Vorkurs Informatik: Erste Schritte der Programmierung mit C++

Tag 3: Objektorientiere Programmierung

8. Oktober 2014

STARKER START ins Studium

A. Nägel, A. Vogel, G. Wittum

Agenda

1. Einführung
2. Objektorientierte Programmierung
3. Objekte und Klassen mit C++
   - Objekte mit C++
   - Modularisierung
   - Konstruktoren und Desktruktoren
   - Vererbung
4. Ein Beispiel: Konten einer Bank

Objektorientierte Programmierung

Objektorientierte Programmierung: Was ist das?

- Es gibt verschiedene *Programmierparadigmen*, d.h., Stile an ein Problem heranzugehen, es zu modellieren und zu programmieren
- Bisher: **Prozedurale Programmierung**
  - Zerlegung in Variablen, Datenstrukturen und Funktionen
  - Funktionen operieren direkt auf Datenstrukturen
- Nun: **Objektorientierte Programmierung (OOP)**
  - Beschreibung eines Systems anhand kooperierender Objekte und ihres Zusammenspiels
  - Abstraktion der Objekte in Klassen

Objekte

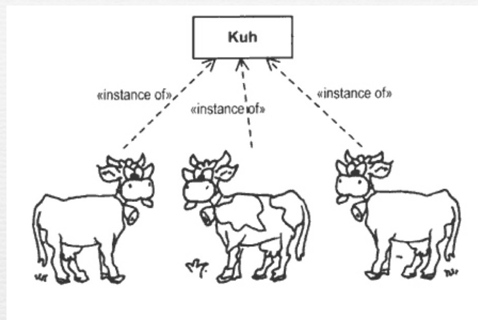
- Gegenstand, auf den sich jemand bezieht, auf den das Denken oder Handeln ausgerichtet ist
- Sind überall und werden als solche wahrgenommen.

Reale Welt	**OOP**
• Zustand	• Attributwerte
• Verhalten	• Methoden

Klassen und Objekte

- Klasse
  - Bauplan für Objekt
  - “Idee” der Objekte
  - Definition aller Attribute und Methoden Klasse allein macht noch nichts
- Objekt
  - ist konkretes Element (Instanz) dieser Klasse

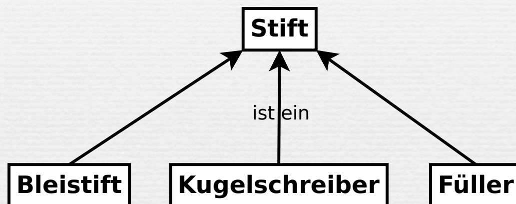
Klassen und Objekte



Klasse Kuh	**Objekt Kuh**
• Name	• Elsa Euter
• Geburtsdatum	• 27. März 2012
• Milchleistung	• 30 l/Tag

Klassenhierarchie

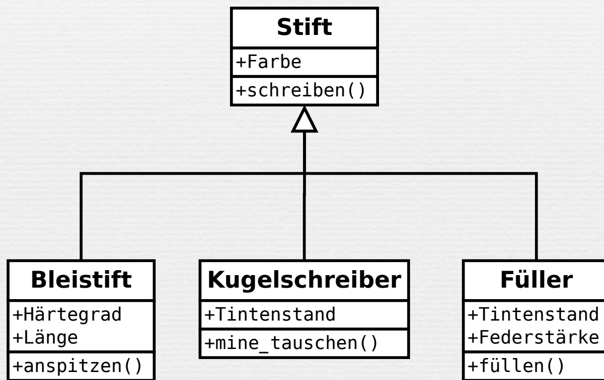
- Verschiedene Arten von Objekten haben oft Gemeinsamkeiten
- Man modelliert dies über eine "ist ein"-Beziehung
- Beispiel:



- Superklasse ≡ Elternklasse ≡ Oberklasse ≡ Basisklasse
- Subklasse ≡ Kindklasse ≡ Unterklasse ≡ abgeleitete Klasse

Vererbung

- Kindklassen erben alle Attribute und Methoden von Elternklassen
- haben zusätzlich eigene Attribute und Methoden
- können Attribute und Methoden der Elternklasse überschreiben



Vorteile

- **Abstraktion:** Betrachtung der Objekte und ihrer Eigenschaften und Fähigkeiten, ohne Festlegung auf Implementierung
- **Datenkapselung:** Objekt interagiert nur über vordefinierte Methoden. Implementierung kann verändert werden, ohne dass andere Teile des Programms geändert werden müssen
- **Vererbung:** klarere Struktur und weniger Redundanz
- **Wiederverwendbarkeit:** Programme können einfacher erweitert und modifiziert werden. Klassen können auch in anderen Programmen verwendet werden.

