C++ Teil 13

Sven Groß

1. Feb 2016

Themen der letzten Vorlesung

- Projekte mit mehreren Dateien: Module
- Klassen-Design am Beispiel `SudokuSolver`
- Numerik-Werbung: Zweiphasenströmungen mit DROPS

Heutige Themen

1. Zugriffskontrolle: `protected`
2. Vererbung
3. Polymorphie
   - Virtuelle Methoden
   - Abstrakte Basisklassen

Zugriffskontrolle: protected

```
class Form2D
{
  private:
    int farbe;
  protected:
    double hoehe, breite;
  public:
    Form2D( double h=0, double b=0); // Konstruktor
    void setzeGroesse( double h, double b)
    { hoehe= h; breite= b; }
};
```

- `protected` markiert **geschützten** Bereich der Klasse
- weitere Stufe zwischen `private` und `public`:

Zugriff	`public`	`protected`	`private`
eigene Methoden + Friends	✓	✓	✓
abgeleitete Methoden	✓	✓	✗
alle anderen	✓	✗	✗

Vererbung

```
class Rechteck: public Form2D
{
  public:
    double Flaeche() const { return hoehe*breite; }
};

class Dreieck: public Form2D
{
    double winkel;
  public:
    double Flaeche() const { return hoehe*breite/2; }
};
```

- Klassen `Rechteck` und `Dreieck` sind von Klasse `Form2D` **abgeleitet** und erweitern diese.
- `Form2D` ist gemeinsame **Basisklasse**.
- abgeleitete Klassen erben Methoden und Attribute der Basisklasse, z.B. `Dreieck` hat Methoden `setzeGroesse`, `Flaeche` und Attribute `winkel, breite, hoehe` (sowie `farbe` ohne Zugriff).

Vererbung (2)

```
Form2D f( 1, 2);
Rechteck r;
// Rechteck r( 1, 2); nicht erlaubt,
//                    Konstruktor nicht geerbt
Dreieck d;
f.setzeGroesse( 3, 4);
r.setzeGroesse( 3, 4);
d.setzeGroesse( 3, 4);
cout << r.Flaeche() << endl;  // 12
cout << d.Flaeche() << endl;  //  6

Form2D *fzeiger= &d;
fzeiger->setzeGroesse( 4, 4);
cout << d.Flaeche() << endl;  //  8
```

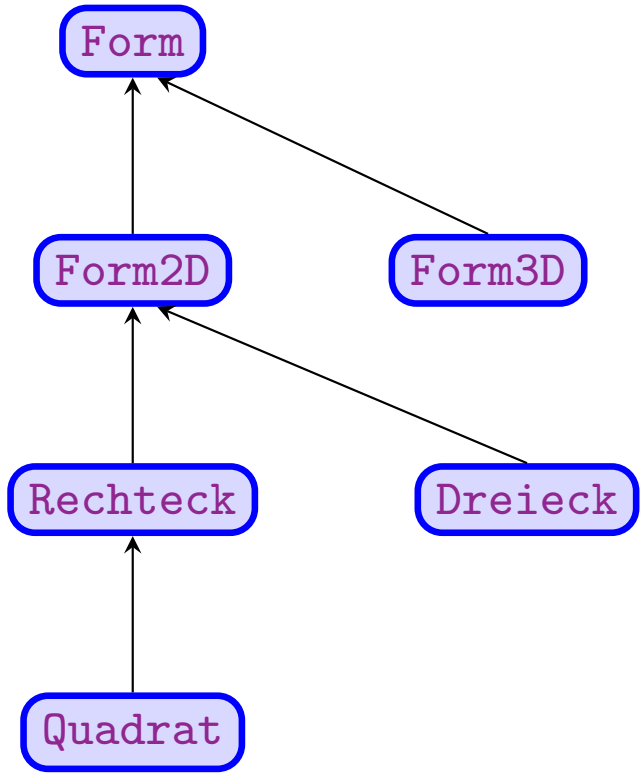
- Konstruktoren, Destruktor, Zuweisungsoperator und Friends werden nicht vererbt
- `Dreieck` ist eine `Form2D`:
  abgeleitete Klasse kann aus Sicht der Basisklasse angesprochen werden

Vererbung – ist-ein vs. hat-ein

- Vererbung bildet **Ist-ein-Relation** ab:
  Ein `Dreieck` *ist eine* `Form2D`
- weitere Vererbungen denkbar:
  z.B. `Quadrat` *ist ein* `Rechteck`,
  `Kugel` *ist eine* `Form3D`

```
class Quadrat: public Rechteck
{...};

class Kugel:   public Form3D
{...};
```



- kennen schon: `ifstream` *ist ein* `istream`

- Aber: `SudokuSolver` ist kein `Sudoku`,
  sondern *hat ein* `Sudoku`
- **Hat-ein-Relation** durch Attribut ausdrücken

