

## Kurseinheit 2: Klassen 1

1. Einführung Objektorientierung
2. Klassen - Schnelleinstieg
3. Instanziierung von Objekten

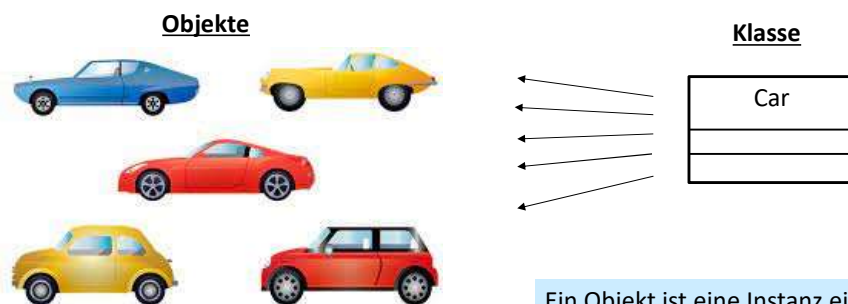
## 1. Einführung in Objektorientierung

### Objekte und Klassen

Was sind **Objekte**?

- Lebewesen und Pflanzen (und Pilze)
- Dinge, die man anfassen kann (Tisch, Fahrrad, Blatt Papier)
- Dateien
- Abstrakte Konzepte (Fakultät, Studiengang, Abteilung, Regierung, Verein)

Aus Softwaresicht (Abstraktion, Vereinfachung) sind sich einige **Objekte** sehr ähnlich. Diese haben den gleichen gewählten Aufbau (Bauplan). Dieser Bauplan wird als **Klasse** bezeichnet.



<https://www.meinauto.de/pics/wpimages/2018/01/Welches-Auto-passt-zu-mir.png>

## 1. Einführung in Objektorientierung

3

### Modellierung

Zur Modellierung der Software wird die Unified Modeling Language angewendet. Diese bietet unterschiedliche Diagramme an. Vorerst wichtig ist das Objekt- und das Klassendiagramm. Details siehe UML\_Kurzeinfuehrung im Moodlekurs.

**Objektdiagramm**

OG-SF-45:Car

Vollständig

**Klassendiagramm**

Car

Nur Objektname.  
Klasse ergibt sich  
implizit.

WOL-FI-43

Klassenname immer Singular!

:Car

Anonymes Objekt

Objekte haben immer unterstrichene Namen. Klassen nicht!

Elektrotechnik, Medizintechnik und Informatik
C++ - KE02: Klassen 1
Prof. Dr.-Ing. Daniel Fischer - Version 3.0.1

## 1. Einführung in Objektorientierung

4

### Klassen und Objekte in C und C++

C

**Klasse**

```
struct sCar
{
    uint32_t u32MaxSpeed;
};
```

C++

**Klasse**

```
class Car
{
public:
    void vSetMaxSpeed(uint32_t u32NewMaxSpeed);
    uint32_t u32GetMaxSpeed(void);

protected: // Important later

private:
    uint32_t u32MaxSpeed_;
};
```

**Objekt (Instanz)**

```
sCar sCar1{ 180U };
```

**Objekt (Instanz)**

```
Car Car2{};
```

Eine Klasse (class) erweitert den komplexen Datentyp struct aus C um **Methoden**. Die Elemente des C-Structs werden in der OO jetzt als **Attribute** bezeichnet.

Elektrotechnik, Medizintechnik und Informatik
C++ - KE02: Klassen 1
Prof. Dr.-Ing. Daniel Fischer - Version 3.0.1

## 1. Einführung in Objektorientierung

5

### Grundprinzipien der Objektorientierung

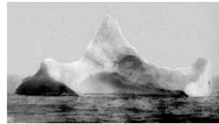
In der Objektorientierung gibt es **vier** Grundprinzipien:

- **Abstraktion**



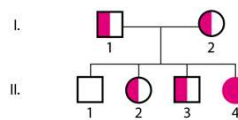
Welche Attribute des realen Objektes sind innerhalb des Softwaresystems relevant?  
MaxSpeed, Füllstand Schiebenwaschanlage, ...

- **Kapselung**



Welche Attribute und Methoden sollen von außerhalb überhaupt sichtbar sein?  
Metapher: Nur 1/7 eines Eisberges ist sichtbar oberhalb der Wasserlinie.

- **Vererbung**



Klassen können von anderen Klassen erben:  
Methoden und Attribute!  
Siehe später!

- **Polymorphismus**



Oft ist ein leicht geändertes Verhalten erwünscht: Statischer und dynamischer Polymorphismus. Siehe später!

## 2. Klassen - Schnelleinstieg

6

### Abstraktion und Kapselung

**Abstraktion:**

In diesem SW-System wäre nur die maximale Geschwindigkeit (u32MaxSpeed\_) relevant. Attribute erhalten ein „\_“ nachgestellt.

```
class Car
{
public:
    void vSetMaxSpeed(uint32_t u32NewMaxSpeed);
    uint32_t u32GetMaxSpeed(void);
protected: // Important ... later
private:
    uint32_t u32MaxSpeed_;
};
```

**Kapselung:**

Nur diese Methoden wären nach außen hin sichtbar!