Nachteile

- **Formulierung:** natürliche Sprache hat keine feste Bindung von Substantiv (Objekt) und Verb (Methode).
- **Klassenhierarchie:** ist in der realen Welt nicht immer so klar. (Beispiel: Jeder Kreis ist eine Ellipse. Soll man das auch so programmieren?)
- **Transparenz:** Kontrollfluss nicht im Quelltext
- **Laufzeitverhalten :** OOP-Anwendungen benötigen häufig längere Laufzeit (mehr Energie)

Objekte und Klassen mit C++

Strukturen

Ein **Struct** definiert einen neuen Datentyp welcher Komponenten unterschiedlichen Typs vereint.

```
struct <Name> {
      <Typ1> Name1;
      <Typ2> Name2;
      ...
};
```

```
struct Complex {
  float re; // Realteil
  double im; // Imaginaerteil
  bool isComplex;
};
int main(){
  Complex x;
  x.re = 2.2; x.im = 1.7; x.isComplex = true;
}
```

Klassen

Ein **Struct** eine besondere (vollständig öffentliche) Klasse.
Daten und zugehörige Funktionen werden in **Klassen** gruppiert.

```
class <Classname> {
  int data1; // Daten
  double data2;
  void doSomethingWithData(); // Funktionen
};
```

Daten werden **Attribute** genannt, Funktionen werden **Methoden** genannt.

Public/Private

Attribute und Methoden können öffentlich sicherbar sein (`public`) oder private (`private`). Private Member/Methoden können nur von der Klasse selbst verwendet werden.

```
class Complex{
  public:
    double real() {return re;}
    double imaginary() {return im;}

  private:
    double re, im; // kÃ¶nnen nur von der Klasse verwendet werden
};
```

Der öffentliche Teil der Klasse ist seine Schnittstelle. Die privaten Attribute werden zur Implementierung genutzt.

const Methoden

Methoden können als const deklariert werden. Dies bedeutet, dass diese Methode die Daten der Klasse nur lesen darf, aber durch die Methode die Werte nicht verändert werden können. Der Compiler kontrolliert dies.

```
class Complex{
  public:
    double real() const {return re;}          // ok, re wird kopiert
    const double& real() const {return re;} // ok, nur konstante
    Referenz
    double& real() const {return re;}       // Fehler: re kÃ¶nnte
                                              //            verÃndert
    werden
    void addToReal(double x) const {re += x;} // Fehler: re
    verÃndert

  private:
    double re, im; // kÃ¶nnen nur von der Klasse verwendet werden
};
```

const Korrektheit

Die const Eigenschaften können verwendet werden, um sicheren Code zu schreiben. Der Compiler überprüft dies.

```
class MyDouble{
  public:
    const double& getValue() {return value;}
    void setValue(double x) {value = x;}

  private:
    double value;
};

int main(){
  MyDouble d;
  const MyDouble& r;    // einen konstante (!) Referenz

  d.setValue(10); // ok, d ist nicht konstant
  r.getValue();   // ok, r konstant, aber const-Methode
  r.setValue(12); // Fehler: r konstant, aber Methode nicht konstant
```

Der this Zeiger

Jede Klasse hat automatisch einen Zeiger this. Dieser zeigt auf die Instanz selbst und ist manchmal hilfreich, z.B. um den Quelltext lesbarer zu machen:

```
class MyDouble{
  public:
    const double& getValue() {return this->value;}
    void setValue(double x) {this->value = x;}

  private:
    double value;
};
```

STARKER START ins Studium

Trennung Deklaration und Implementierung

```
class MyDouble{
 public:
  const double& getValue()
     ;
  void setValue(double x);

 private:
  double value;
};
```

mydouble.h

```
#include "mydouble.cpp"

const double& MyDouble::getValue()
     const {
  return value;
}

void MyDouble::setValue(double x) {
  value = x;
}
```

mydouble.cpp

Methoden können außerhalb der Klasse definiert werden – und damit in einer cpp-Datei. Die Methodennamen müssen dabei mit einem `<Classname>::` Prefix geschrieben werden.

Konstruktor

Wird eine Instanz einer Klasse angelegt, dann wird immer eine Methode aufgerufen, die **Konstruktur** genannt wird.

Wird vom Benutzer keine eigene Implementierung vorgenommen, dann wird automatisch ein Standardkonstruktor erzeugt.

```
class MyClass {
  public:
    // Standardkonstruktor
    MyClass()
    {
      // wird beim Anlegen eines Objekts aufgerufen
    }
};
```

Eigene Konstruktoren

Selbstgeschriebene Konstruktoren werden verwendet, wenn beim Anlegen eines Objekts weitere Dinge geschehen soll. Zudem kann man die Attribute initialisieren.

```
class MyClass {
  public:
    // eigener Konstruktor
    MyClass(double x) : m1(x)
    {
      m2 = 5*x;
    }

  private:
    double m1, m2;
};
```

Es werden immer zunächst die Attribute initialisiert, dann wird der [...]block des Konstruktors ausgeführt.

Destruktoren

Wird ein Objekt gelöscht, so wird der **Destruktor** aufgerufen.

```
class MyClass {
  public:
    // Destruktor
    ~MyClass()
    {
       // wird beim LÃ¶schen eines Objekts aufgerufen
    }
};
```

Der Konstruktor dient meist dazu, dynamisch angelegten Speicher freizugeben.

