebenso: Überschreiben von existierenden Methoden
- Wichtige Eigenschaft: Jede Instanz einer Unterklasse ist gleichzeitig Instanz der Oberklasse
  - Beispiel: „Jeder Hund ist gleichzeitig ein Tier“

Vererbung in UML



Beispiel für eine Vererbung in UML

- Besondere Kennzeichnung: Weiße Pfeilspitze, zeigt auf Basisklasse
- Wird in UML auch **Generalisierung** genannt

Grundlagen der Vererbung

- Vererbung findet in C++ bei der Klassendeklaration statt
- Auflistung Basisklassen hinter Klassennamen
- Initialiserung der Basisklasse im Konstruktor
- Parameter werden entsprechend weitergeleitet

```
1  using std::string;
2  using std::cout;
3  class Animal {
4  private:
5      string name;
6      string laut;
7  public:
8      Animal() = delete;
9      Animal(string n, string l):
10         name(n), laut(l) {}
11 };
12
13 class Dog : public Animal {
14 public:
15     Dog(string name, string laut):
16         Animal(name, laut) {}
17     void bellen() {
18         cout << this->laut;
19     }
```

Initialisierungsreihenfolge

- Es gilt: Ober- vor Unterklasse
- Erst Konstruktoren der Oberklasse(n) aufrufen
- Danach folgt Initialisierung der Kindklasse
- Randbemerkung: Der Vorgang des Erbens wird häufig auch Ableitung genannt

  → „Dog wird von Animal abgeleitet“
- Auch hier gilt: Die Begriffe **Erben** und **Ableiten** werden häufig synonym verwendet

```
1  using std::string;
2  using std::cout;
3  class Animal {
4  private:
5      string name;
6      string laut;
7  public:
8      Animal() = delete;
9      Animal(string n, string l):
10         name(n), laut(l) {}
11 };
12
13 class Dog : public Animal {
14 public:
15     Dog(string name, string laut):
16         Animal(name, laut) {}
17     void bellen() {
18         cout << this->laut;
19     }
```

Sichtbarkeit bei Vererbung

- Ist jetzt alles geklärt? Leider nein 😕
- `private` bedeutet wirklich **privat**!

  → **nur** die Klasse kann zugreifen
- Die Methode `bellen()` erzeugt Compiler-Fehler

cpp Run ▶

```cpp
#include <string>
#include <iostream>
using std::cout;
using std::string;

class Animal {
private:
    string name;
    string laut;
public:
    Animal() = delete;
    Animal(string n, string l):
        name(n), laut(l) {}
};

class Dog : public Animal {
private:
    bool lieb = true;
```

Selektive Vererbung

- Problem: Nur ein Teil der Basisklasse vererben
- Lösung: Schlüsselwort `protected` 🔒
  - Verwendung wie `private` und `public`
  - Member mit `protected` sind für Kindklassen sichtbar

```
1  class Animal {
2  private:   // Unsichtbar in Kind
3      bool secret = true;
4  protected: // Ab hier: sichtbar
5      std::string name;
6      std::string laut;
7  public:
8      Animal() = delete;
9      Animal(string n, string l):
10         name(n), laut(l) {}
11 };
```

- Der Bereich `protected` ist so etwas wie ein Familiengeheimnis

  → Die Kinder kennen es, aber außerhalb der Familie niemand

- `private` entspricht hingegen einem persönlichen Geheimnis

  → Ist nur der Klasse selbst bekannt

Vererbung

Jetzt funktioniert alles ✅

cpp Run ▶

```cpp
#include <iostream>

using namespace std;

class Animal {
private:   // Unsichtbar in Kind
    bool secret = true;
protected: // Ab hier: sichtbar
    std::string name;
    std::string laut;
public:
    Animal() = delete;
    Animal(string n, string l):
        name(n), laut(l) {}
};

class Dog : public Animal {
public:
```

Selektive Vererbung

- Eigenart von C++: Basisklassen automatisch als `private` deklariert
  - Folge: **Alle** Member der Basisklasse werden automatisch `private`
  - Kein Zugriff durch Kindklasse mehr erlaubt
  - Analog für `protected` (verbietet `public`)
- Umgehung durch **explizite** Deklaration als `public`
- Achtung: **Sehr** beliebter Flüchtigkeitsfehler

```
1  class Animal {
2  private:   // Unsichtbar in Kind
3      int secret;
4  protected: // Ab hier: sichtbar
5      string name;
6      string laut;
7  public:
8      string get_name() {
9          return name;
10     }
11 };
```

```
1  // Alles aus Animal wird private
2  class Dog : Animal {};
3
4  // Ändert public zu protected
5  class Dog : protected Animal {};
6
7  // Keine Veränderungen
8  class Dog : public Animal {};
```

