

Einführung in die Programmierung

Wintersemester 2019/20

<https://ls11-www.cs.tu-dortmund.de/teaching/ep1920vorlesung>

Dr.-Ing. Horst Schirmeier
(mit Material von Prof. Dr. Günter Rudolph)

Arbeitsgruppe Eingebettete Systemsoftware (LS 12)
und Lehrstuhl für Algorithm Engineering (LS11)

Fakultät für Informatik

TU Dortmund

Kapitel 8: Klassen

Inhalt

- Einführung in das Klassenkonzept
- Attribute / Methoden
- Konstruktoren / Destruktoren
- Schablonen

Klassen

Kapitel 8

Klassen

Kapitel 8

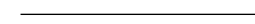
Ziele von Klassen

- Kapselung von Attributen (wie `struct` in Programmiersprache C)
- Kapselung von klassenspezifischen Funktionen / Methoden
- Effiziente Wiederverwendbarkeit
 - Vererbung → Kapitel 10
 - Virtuelle Methoden → Kapitel 11
- Grundlage für Designkonzept für Software

Schlüsselwort: `class`

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

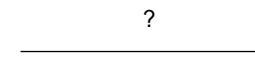
```
struct Punkt {
    double x, y;
};
```



```
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**)

- ⇒ Kontrolle über Zugriffsmöglichkeit sollte steuerbar sein.
- ⇒ Man benötigt Mechanismus, um auf Komponenten zugreifen zu können!
- ⇒ sogenannte **Methoden**

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

Komponenten sind
privat (**private**)

prozedural

```
struct Punkt {
    double x, y;
};
void setzeX(Punkt &p, double w);
void setzeY(Punkt &p, double w);
double leseX(Punkt const &p);
double leseY(Punkt const &p);
```

objektorientiert

```
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 const &p);
    double norm();
};
```

} Methoden

Klasse = Beschreibung von **Eigenschaften** und **Operationen**

- ⇒ Eine Klasse ist also die Beschreibung des **Bauplans** (Konstruktionsvorschrift) für konkrete (mit Werten belegte) **Objekte**
- ⇒ Eine Klasse ist **nicht** das Objekt selbst
- ⇒ Ein Objekt ist eine **Instanz** / Ausprägung einer Klasse

Objekt = Zusammenfassung von Daten / Eigenschaften und Operationen

Zugriff auf Daten **nur über Operationen** der Klasse;
man sagt auch: „dem Objekt wird eine Nachricht geschickt“

Objektname.Nachricht(Daten)

Methode = Operation, die sich auf ein Objekt einer Klasse anwenden lässt
(Synonyme: Element- oder Klassenfunktion)

- **Klasse:**
Beschreibung einer Menge von Objekten mit gemeinsamen Eigenschaften und Verhalten.
Ist ein **Datentyp!**
- **Objekt:**
Eine konkrete Ausprägung, eine Instanz, ein Exemplar der Klasse.
Belegt Speicher!
Besitzt Identität.
Objekte tun etwas; sie werden als Handelnde aufgefasst.
- **Methode / Klassenfunktion:**
Beschreibt das Verhalten eines Objektes.
Kann als spezielle Nachricht an das Objekt aufgefasst werden.

Anwendungsproblem:

⇒ **Modellierung** ⇒ Reduzierung auf das „Wesentliche“

... „wesentlich“ im Sinne unserer Sicht auf die Dinge bei diesem Problem.
→ es gibt verschiedene Sichten auf dasselbe Problem!

⇒ schon bei der Problemanalyse denken im Sinne von **Objekten** und ihren **Eigenschaften** und **Beziehungen** untereinander

Objektorientierte Programmierung (OOP):

- Formulierung eines Modells in **Konzepten & Begriffen der realen Welt**
- nicht in computertechnischen Konstrukten wie Haupt- und Unterprogramm

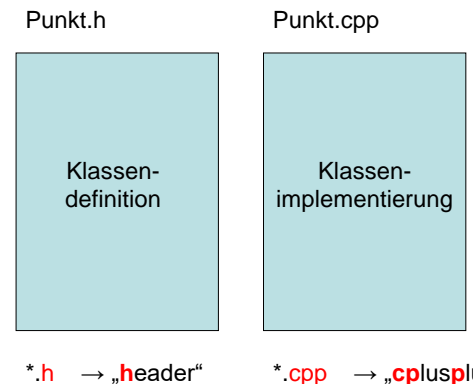
```
class Punkt {
private:
    double x, y;
public:
    void setzeX(double w) { x = w; }
    void setzeY(double w) { y = w; }
    double leseX() { return x; }
    double leseY() { return y; }
    void verschiebe(double dx, double dy);
    bool gleich(Punkt const &p);
    double norm();
};

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

Prinzip des 'information hiding'

Trennung von **Klassendefinition** und **Implementierung**

⇒ am besten in verschiedenen Dateien!



bei Implementierung außerhalb der Klassendefinition:
Angabe des Klassennamens nötig!
↓
Datentyp **Klassenname::**Methode(...){
}

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 const &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 <cmath>
#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 const &p) {
    return (x == p.leseX() && y == p.leseY());
}
double Punkt::norm() {
    return sqrt(x * x + y * y);
}
```

Überladen von Methoden

```
class Punkt {
private:
    double x, y;
public:
    bool gleich(Punkt const &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;
```

Initialisierung umständlich:

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

?

→

nein!
(x, y sind private)

wie bei struct Punkt ?

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

⇒ **Konstruktoren**

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

!

