Fachhochschule Bingen

Programmieren 2
Vererbung, virtuelle Methoden, abstrakte Methoden, Polymorphismus,

Prof. Dr. Maximilian Mengel,
Professur Programmiermethodik,
Grundlagen der Informatik und Multimedia
Gebäude 1, Raum 212
Tel.: 06721-409 152
E-Mail: mengel@fh-bingen.de

Vererbung: Aufeinander aufbauende Klassen

- Ohne Vererbung müssten mehrere ähnliche Klassen alle separat implementiert werden
  - Redundanter Code
  - Hoher Pflegeaufwand
  - Keine Transparenz der Verwandtschaft
- Vererbung:
  - Allgemeine Klasse, mit allgemeinen Eigenschaften und Methoden (Basisklasse)
  - Spezielle davon abgeleitete Klassen ergänzen nur die speziellen Eigenschaften und Methoden
  - Weniger Redundanter Code
  - Geringerer Pflegeaufwand

Beispiel

- Klasse Boot: Allgemeine ...
  - Eigenschaften: Länge, Breite, Zuladung, Tiefgang
  - Methoden: zuladen, abladen
- Klasse Motorboot: Spezielle ...
  - Eigenschaften: Leistung, Geschwindigkeit
  - Methoden: Gas geben, schalten, ...
- Klasse Segelboot: Andere spezielle ...
  - Eigenschaften: Masten, Segel
  - Methoden: Segel setzen, Segel einholen

Vererbung

- Bei hierarchisch aufgebauten Klassen erbt die neue Unterklasse die Methoden und Eigenschaften der Oberklasse
  - Beispiel: alle Eigenschaften des Boots auch bei einem Motorboot und Segelboot
- Die abgeleitete Klasse besitzt jedoch zusätzliche Eigenschaften und Methoden
  - Motorboot: Eigenschaften und Methoden des Motors
  - Segelboot: Eigenschaften und Methoden der Segel

Klassenhierarchie

- Eine Abgeleitete Klasse kann für eine weitere abgeleitete Klasse die Oberklasse sein
  - Segelboot
    - Segeljolle: Gleitpunkt
    - Jacht: Kielgewicht
- Eine Abgeleitete Klasse kann auch von mehreren Oberklassen abgeleitet sein
  - Motorboot & Segelboot als Oberklassen
    - Abgeleitet: Motorsegler
- => In C++ lassen sich komplexe Klassenhierarchien aufbauen

Syntax einer Abgeleiteten Klasse in C++

- Die bereits bekannte Syntax einer Klassendefinition wird folgendermaßen erweitert:
  - Nach dem Namen der abgeleiteten Klasse und vor der **{** kommt ein **:** gefolgt von einem Freigabe-Schlüsselwort und dem Namen der Oberklasse.
  - Bei mehreren Oberklassen werden jeweils mit Komma getrennt das Freigabe-Schlüsselwort und der Oberklassen-Name aufgeführt
  - Freigabe-Schlüsselworte:
    - private, protected, public

Beispiel: einfache Vererbung

```
class Boot {
  public:
    int breite;
    int laenge;
    Boot(int b, int l)
      { breite=b; laenge=l; }
    };

class Motorboot : public Boot {
  public:
    int leistung;
    Motorboot(int b, int l, int kw)
      { breite=b; laenge=l; leistung = kw;}
   };
```


Beispiel: mehrfache Vererbung

```
class Motorboot {
  ...
  };

class Segelboot {
  ...
  };

class Motorsegler: public Motorboot, public Segelboot {
  ...
  };
```

Zugriffsrechte: Schlüsselwort vor Oberklassennamen

- Durch das Schlüsselwort vor dem Oberklassennamen wird ausgedrückt:
  - Public:
    - Alle Eigenschaften und Methoden der Oberklasse stehen mit den dort vereinbarten Zugriffsrechten auch in der abgeleiteten Klasse zu Verfügung
  - Private:
    - Alle Eigenschaften und Methoden der Oberklasse werden zu privaten Eigenschaften und Methoden der abgeleiteten Klasse und sind von außen nicht zugreifbar

Das Schlüsselwort: Protected

- Bei der Vererbung möchte man häufig:
  - a) Abgeleiteten Klassen den Zugriff auf bestimmte Eigenschaften und Methoden erlauben
  - b) Diese Eigenschaften und Methoden jedoch von außen nicht zugreifbar machen
- Protected-Abschnitt **innerhalb einer Klasse**
  - Abgeleitete Klasse darf Eigenschaften und Methoden nutzen; aber: von außen nicht nutzbar
- Protected **vor Oberklassenname**:
  - Alle public-Eigenschaften und –Methoden der Oberklasse werden behandelt als wären sie dort protected vereinbart
- HINWEIS: Einschränkungen können immer nur verschärft, nie gelockert werden!

