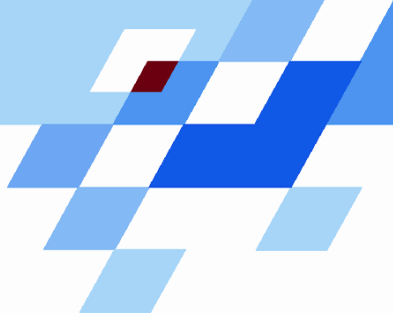


Wintersemester 2006/07

Einführung in die Informatik für Naturwissenschaftler und Ingenieure

(alias **Einführung in die Programmierung)**

(Vorlesung)

Prof. Dr. Günter Rudolph

Fachbereich Informatik

Lehrstuhl für Algorithm Engineering

Inhalt

- Einführung
- Konstruktoren / Destruktoren

Ziele von Klassen

- Kapselung von Attributen (wie **`struct`** in Programmiersprache C)
- Kapselung von klassenspezifischen Funktionen / Methoden
- Effiziente Wiederverwendbarkeit
  - Vererbung
  - Virtuelle Methoden
- Grundlage für Designkonzept für Software

Schlüsselwort: `class`

- Datentypdefinition / Klassendefinition analog zu **`struct`**

```
struct Punkt {
  double x, y;
};
```

→

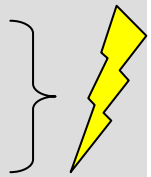
```
class Punkt {
  double x, y;
};
```

Unterschied:

```
Punkt p;
p.x = 1.1;
p.y = 2.0;
```

→ ?

```
Punkt p;
p.x = 1.1;
p.y = 2.0;
```



Zugriff gesperrt!

Schlüsselwort: `class`

- Datentypdefinition / Klassendefinition analog zu **`struct`**

```
struct Punkt {
  double x, y;
};
```

Komponenten sind öffentlich! (**`public`**)

```
class Punkt {
  double x, y;
};
```

Komponenten sind privat! (**`private`**)

⇒ Kontrolle über Zugriffsmöglichkeit sollte steuerbar sein!

⇒ Man benötigt Mechanismus, um auf Komponenten zugreifen zu können!

prozedural

```
struct Punkt {
  double x, y;
};
void SetzeX(Punkt &p, double w);
void SetzeY(Punkt &p, double w);
double LeseX(Punkt &p);
double LeseY(Punkt &p);
```

objekt-orientiert

```
class Punkt {
  double x, y;
public:
  void SetzeX(double w);
  void SetzeY(double w);
  double LeseX();
  double LeseY();
};
```

⇒ Schlüsselwort **`public`** : alles Nachfolgende ist öffentlich zugänglich!

```
struct Punkt {

double x, y;

};
```

```
void Verschiebe(Punkt &p,
                double dx, double dy);
bool Gleich(Punkt &a, Punkt& b);
double Norm(Punkt &a);
```

```
class Punkt {

private:
  double x, y;

public:

  void   SetzeX(double w);
  void   SetzeY(double w);
  double LeseX();
  double LeseY();
  void   Verschiebe(double dx, double dy);
  bool   Gleich(Punkt &p);
  double Norm();
};
```

```
class Punkt {

private:
  double x, y;

public:

  void   SetzeX(double w) { x = w; }
  void   SetzeY(double w) { y = w; }
  double LeseX() { return x; }
  double LeseY() { return y; }
  void   Verschiebe(double dx, double dy);
  bool   Gleich(Punkt &p);
  double Norm();
};
```

Implementierung: direkt in der Klassendefinition

```
void Punkt::Verschiebe(double dx, double dy) {
  x += dx;
  y += dy;
}
```

Implementierung: außerhalb der Klassendefinition

Prinzip des *‘information hiding‘*

Trennung von Klassendefinition und Implementierung

⇒ am besten in verschiedenen Dateien!

Punkt.h

Klassen-
definition

Punkt.cpp

Klassen-
implementierung

bei Implementierung
außerhalb der Klassendefinition:

Angabe des Klassennames nötig!

↓

Datentyp Klassenname::Methode(…){
}

*.h → „**h**eader“ *.cpp → „**cp**lus**p**lus“

Datei: Punkt.h

```
class Punkt {

private:
  double x, y;

public:

  void   SetzeX(double w);
  void   SetzeY(double w);
  double LeseX();
  double LeseY();
  void   Verschiebe(double dx, double dy);
  bool   Gleich(Punkt &p);
  double Norm();
};
```

Die **Klassendefinition** wird nach außen (d.h. **öffentlich**) **bekannt** gemacht!

