



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



Kapitel 10: Klassen



Inhalt

- Einführung
- Konstruktoren / Destruktoren
- Kopierkonstruktor
- Selbstreferenz
- Überladen von Operatoren
- ...

Kapitel 10: Klassen



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;
}
```

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

Kapitel 10: Klassen



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();
};
    
```

```

static int cnt = 1;
Punkt::Punkt() : id(cnt++) {
    cout << "K" << id << endl;
}
Punkt::~Punkt() {
    cout << "D" << id << endl;
}
    
```

„Hack!“
Nur für
Demoszwecke!

Feld / Array

Punkt.h Punkt.cpp

```

int main() {
    cout << "Start" << endl;
    {
        cout << "Block Start" << endl;
        Punkt menge[3];
        cout << "Block Ende" << endl;
    }
    cout << "Ende" << endl;
    return 0;
}
    
```

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

Regeln für die Anwendung für Konstruktoren und Destruktoren

1. **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.
2. **Globale Objekte**
 - Konstruktor wird zu Beginn der Lebensdauer (vor main) aufgerufen;
 - Destruktor wird hinter der schließenden Klammer von main aufgerufen.
3. **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.

Kopierkonstruktor (copy constructor)

```

class Punkt {
private:
    double x, y;
public:
    Punkt(double ax, double bx);
    Punkt(const Punkt& p);
    ~Punkt();
};
    
```

Kann wie eine Zuweisung interpretiert werden!

Kopierkonstruktor

```

Punkt::Punkt(const Punkt& p) {
    x = p.x; y = p.y;
}
    
```

Entstehendes Objekt wird mit einem bestehenden Objekt initialisiert!

alternativ:

```

Punkt::Punkt(const Punkt& p) : x(p.x), y(p.y) { }
    
```

wirkt wie Zuweisung; geht nur bei Klasselementen

Kopierkonstruktor (copy constructor)

Bauplan:

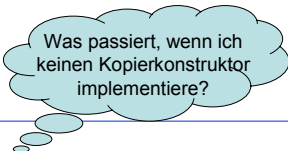
ObjektTyp (const ObjektTyp & bezeichner);

→ Kopierkonstruktor liefert / soll liefern byteweises Speicherabbild des Objektes

Wird **automatisch** aufgerufen, wenn:

1. ein neues Objekt erzeugt und mit einem bestehenden initialisiert wird;
2. ein Objekt per Wertübergabe an eine Funktion gereicht wird;
3. ein Objekt mit `return` als Wert zurückgegeben wird.

```
Punkt a(1.2, 3.4); // Neu
Punkt b(a);      // Kopie
Punkt c = b;     // Kopie
b = a;          // Zuweisung!
```



Kopierkonstruktor (copy constructor)

Wird für eine Klasse **kein** Kopierkonstruktor implementiert, dann erzeugt ihn der Compiler **automatisch!**

Achtung!

Es wird dann ein **byteweises Speicherabbild** des Objektes geliefert!

⇒ „flache Kopie“ (engl. *shallow copy*)

Problem:

- Konstruktor fordert dynamischen Speicher an → nur Kopie des Zeigers
- Konstruktor öffnet exklusive Datei (o.a. Resource) → nicht teilbar! Crash!

⇒ dann „tiefe Kopie“ (engl. *deep copy*) nötig!
 ⇒ man **muß** Kopierkonstruktor (und Destruktor) implementieren!

Klassendefinition

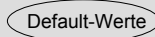
```
class CZeit {
private:
    int mStd, mMin, mSek;
public:
    CZeit(); // Konstruktor
    CZeit(int std, int min = 0, int sek = 0); // Konstruktor
    CZeit(const CZeit& aZeit); // Kopierkonstruktor

    void Anzeigen();

    int Std();
    int Min();
    int Sek();

    static CZeit Jetzt(); // statische Klassenfunktion
};

CZeit addiere(CZeit z1, CZeit z2); // globale Funktion
```



Konstruktoren

```
CZeit::CZeit() : mStd(0), mMin(0), mSek(0) { }

CZeit::CZeit(int aStd, int aMin, int aSek) :
    mStd(aStd), mMin(aMin), mSek(aSek) {
    assert(mStd >= 0 && mStd < 24);
    assert(mMin >= 0 && mMin < 60);
    assert(mSek >= 0 && mSek < 60);
}

CZeit::CZeit(const CZeit& aZeit) :
    mStd(aZeit.mStd), mMin(aZeit.mMin), mSek(aZeit.mSek) { }
```

statische Elementfunktion

```
CZeit CZeit::Jetzt() {
    time_t jetzt = time(0);
    struct tm *hms = localtime(&jetzt);
    CZeit z(hms->tm_hour, hms->tm_min, hms->tm_sec);
    return z;
}
```

Kapitel 10: Klassen

```
int CZeit::Std() { return mStd; }
int CZeit::Min() { return mMin; }
int CZeit::Sek() { return mSek; }
void CZeit::Anzeigen() {
    cout << mStd << ':' << mMin << ':' << mSek << endl;
}
```

```
CZeit addiere(CZeit z1, CZeit z2) {
    CZeit zeit;
    zeit.mStd = z1.mStd + z2.mStd;
    // usw
}
```