Zugriffsfehler C++

```
class Boot{
    int vmax;  // keine Angabe ==> privat:
    public:
       Boot(int h) {vmax = h;}
    };

class Motorboot : public Boot {
   int leistung;
   public:
     ...
       void drucke();
   };

void Motorboot::drucke() {
       cout << "Geschwindigkeit: " << vmax << "\n";
              // FEHLER auf vmax kann nicht zugegriffen werden
       cout << "Leistung" << leistung <<"\n"; // OK
       }
```


Protected C++

```
class Boot {
    protected:
       int vmax;
    public:
       Boot(int h) {vmax = h;}
    };

class Motorboot: public Boot {//auch protected oder privat würde funktionieren
   int leistung;
   public:
        ...
       void drucke();
   };

void Motorboote ::drucke() {
       cout << "Geschwindigkeit: "<<vmax << "\n";
              // jetzt klappts !!!
       cout << "Leistung" << leistung <<"\n";
       }
```

Konstruktoren/Destruktoren in Klassenhierarchien

- Wird ein Objekt einer abgeleiteten Klasse instanziert, dann wird
  - a) zuerst der Konstruktor der Oberklasse
  - b) dann der Konstruktor der Abgeleiteten Klasse aufgerufen
- Wird ein Objekt zerstört, dann wird
  - a) erst der Destruktor der Abgeleiteten Klasse
  - b) der Destruktor der Oberklasse aufgerufen
- Bei mehreren aufeinander aufbauenden Oberklassen gilt entsprechendes
- Bei mehreren Basisklassen, einer Abgeleiteten Klasse werden die Konstruktoren der Basisklassen von links nach rechts, die Destruktoren von rechts nach links aufgerufen

Beispiel 1: Konstruktoren / Destruktoren in Klassenhierarchie

```
class base {
    public:
      base() { cout << „Konstruktor base \n“; }
      ~base() { cout << „Destruktor base \n“; }
    };

class derived1: public base {
   public:
        derived1() {cout << „Konstruktor derived1 \n“; }
        ~derived1() { cout << „Destruktor derived1 \n“; }
   };

class derived2: public derived1 {
   public:
        derived2() {cout << „Konstruktor derived2 \n“; }
        ~derived2() { cout << „Destruktor derived2 \n“; }
   };
```


Beispiel 1: Ausgabe

```
main()
{
   derived2 ob;
}
```

Obiges „Programm“ erzeugt folgende Ausgaben:

Konstruktor base
Konstruktor derived1
Konstruktor derived2
Destruktor derived2
Destruktor derived1
Destruktor base

Beispiel 2: Konstruktoren / Destruktoren in Klassenhierarchie

```
class base1 {
    public:
      base1() { cout << „Konstruktor base1 \n“; }
      ~base1() { cout << „Destruktor base1 \n“; }
    };

class base2 {
   public:
        base2() {cout << „Konstruktor base2 \n“; }
        ~base2() { cout << „Destruktor base2 \n“; }
   };

class derived: public base1, public base2 {
   public:
        derived() {cout << „Konstruktor derived \n“; }
        ~derived() { cout << „Destruktor derived \n“; }
   };
```

Beispiel 2: Ausgabe

```
main()
{
   derived ob;
}
```

Obiges „Programm“ erzeugt folgende Ausgaben:

Konstruktor base1
Konstruktor base2
Konstruktor derived
Destruktor derived
Destruktor base2
Destruktor base1

Aufruf eines Oberklassen-Konstruktors mit Parametern

- Das genannte Verhalten gilt für die parameterlosen Standard-Konstruktoren
- Häufig möchte man in der Abgeleiteten Klasse den Oberklassen-Konstruktor aufrufen und diesem Parameter übergeben. Dies ist in C++ möglich:
  - Nach den Parametern des Konstruktors der Abgeleiteten Klasse und vor der { des Anweisungsblocks schreibt man einen : und den Aufruf des Oberklassen-Konstruktors.
  - Parameter des Konstruktors der Abgeleiteten Klasse können hierbei als Aufrufparameter des Oberklassen-Konstruktors dienen

Beispiel: Konstruktoren mit Parametern

```
class base {
  protected:
    int i;
  public:
    base(int x) {i = x;}
   };

class derived: public base {
  int j;
  public:
    derived(int x, int y) : base(y)
      {j = x;}
    void show() {cout << i << " " << j << "\n";
   };
```


Initialisierung von Eigenschaften

- Ähnlich wie Parameter an Konstruktoren der Oberklassen übergeben werden, können auch Eigenschaften direkt initialisiert werden
  - Nach der Parameterliste wird der Eigenschaftenname mit (Initialisierungswert / -Parameter) aufgeschrieben.
  - Mehrere Eigenschaften können (wie auch beim Aufruf von mehreren Konstruktoren) per Komma getrennt Initialisiert werden.