Die **Implementierung** der Methoden wird nach außen hin **verborgen**!

Datei: Punkt.cpp

```
#include <math.h>
#include "Punkt.h"

void Punkt::SetzeX(double w) { x = w; }
void Punkt::SetzeY(double w) { y = w; }
double Punkt::LeseX() { return x; }
double Punkt::LeseY() { return y; }

void Punkt::Verschiebe(double dx, double dy) {
  x += dx;
  y += dy;
}

bool Punkt::Gleich(Punkt &p) {
  return x == p.LeseX() && y == p.LeseY() ? true : false;
}

double Punkt::Norm() {
  return sqrt(x * x + y * y);
}
```

Überladen von Methoden

```
class Punkt {

private:
  double x, y;

public:

  bool Gleich(Punkt &p);
  bool Gleich(double ax, double ay) {
    return (x == ax && y == ay) ? true : false;
};
```

mehrere Methoden mit **gleichem Namen**

wie unterscheidbar? → durch ihre verschiedenen Signaturen / Argumentlisten!

```
Punkt p1, p2;
// …
if (p1.Gleich(p2) || p1.Gleich(1.0, 2.0)) return;
```

Umständlich:

```
Punkt p;
p.SetzeX(1.3);
p.SetzeY(2.9);
```

?

wie bei `struct Punkt` ?

```
Punkt p = { 1.3, 2.9 };
```

⇒ **Konstruktoren**

```
class Punkt {
private:
  double x, y;
public:
  Punkt() { x = y = 0.0; }
  Punkt(double ax, double ay) {
    x = ax; y = ay;
};
```

!

identisch zu:
`Punkt p1(0,0);`

```
Punkt p1;

Punkt p2(1.3, 2.9);
```

Aufgaben eines Konstruktors:

- Saubere Initialisierung eines Objekts
  → man kann erzwingen, dass nur initialisierte Instanzen erzeugt werden

- ggf. Bereitstellung von dynamischen Speicherplatz

- ggf. Benachrichtigung eines anderen Objekts über Erzeugung (Registrierung)

- durch Überladen: bequeme Möglichkeiten zur Initialisierung
  Bsp: Default-Werte

```
Punkt();                          z.B. wie Punkt(0.0, 0.0)
Punkt(double x);                  z.B. wie Punkt(x, 0.0);
Punkt(double x, double y);
```

- was immer gerade nötig ist …

Merke:

- **Konstruktoren** heißen exakt wie die Klasse, zu der sie gehören!
- Wenn eine Instanz einer Klasse angelegt wird
  → **automatischer Aufruf** des Konstruktors!
- Da nur Instanz angelegt wird (Speicherallokation und Initialisierung)
  wird **kein Wert zurückgegeben**
- **kein Rückgabewert** (auch nicht `void`)
- Konstruktoren können **überladen** werden
- bei **mehreren Konstruktoren** wird der ausgewählt,
  der am besten zur Signatur / Argumentliste passt → eindeutig!

Instanzen von Klassen können auch **dynamisch erzeugt** werden:

```
Punkt *p1 = new Punkt(2.1, 3.3);
Punkt *p2 = new Punkt();
Punkt *p3 = new Punkt;
```

} gleichwertig!

Achtung!

Das Löschen nicht vergessen! Speicherplatzfreigabe!

```
delete p1;
```

etc.

Destruktoren

- dual zu Konstruktoren
- **automatischer Aufruf**, wenn Instanz Gültigkeitsbereich verlässt
- heißen exakt wie die Name der Klasse, zu der sie gehören
  Unterscheidung von Konstruktoren bzw. Kennzeichnung als Destruktor durch vorangestellte Tilde ~
  Bsp: `~Punkt();`
- Destruktoren haben **niemals** Parameter
- **Zweck:** Aufräumarbeiten
  - z.B. Schließen von Dateien
  - z.B. Abmeldung bei anderen Objekten (Deregistrierung)
  - z.B. **Freigabe von dynamischen Speicher**, falls vorher angefordert
  - … und was immer gerade nötig ist

Illustration:

```
Punkt::Punkt(double ax, double ay) {
  x = ax; y = ay;
  cout << "Konstruktor aufgerufen!" << endl;
}

Punkt::~Punkt() {
  cout << "Destruktor aufgerufen!" << endl;
}
```

```
int main() {
  cout << "Start" << endl;
  {
    Punkt p(1.0, 2.0);
  }
  cout << "Ende" << endl;
}
```

Ausgabe:

```
Start
Konstruktor aufgerufen!
Destruktor aufgerufen!
Ende
```