⇝ `SudokuSolver` besitzt ein `Sudoku`-Objekt als Attribut

Vererbung und Konstruktoren

- Konstruktoren in abgeleiteten Klassen erzeugen nur eigene Attribute, geerbte Attribute werden durch Basisklasse erzeugt!
- Reihenfolge: erst Basisklassen-Attribute, dann die eigenen Attribute

```
Dreieck::Dreieck( double b, double h, double w)
 : Form2D( b, h),
   winkel( w)
{}
```

oder ohne Initialisierungsliste:

```
Dreieck::Dreieck( double b, double h, double w)
{
  // Basisklassen-Atrribute wurden per
  // Default-Konstruktor Form2D() erzeugt
  winkel= w;
  breite= b;
  hoehe= h;
}
```

Vererbung und implizite Elemente

- Konstruktoren, Destruktor, Zuweisungsoperator werden nicht vererbt
  $\rightsquigarrow$ implizit definiert, falls nichts anderes angegeben
- Destruktoren in abgeleiteten Klassen kümmern sich um eigene Attribute, danach wird automatisch Destruktor der Basisklasse gerufen

```
Dreieck::~Dreieck()
{
  cout << "Dreieck␣wird␣zerstoert..." << endl;
} // hier wird automatisch ~Form() aufgerufen
```

- impliziter Kopierkonstruktor ruft Kopierkonstruktor der Basisklasse auf, anschließend werden Kopien der eigenen Attribute erzeugt
- impliziter Zuweisungsoperator ruft Zuweisungsoperator der Basisklasse auf, anschliesend werden eigene Attribute zugewiesen

Vererbung – Zugriffsklassen

- fast immer `public`-Vererbung:
  Zugriffsklassen der geerbten Elemente bleiben erhalten

z.B.

```
Dreieck::setzeGroesse          public
Dreieck::hoehe                 protected
Dreieck::breite                protected
// farbe ohne Zugriff in Dreieck
```

- selten: `protected`-Vererbung:
  alle geerbten `public`-Elemente werden `protected`

```
class Rechteck: protected Form2D
{
  public:
    double Flaeche() const { return hoehe*breite; }
};
```

$\rightsquigarrow$ `Rechteck::setzeGroesse` ist `protected`

- selten: `private`-Vererbung:
  alle geerbten `public`- und `protected`-Elemente werden `private`

Beispiel Vererbung

```
class Form2D {
  private:
    int farbe;
  protected:
    double hoehe, breite;
  public:
    Form2D( double h=0, double b=0); // Konstruktor
    void setzeGroesse( double h, double b);
    double Flaeche() const { return -1; }
};

class Rechteck: public Form2D {
  public:
    double Flaeche() const { return hoehe*breite; }
};

class Dreieck: public Form2D {
    double winkel;
  public:
    double Flaeche() const { return hoehe*breite/2; }
};
```

Beispiel Vererbung (2)

```
Form2D f( 1, 2);
Rechteck r;
// Rechteck r( 1, 2); nicht erlaubt,
//                    Konstruktor nicht geerbt
Dreieck d;
r.setzeGroesse( 3, 4);
d.setzeGroesse( 3, 4);
cout << r.Flaeche() << endl;  // 12
cout << d.Flaeche() << endl;  //  6

Form2D *fzeiger= &d;
fzeiger->setzeGroesse( 4, 4);
cout << d.Flaeche() << endl;  //  8
cout << fzeiger->Flaeche() << endl;  //  -1
```

- `Dreieck` ist eine `Form2D`:
  abgeleitete Klasse kann aus Sicht der Basisklasse angesprochen werden
- Schön wäre, wenn `fzeiger->Flaeche()` Fläche des Dreiecks liefern würde

⇝ möglich mit **virtuellen Methoden**

Virtuelle Methoden und Polymorphie

```
class Form2D {
  ...
public:
  ...
  virtual double Flaeche() const { return -1; }
};
```

- **Virtuelle Methoden** ermöglichen Zugriff auf abgeleitete Methoden via Basisklasse
- **Polymorphie:** Basisklasse kann verschiedene Gestalten annehmen, verhält sich wie abgeleitete Klassen
- Klassen mit virtuellen Methoden heißen **polymorph**

```
Form2D *fzeig= &f,
       *rzeig= &r,
       *dzeig= &d;
cout << fzeig->Flaeche() << endl;   //   -1
cout << rzeig->Flaeche() << endl;   //   12 dank Polymorphie
cout << dzeig->Flaeche() << endl;   //    8 dank Polymorphie
```

Polymorphie und virtueller Destruktor

Normaler Destruktor führt zu Problemen:

```
Form2D *zeig= new Dreieck; // Dreieck-Konstruktor
...                        // -> Form2D-Konstruktor

delete zeig;               // nur Form2D-Destruktor !!!
```

- Polymorphie benötigt i.d.R. virtuellen Destruktor der Basisklasse:
  `virtual ~Form2D();`