ACHTUNG !
Externer Zugriff auf private
Daten! Zugriff gesperrt!
⇒ Funktioniert so nicht!

```
CZeit addiere(CZeit z1, CZeit z2) {
    int std = z1.Std() + z2.Std();
    int min = z1.Min() + z2.Min();
    int sek = z1.Sek() + z2.Sek();
    CZeit zeit(std, min, sek);
    return zeit;
}
```

Hier muss noch dafür
gesorgt werden, dass der
Konstruktor keine Assertion
„wirft!“

Kapitel 10: Klassen

„Verschönerung“ der Zeitanzeige

bisher:
CZeit z(12,5,34); z.Anzeigen();
liefert Ausgabe: 12:5:34

```
void CZeit::Anzeigen() {
    cout << mStd << ':' << mMin << ':' << mSek << endl;
}
```

wir wollen haben: 12:05:34

```
void CZeit::Anzeigen() {
    if (mStd < 10) cout << "0";
    cout << mStd << ":";
    if (mMin < 10) cout << "0";
    cout << mMin << ":";
    if (mSek < 10) cout << "0";
    cout << mSek << endl;
}
```

... liefert 12:05:34 ☑

Kapitel 10: Klassen

Testprogramm

```
#include <iostream>
#include "CZeit.h"

using namespace std;

int main(){
    CZeit z(CZeit::Jetzt());
    z.Anzeigen();
    CZeit a(1,0,0);
    CZeit sum;
    sum = addiere(z, a);
    sum.Anzeigen();
    return 0;
}
```

Ausgabe (z.B.):

14:45:21

15:45:21

schöner wäre ja: sum = z + a;

Überladen von Operatoren!

Kapitel 10: Klassen

Normalisierung

```
class CZeit {
private:
    int mStd, mMin, mSek;
    void normalize();
public:
    ...
};
```

Private Hilfsfunktion:
kann nur **innerhalb** der
Klassenimplementierung
aufgerufen werden!

```
void CZeit::normalize() {
    mMin += mSek / 60;
    mSek %= 60;
    mStd += mMin / 60;
    mMin %= 60;
    mStd %= 24;
}
```

setzt beliebige nichtnegative
Werte auf „normale“ Werte:
 $0 \leq mStd < 24$
 $0 \leq mMin < 60$
 $0 \leq mSek < 60$



Überladen von Operatoren

```
bool operator== (const CZeit& aZeit) {
    if (aZeit.mStd != Std()) return false;
    if (aZeit.mMin != Min()) return false;
    if (aZeit.mSek != Sek()) return false;
    return true;
}
```

Test auf
Gleichheit

```
CZeit& operator= (const CZeit& aZeit) {
    mStd = aZeit.mStd;
    mMin = aZeit.mMin;
    mSek = aZeit.mSek;
    return *this;
}
```

Zuweisung



Wenn für eine Klasse der Zuweisungsoperator **nicht** überschrieben wird, dann macht das der Compiler **automatisch!**

Vorsicht!

Speicher des Objektes wird **byteweise** überschrieben!

Problem:

z.B. wenn Objekt dynamischen Speicher verwendet
⇒ gleiche Problematik wie beim Kopierkonstruktor

Merke:

Wenn die Implementierung eines **Kopierkonstruktors** nötig ist, dann höchstwahrscheinlich auch **Destruktor** und überschriebene **Zuweisung!**



Unterschied zwischen Kopierkonstruktor und Zuweisung

Kopierkonstruktor:

Initialisierung einer **neu** deklarierten Variable von **existierender** Variable

Zuweisung:

- wirkt zwar wie Kopierkonstruktor (flache Kopie bzw. tiefe Kopie), überschreibt jedoch Speicher der **existierenden** Variable mit dem Speicher der zuweisenden, **existierenden** Variable
- zusätzlich ggf. Aufräumen: Freigabe dynamischer Speicher!
- außerdem: Rückgabe einer Referenz auf sich selbst



Überladen von Operatoren

```
CZeit& CZeit::operator+= (const CZeit& aZeit) {
    mStd += aZeit.mStd;
    mMin += aZeit.mMin;
    mSek += aZeit.mSek;
    normalize();
    return *this;
}
```

Addition mit
Zuweisung

```
CZeit CZeit::operator+ (const CZeit& aZeit) {
    CZeit z(*this);
    z += aZeit; // Normalisierung via +=
    return z;
}
```

Addition



Test

```
int main() {

    CZeit x(0,0,86399);           // 23:59:59
    x.Anzeigen();
    x.addSek(1);                  // 00:00:00
    x.Anzeigen();
    x.addStd(12).addMin(13).addSek(14).Anzeigen(); // 12:13:14

    CZeit z1 = CZeit::Jetzt();   // statische Elementfunktion
    CZeit z2 = z1;               // Zuweisung, nutzt aber Kopierkonstruktor!
    z1.Anzeigen();
    if (z1 == z2)                // Operator == überladen
        cout << "Gleich!" << endl;

    CZeit a(1), b(0,1), c(0,0,1); // Nutzung der Defaults
    z1 += a;                      // Operator += überladen
    z1.Anzeigen();
}
```

Fortsetzung auf nächster Folie ... →



Fortsetzung ...

```
CZeit s;                          // Konstruktor für 00:00:00
s.Anzeigen();
s += a + b + c;                    // Operatoren += und + überladen
s.Anzeigen();

CZeit t;
t = a + b + c;                      // Operatoren = und + überladen;
// Zuweisung nutzt überladenes =
t.Anzeigen();
}
```