```
Punkt p1;
Punkt p2(1.3, 2.9);
```

identisch zu:
Punkt p1(0, 0);

Initialisierer-Liste

Konstruktoren

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

Konstruktoren (ab C++11)

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

Aufgaben eines Konstruktors:

- Saubere **Initialisierung** eines Objekts
→ man kann erzwingen, dass vollständig initialisierte Instanzen erzeugt werden
- ggf. Bereitstellung von **dynamischem 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 ...

Delegation / Delegierung von Konstruktoren (C++11)

- Bei überladenen Konstruktoren: Ein Konstruktor ruft einen anderen auf
- Die Arbeit wird also **delegiert** (*delegating constructors*)
- **Vermeidet duplizierten Code** bei aufwendigen Konstruktoren

```
Punkt::Punkt() : Punkt(0.0, 0.0){}
// Aufruf des anderen Konstruktors in Initialisierer-Liste

Punkt::Punkt(double ax, double ay) {
    x = ax; y = ay;
}
```

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 Rückgabety (auch nicht `void`)
- Konstruktoren können **überladen** werden
- bei **mehreren Konstruktoren** wird der ausgewählt, der am besten zur Signatur / Argumentliste passt → eindeutig!
- Konstruktoren können Aufgaben an überladene Konstruktoren **delegieren** (C++11)

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;
delete p2;
delete p3;
```

Destruktoren

- Gegenstück 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 **keine 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;
    return 0;
}
```

Ausgabe:

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

Noch ein Beispiel ...

```
Punkt::Punkt(double ax, double ay) : x(ax), y(ay) {
    cout << "K: " << x << " " << y << endl;
}
Punkt::~~Punkt() {
    cout << "D: " << x << " " << y << endl;
}
```

```
int main() {
    cout << "Start" << endl;
    Punkt p1(1.0, 0.0);
    Punkt p2(2.0, 0.0);
    cout << "Ende" << endl;
    return 0;
}
```

Ausgabe:

```
Start
K: 1.0 0.0
K: 2.0 0.0
Ende
D: 2.0 0.0
D: 1.0 0.0
```

Konstruktoren:

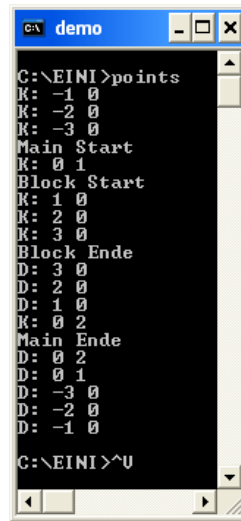
Aufruf in Reihenfolge der Datendefinition

Destruktoren:

Aufruf in **umgekehrter** Reihenfolge

Großes Beispiel ...

```
Punkt g1(-1.0, 0.0);
Punkt g2(-2.0, 0.0);
int main() {
    cout << "Main Start" << endl;
    Punkt q1(0.0, 1.0);
    {
        cout << "Block Start" << endl;
        Punkt p1(1.0, 0.0);
        Punkt p2(2.0, 0.0);
        Punkt p3(3.0, 0.0);
        cout << "Block Ende" << endl;
    }
    Punkt q2(0.0, 2.0);
    cout << "Main Ende" << endl;
}
Punkt g3(-3.0, 0.0);
```



```
class Punkt {
private:
    int id;
public:
    Punkt();
    ~Punkt();
}; // Punkt.h

#include "Punkt.h"
Punkt::Punkt() {
    static int cnt = 0; // ← statische lokale Var.
    id = ++cnt;
    cout << "K" << id << endl;
}
Punkt::~Punkt() {
    cout << "D" << id << endl;
} // Punkt.cpp
```

Feld / Array

```
#include "Punkt.h"
int main() {
    cout << "Start" << endl;
    {
        cout << "Block Start" << endl;
        Punkt menge[3];
        cout << "Block Ende" << endl;
    }
    cout << "Ende" << endl;
} // Haupt.cpp
```

Ausgabe: Start
Block Start
K1
K2
K3
Block Ende
D3
D2
D1
Ende