Kopier-Konstruktor

Wird ein Objekt durch die Kopie eines anderen angelegt, so wird der **Copykonstruktor** aufgerufen.

```
class MyClass {
  public:
    // Copykonstruktor
    MyClass(const MyClass& orig)
    {
       // wird beim Anlegen durch Kopie aufgerufen
    }
};
```

Beispiel: `MyClass b; MyClass a(b); MyClass c=b;`

Zuweisungsoperator

Wird ein Objekt einem anderen zugewiesen, so wird der **Zuweisungsoperator** aufgerufen.

```
class MyClass {
  public:
    // Zuweisungsoperator
    MyClass& operator=(const MyClass& orig)
    {
       // wird beim Zuweisung aufgerufen
    }
};
```

Beispiel: `MyClass b; MyClass a; a = b;`

Exkurs: Operatoren einer Klasse

Eine Klasse kann verschiedene Operatoren selber definieren. Darunter:
*, *=, /, /=, +, +=, -, -=, «

```
#include <iostream>
class MyClass {
  public:
    MyClass& operator*=(const MyClass& orig) { /* ... */ }
    MyClass& operator*=(double x) { /* ... */ }
    MyClass& operator*=(int x) { /* ... */ }
    /* ... */
};
```

Beispiel: `MyClass a; a *= 5;`

Vererbung

Man kann eine bestehende Klasse um Funktionalität erweitern. Dazu leitet man eine neue Klasse von der bestehenden ab.

Faustregel: Eine Ableitung repräsentiert eine **"ist-ein"-Beziehung**. Eine **"hat-ein"-Beziehung** besser als Attributvariable realisieren.

Man unterscheidet drei verschiedene Arten der Ableitung:

`public, protected, private`

Die relevanteste Art ist `public`.

public Ableitungen

```
class Base {
  public:    int PublicAttribut;
  protected: int ProtectedAttribut;
  private:   int PrivateAttribut;
};

class Derived : public Base {
  // public    Attribut von Base: sind public    in Derived
  // protected Attribut von Base: sind protected in Derived
  // private   Attribut von Base: kein Zugriff   in Derived
};
```

Implizite Konvertierung von Klassen

Da eine abgeleitete Klasse immer auch die Basisklasse enthält, kann man diese immer dort verwenden, wo die Basisklasse gefordert ist.

```
class Base {};
class Derived : public Base {};

int main(){
  Derived d;
  Base& b = d;   // ok: Derived beinhaltet Base
  Base* pb = &d; // ok: Derived beinhaltet Base
}
```

Dies nennt man **implizite Typkonvertierung**.

Überschriebene Methoden

Eine abgeleitete Klasse kann Methoden der Basisklasse überschreiben. Es darf also Methoden mit gleicher Signatur wie in der Basisklasse geben. Die Methode wird dann ersetzt.

```
class Base {
  int value() {return 5;}
};

class Derived : public Base {
  int value() {return 10;}
};

int main(){
  Derived d;
  std::cout << d.value(); // 10
}
```

Polymorphie

Polymorphismus bedeutet **Vielgestaltigkeit**.

Gemeint ist: Zur *Laufzeit* eines Programms kann eine Referenz oder ein Pointer auf eine Basisklasse auf verschiedene Objekte – und damit auf verschiedene abgeleitete Klassen – verweisen.

Wird diese Zuordnung erst zur Laufzeit vorgenommen nennt man dies **dynamisches** Binden. Geschieht es zur Compilierzeit, so wird es **statisches** Binden genannt.

Virtuelle Methoden

Methoden können als `virtual` deklariert werden. Wird dann diese Methode bei einer Referenz oder einem Pointer auf ein Objekt aufgerufen, so wird zunächst der tatsächlich Typ des Objekts bestimmt und dann die Methode dieses Typs aufgerufen.

```
class Base {
  virtual int value() {return 5;}
};
class Derived : public Base {
  virtual int value() {return 10;}
};

int main(){
  Derived d; Base b;
  Base& r1 = d; std::cout << r1.value(); // 10
  Base& r2 = b; std::cout << r2.value(); // 5
}
```

Rein virtuelle Methoden

Methoden können als rein virtuell deklariert werden. Dies bedeutet, dass die Basisklasse keine Implementierung vornimmt. Von der Basisklasse kann dann keine Instanz erzeugt werden. Abgeleitete Klassen müssen diese Methode dann implementieren. Die Basisklasse nennt man **Abstrakte Klasse**.

```
class Base {
  virtual int value() = 0;  // rein virtuelle Methode
};
class Derived : public Base {
  virtual int value() {return 10;}
};

int main(){
  Base b;         // Fehler: Base ist eine abstrakte Klasse
  Derived d;      // ok: Derived liefert die geforderte
     Implementierung
```

Virtueller Destruktor

Wird eine Klasse polymorph verwendet, so sollte beim Löschen eines Objekts immer das konkrete Objekt und nicht nur die Basisklasse freigegeben werden. Daher sollte immer der Destruktor der abgeleiteten Klasse aufgerufen werden. Machen Sie daher den Destrukor von virtuellen Klassen immer virtuell.

```
class Base {
  virtual int value() {return 5;}
  virtual ~Base() {}
};
```

Ein Beispiel: Konten einer Bank