Beispiel: Initialisierung von Eigenschaften

```
class base {
  protected:
    int i;
  public:
    base(int x): i(x) {}
  };

class derived: public base {
  int j, k;
  public:
    derived(int x, int y) : base(y), j(x), k(x+y)
      {}
  };
```


Virtuelle Methoden

- Manchmal „passen“geerbten Methoden der Oberklasse nicht zu der Abgeleiteten Klasse; dann muss die entsprechende Methode in der Abgeleiteten Klasse angepasst (sprich neu implementiert) werden
- Dieses Überschreiben einer geerbten Methode ist nur erlaubt, wenn die Oberklasse dies erlaubt
- Die entsprechende Methode muss (in der Oberklasse) als virtuell vereinbart sein:
  - Im Methodenkopf muss noch vor dem Rückgabewert das Schlüsselwort **virtual** stehen
  - virtual gilt für diese Methode dann in allen Abgeleiteten Klassen (auch über mehrere Stufen hinweg)
  - virtual kann (muss jedoch nicht) in den Abgeleiteten Klassen wiederholt werden

Beispiel: virtuelle Methode

```
class base {
  ...
  public:
    virtual void tutEtwas(){cout << “ich tue was...\n“}
  };

class derived: public base {
  ...
  public:
    virtual void tutEtwas() // hier virtual kein muss !
      { cout << “ich tue etwas anderes !!\n“; }
  };
```


Abstrakte Klasse und rein virtuelle Methoden

- Manchmal möchte man Oberklassen schreiben, die verschiedene Methoden und Eigenschaften der Abgeleiteten Klassen definieren, die aber:
  - A) weder alle Methoden selbst Implementieren
  - B) noch überhaupt geeignet sind Objekte anzulegen
- Solche Klassen werden **abstrakt** genannt
- Methoden die in einer Klasse vereinbart, aber nicht implementiert werden werden **rein virtuell** genannt
- Eine Klasse die mindestens eine rein virtuelle Methode besitzt ist immer auch abstrakt

Rein virtuelle Methoden

- Eine von einer abstrakten Klasse Abgeleiteten Klasse, die **nicht** alle rein virtuell definierten Methoden implementiert ist ebenfalls abstrakt
- Im Umkehrschluss ist eine Abgeleitete Klasse nicht abstrakt wenn **alle** als rein virtuell definierten Methoden implementiert sind
- Um eine Methode als rein virtuell zu vereinbaren wird anstelle des Methoden-rumpfs einfach ein = 0 geschrieben
- **Von abstrakten Klassen können keine Objekte angelegt werden !!!**

Beispiel: rein virtuelle Methode

```
class base {
  ...
  public:
    virtual void irgendwas() = 0;
  };

class derived: public base {
  ...
  public:
    virtual void irgendwas()
      { cout << "jetzt tue ich etwas!!\n"; }
  };
```


Zeiger auf Basisklassen und auf Abgeleitete Klasse

- Ein Zeiger auf eine Klasse kann i. Allg. nicht auf ein Objekt einer anderen Klasse zeigen
- Wichtige Ausnahme:
  - Ein Zeiger (und auch eine Referenz) auf eine Basisklasse darf immer auch auf Objekte von Abgeleiteten Klassen zeigen
  - Umgekehrt funktioniert dies nicht
- Da der Eigenschaften- und Methodenzugriff über einen Zeiger der Basisklasse erfolgt können nur die Eigenschaften und Methoden der Basisklasse genutzt werden

Beispiel: Zeiger auf Basisklasse, Teil1

```
class base {
  public:
    int i;
    int getiQuadr() { return i*i; }
  };

class derived: public base {
  public:
    int j;
    int getjQuadr() { return j*j; }
  };
```

Beispiel: Zeiger auf Basisklasse, Teil 2

```
main()
{
  base *bp;
  derived d;
  bp = &d;   // OK, da derived von base abgeleitet

  bp->i = 5; // OK da Eigenschaft von base
  cout << bp->getiQuadr <<"\n"; //OK da Methode von base

  /* folgendes funktioniert nicht:
  bp->j = 5; // Eigenschaft von derived !!
  cout << bp->getjQuadr <<"\n"; //Methode von derived !!
  */

  // folgendes funktioniert mittels cast, ist aber unüblich
  ((derived*)bp)->j = 5;
}
```


Polymorphie

- In C steht bereits vor der Ausführung immer fest welcher Code durch den Aufruf einer bestimmten Funktion abgearbeitet wird
- Bei C++ existiert die Möglichkeit das erst zur Laufzeit festgelegt wird welcher Code wirklich abgearbeitet wird.
- Dies nennt man Polymorphie zur Laufzeit oder auch spätes Binden

Polymorphie zur Laufzeit

- Diese Polymorphie tritt auf wenn folgende Dinge gleichzeitig zutreffen:
  - Auf eine Objekt einer Abgeleiteten Klassen wird über einen Zeiger oder eine Referenz auf die Basisklasse zugegriffen
  - Es werden virtuelle oder rein virtuelle Methoden genutzt
- Die Idee ist hierbei folgende:
  - In der Basisklasse wird ein Interface aus Methoden festgelegt, das dann für alle Abgeleiteten Klassen gilt.
  - Implementierung des Interfaces : speziell in den Abgeleiteten Klassen oder allgemein in der Basisklasse
  - Mittels Zeiger können Objekte aller Klassen gleich genutzt werden (da alle das gleiche Interface besitzen), aber es wird automatisch immer die richtige Implementierung des Interfaces ausgeführt
  - Beispiel: siehe C++-Datei Polymorphie.cpp