⇝ `delete` bewirkt Aufruf des Dreieck-Destruktors im obigen Beispiel

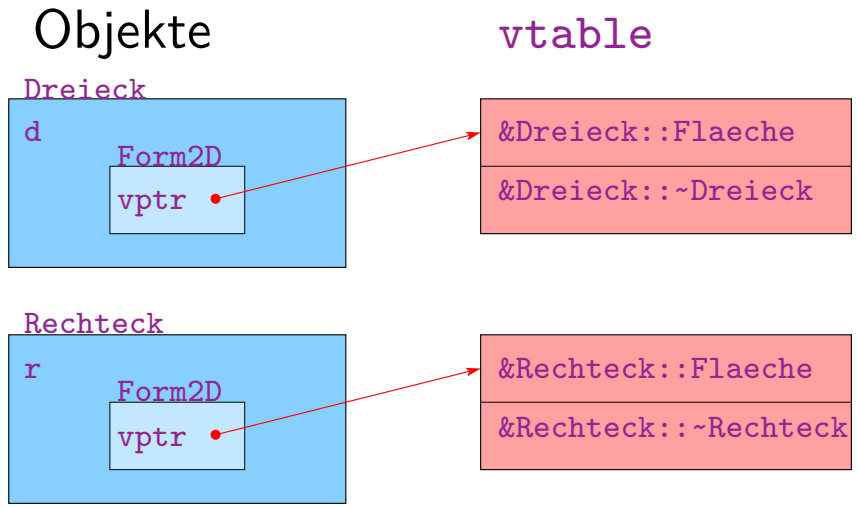
- Besonders wichtig bei dynamischer Speicherverwaltung,
  sonst Speicherlecks
- **Best Practice:** polymorphe Klassen immer mit virtuellem Destruktor

Späte Bindung und vtables

- normale Methoden: Aufruf bekannt zur Compile-Zeit, **frühe Bindung**
- virtuelle Methoden: Aufruf erst zur Laufzeit bekannt, **späte Bindung**

⇝ Wie funktioniert das?

- Polymorphe Klasse speichert intern Zeiger `vptr` auf Tabelle `vtable` mit Funktionszeigern
- `vtable` wird durch Konstruktor der abgeleiteten Klasse automatisch richtig initialisiert
- Aufruf virtueller Methoden indirekt über `vptr` statt direkt, daher etwas teurer
- passiert automatisch im Hintergrund



Abstrakte Basisklassen

- **Rein virtuelle Methoden** (*pure virtual*) gekennzeichnet durch `= 0`

```
class Form2D {
  ...
public:
  ...
  virtual double Flaeche() const = 0;
};
```

- Klassen mit rein virtuellen Methoden heißen **abstrakt**
  - können keine Objekte bilden
  - einziger Zweck: **Schnittstellen-Definition**
- Abgeleitete Klassen müssen rein virtuelle Methoden überschreiben (Implementierung der Schnittstelle), um Objekte bilden zu können

```
// Form2D f; // nicht erlaubt, da abstrakte Basisklasse

Rechteck r;   // Rechteck nicht abstrakt,
              // da Methode Flaeche() ueberschrieben wird
Form2D *zeig= &r;   // ok
```

Abstrakte Basisklassen – Beispiel

```
1  class Form2D { // als abstrakte Klasse
2  protected:
3    double hoehe, breite;
4  public:
5    Form2D( double h, double b) { setzteGroesse( h, b); }
6    void setzeGroesse( double h, double b);
7    virtual double Flaeche() const = 0; // rein virtuell
8    virtual string Name() const { return "Form2D"; }
9    virtual ~Form2D() {} // best practice
10 };
11
12 class Dreieck: public Form2D { // nicht abstrakt
13 public:
14   Dreieck( double h, double b) : Form2D( h, b) {}
15   double Flaeche() const { return hoehe*breite/2; }
16   string Name() const { return "Dreieck"; }
17 };
18
19 class Rechteck: public Form2D { // nicht abstrakt
20 public:
21   Rechteck( double h, double b) : Form2D( h, b) {}
22   double Flaeche() const { return hoehe*breite; }
23   string Name() const { return "Rechteck"; }
24 };
25
26 class Quadrat: public Rechteck { // nicht abstrakt
27 public:
28   Quadrat( double b) : Rechteck( b, b) {}
29   string Name() const { return "Quadrat"; }
30 };
```

Abstrakte Basisklassen – Beispiel (2)

```
void Info( const Form2D& f)
{
  // funktioniert dank Polymorphie
  cout << f.Name() << "␣hat␣Flaeche␣" << f.Flaeche() << endl;
}

int main()
{
  Dreieck  d( 2, 3);
  Rechteck r( 2, 3);
  Quadrat  q( 4);

  Info( d); // Dreieck hat Flaeche 3
  Info( r); // Rechteck hat Flaeche 6
  Info( q); // Quadrat hat Flaeche 16
  return 0;
}
```