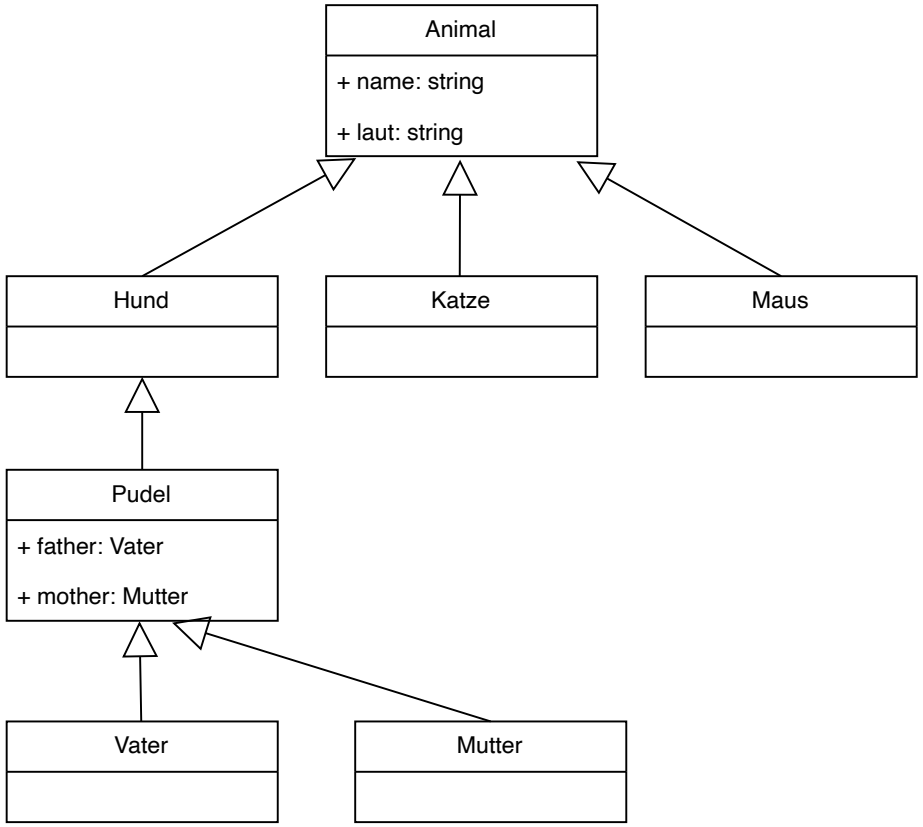
Mehrfachvererbung

- Ableitung von mehreren Basisklassen
- Prinzipiell genau gleich wie bisher
- Einzige Besonderheit: Reihenfolge der Konstruktoraufrufe der Basisklassen hängt von der Auflistung während der Deklaration ab (von links nach rechts)

```
1  class Pet {
2  protected:
3      bool is_pet; // Haustier-Check
4  };
5
6  class Animal {
7  private:   // Unsichtbar in Kind
8      int secret;
9  protected: // Ab hier: sichtbar
10     string name;
11     string laut;
12 public:
13     string get_name() {
14         return name;
15     }
16 };
17
18 // Erst Init. von Pet, dann Animal
19 class Dog : public Pet, public Animal {};
```

Mehrfachvererbung

- Klassen können beliebig oft abgeleitet werden
- Das wird auch so gemacht!
- Darstellung aller Vererbungen über sogenannte Klassenhierarchie (typischerweise UML-Grafik)



Mehrfachvererbung

- Problem: Praktische Umsetzung in C++ dieser Klassenhierarchie nicht möglich
- Bei Deklaration von `Poodle` sind die Klassen `Mother` und `Father` noch gar nicht deklariert
- Hier hilft Forward Declaration
- Aus *C* kennen wir etwas Verwandtes: `extern`

```
1  // Forward Declaration
2  class Father;
3  class Mother;
4
5  class Poodle {
6      Father vater;
7      Mother mutter;
8  }
```

Großes Beispiel

- Praktisch umgesetzt sieht diese Hierarchie folgendermaßen aus

cpp Run ▶

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

using std::string;
using std::cout;

class Animal {
protected: // Ab hier: sichtbar
    std::string name;
    std::string laut;
public:
    Animal() = delete;
    Animal(string n, string l):
        name(n), laut(l) {}
};

class Dog : public Animal {
public:
```

Umgang mit verschiedenen Kindklassen

- Szenario: Abspeichern von Objekten verschiedener Kindklassen
  - Zum Beispiel: Array aller Katzen und Hunde
  - Mit klassischen Arrays geht das nicht → zwei verschiedene Typen
- Lösung: Beide sind Objekte vom Typ `Animal` → Array vom Typ `Animal`

```
1  Animal anim_array[10]; // Default-Init.
2
3  anim_array[0] = new Dog("Paul", "Wuff");
```

- Wichtig: Methoden von Kindklassen sind nicht unmittelbar zugreifbar
  - Diese Eigenschaft wird als *Slicing* bezeichnet → mehr im nächsten Kapitel
  - Umweg über *Type Casting* möglich
- Das Gleiche gilt natürlich auch für Zeiger!

```
1  Animal* a = new Dog("Paul", "Wuff");
2
3  unique_ptr<Animal> = make_unique<Dog>("Paul", "Wuff");
```

Ausblick: Abstrakte Klassen

- Die bisherigen Beispielklassen sind alle noch instanziierbar gewesen
- Jetzt: Basisklassen als Schablonen
  - Können nicht mehr direkt instanziiert werden
  - Stattdessen **müssen** Kindklassen abgeleitet werden

```
1  class Form {
2      virtual void get_area() = 0;
3  };
4
5  class Rechteck : public Form {/*...*/};
6  class Quadrat  : public Form {/*...*/};
7  class Kreis    : public Form {/*...*/};
8  // ...
```

- Das Schlüsselwort `virtual` hilft hierbei → mehr im nächsten Kapitel

Zusammenfassung

- Vererbung erlaubt akkurate Modellierung der Realität
- Initialisierungsreihenfolge der Konstruktoren
- Sichtbarkeitsmodifikatoren entscheiden über Weitergabe von Informationen
  - `public` → für alle zugreifbar
  - `protected` → nur für die Klasse und Unterklassen
  - `private` → nur für die Klasse selbst

Kapitel 14 - Vererbung