```
class ZweiPunkte {
private:
    Punkt p[2]; // ←
public:
    ZweiPunkte() { // ←
        cout << "K ZweiPunkte" << endl;
    }
    ~ZweiPunkte() {
        cout << "D ZweiPunkte" << endl;
    }
};

int main() {
    cout << "Start" << endl;
    {
        ZweiPunkte zp;
    }
    cout << "Ende" << endl;
    return 0;
}
```

Konstruktoren von *Klassenkomponenten* werden **vor** dem *Konstruktor der Klasse* aufgerufen.
Bei Destruktoren **umgekehrt!**

Ausgabe: Start
K1
K2
K ZweiPunkte
D ZweiPunkte
D2
D1
Ende

Regeln für die Anwendung für Konstruktoren und Destruktoren

- Allgemein**
Bei mehreren globalen Objekten oder mehreren lokalen Objekten innerhalb eines Blockes werden
 - die Konstruktoren **in der Reihenfolge der Datendefinitionen** und
 - die Destruktoren in **umgekehrter Reihenfolge** aufgerufen.
- Globale Objekte**
 - Konstruktor wird **zu Beginn des Programms** (vor main) aufgerufen;
 - Destruktor wird hinter der schließenden Klammer von main aufgerufen.
- Lokale Objekte**
 - Konstruktor wird **an der Definitionsstelle** des Objekts aufgerufen;
 - Destruktor wird beim **Verlassen des definierenden Blocks** aufgerufen.

Regeln für die Anwendung für Konstruktoren und Destruktoren

4. Dynamische Objekte

- Konstruktor wird bei `new` aufgerufen;
- Destruktor wird bei `delete` für zugehörigen Zeiger aufgerufen.

5. Objekt mit Klassenkomponenten

- Konstruktor der Komponenten wird **vor dem der umfassenden Klasse** aufgerufen;
- am Ende der Lebensdauer werden Destruktoren **in umgekehrter Reihenfolge** aufgerufen.

6. Feld von Objekten

- Konstruktor wird bei Datendefinition für jedes Element **beginnend mit Index 0** aufgerufen;
- am Ende der Lebensdauer werden Destruktoren in umgekehrter Reihenfolge aufgerufen.

Klassen-Schablonen / -Templates

Zur Erinnerung:

- Wir kennen schon **Funktionsschablonen**:

```
template<typename T>
void sort(unsigned int const size, T[] data);
```

- Damit lassen sich Datentypen als „Parameter“ an Funktionen übergeben.
→ führt zu allgemeineren Funktionen & besserer Wiederverwendbarkeit
→ **Das geht auch mit Klassen.**

Klassen-Schablonen / -Templates

Normale Klasse

```
class Punkt {
    double x, y;
public:
    Punkt(double x, double y);
    void setzeX(double w);
    void setzeY(double w);
    double leseX();
    double leseY();
};
```

Klassen-Schablone / -Template

```
template<typename T>
class Punkt {
    T x, y;
public:
    Punkt(T x, T y);
    void setzeX(T w);
    void setzeY(T w);
    T leseX();
    T leseY();
};
```

Klassen-Schablonen / -Templates

Bedeutung: Nachfolgende Klasse hat Datentyp `T` als Parameter

`T` kann als Typ für

- Attribute
- Konstruktor-/Methodenparameter
- Rückgabewerte
- lokale Variablen innerhalb von Methoden verwendet werden.

Klassen-Schablone / -Template

```
template<typename T>
class Punkt {
    T x, y;
public:
    Punkt(T x, T y);
    void setzeX(T w);
    void setzeY(T w);
    T leseX();
    T leseY();
};
```



```

template<typename T>
class Punkt {
    T x, y;
public:
    Punkt(T v, T w) : x(v), y(w) {}
    void setzeX(T w) { x = w; }
    void setzeY(T w) { y = w; }
    T leseX() { return x; }
    T leseY();
};

```

← Implementierung in der
Schablonendefinition –
wie bei Klassen

```

template<typename T>
T Punkt<T>::leseY() {
    return y;
}

```

← Implementierung
außerhalb der
Schablonendefinition

Verwendung

```

Punkt<int> p1(0,0);
p1.setzeX(13);
Punkt<double> p2(-0.1, 231.1);
Punkt<int> *ptr = new Punkt<int>(23, 19);
delete ptr;

```

Klassen-Schablonen / -Templates

- Genau wie Funktionsschablonen können auch Klassenschablonen **mehr als einen Typparameter** haben.
- Statt `template<typename T>...` findet man manchmal noch die äquivalente, alte Schreibweise `template<class T>...`
- Schablonen sind besonders nützlich für Datenstrukturen, die **beliebige Typen speichern** sollen → nächstes Kapitel
- Bei der Verwendung einer Klassenschablone erzeugt der Compiler **automatisch** die konkrete Klasse.
 - Dafür muss der Compiler **Zugriff auf die komplette Definition** haben.
 - Implementierung **komplett im Header**, keine Trennung in .h- und .cpp-Dateien!