Attribute sollten immer private sein.

```
Car Car2{};
//Car2.u32MaxSpeed_ = 130U; Error
Car2.vSetMaxSpeed(200U);
```

Statt class kann in C++ auch das Keyword struct verwendet werden. Dies wird auch in [Los19] so praktiziert. Unterschied: Wird kein Keyword public/protected/private anfänglich verwendet, so ist der **Default** bei **class private** und bei **struct public**.

## 2. Klassen - Schnelleinstieg

7

### Deklaration und Definition einer Klasse

Car.h

```
class Car
{
public:
    void vSetMaxSpeed(uint32_t u32NewMaxSpeed);
    uint32_t u32GetMaxSpeed(void);

protected: // Important ...

private:
    uint32_t u32MaxSpeed_;
```

Car.cpp

```
#include "Car.h "

static const uint32_t cu32MaxSpeed = 400U;

void Car::vSetMaxSpeed(uint32_t u32NewMaxSpeed)
{
    if (u32NewMaxSpeed < cu32MaxSpeed)
    {
        u32MaxSpeed_ = u32NewMaxSpeed;
    }
}

uint32_t Car::u32GetMaxSpeed(void)
{
    return u32MaxSpeed_;
}
```

Klassennamen und  
Scope-Operator nicht  
vergessen!

Deklaration und Definition (Implementierung) in getrennte Dateien.

## 2. Klassen - Schnelleinstieg

8

### Implizites Inlining

Car.h

```
class Car
{
public:
    void vSetMaxSpeed(uint32_t u32NewMaxSpeed)
    {
        if (u32NewMaxSpeed < cu32MaxSpeed)
        {
            u32MaxSpeed_ = u32NewMaxSpeed;
        }
    }
    uint32_t u32GetMaxSpeed(void)
    {
        return u32MaxSpeed_;
    }

protected: // Important ... later

private:
    uint32_t u32MaxSpeed_;
};
```

Kürzere Funktionen (LOCpro niedrig)  
werden oft schon in der h-Datei  
implementiert.  
Dies ist als implizites Inlining zu  
verstehen.

Ob die Funktion später tatsächlich  
inline ist, entscheidet der Compiler  
basierend auf LOCpro und  
Optimization Level.

#### Empfehlung **nur** für die Klausur:

Alle Funktionen inlinen – erspart  
Schreibarbeit, ebenso hilft dies bei  
der Übersichtlichkeit.

## 2. Klassen - Schnelleinstieg

9

### Klassenvariable und -methode

Car.h

Deklaration in h-Datei!

```
class Car
{
public:
    static uint32_t u32GetNumberOfInstances(void);
    // ...

private:
    static uint32_t u32NumberOfInstances_c;
    // ...
};
```

**static nur in h-Datei!**

Klassenmethode:  
Kann ohne Objekt immer aufgerufen werden.

Klassenvariable:  
Ist auch ohne Objekt immer verfügbar.

Car.cpp

Definition in cpp-Datei!

```
uint32_t Car::u32GetNumberOfInstances(void)
{
    return u32NumberOfInstances_c;
}

uint32_t Car::u32NumberOfInstances_c = 0U;
```

**static nicht in cpp-Datei!**

Klassenmethode kann nicht auf Attribute zugreifen.

## 2. Klassen - Schnelleinstieg

10

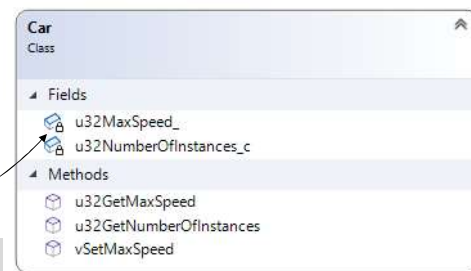
### Klassendiagramm in Class Designer von MSVS

Leider verfügt der Class Designer nicht über den vollen Umfang der UML-Funktionalität. Klassenvariablen und -methoden müssten unterstrichen sein. Durch „\_“ sind Klassenvariablen hier unterscheidbar. Sichtbarkeit mit Schloss dargestellt (privat)

```
class Car
{
public:
    void vSetMaxSpeed(uint32_t u32NewMaxSpeed);
    uint32_t u32GetMaxSpeed(void);
    static uint32_t u32GetNumberOfInstances(void);

protected:

private:
    uint32_t u32MaxSpeed_;
    static uint32_t u32NumberOfInstances_c;
};
```



Klassendiagramm kann automatisch generiert werden.

Class Designer mitinstallieren, bzw. mit MSVS Installer nachinstallieren. Projekt->Add->New Item->Utility->Class Diagramm. h-Datei rüberziehen.

## 2. Klassen - Schnelleinstieg

11

### this-Zeiger

Der Compiler ergänzt alle Objektmethoden um den this-Zeiger, welcher auf die Daten des Objektes zeigt. Datentyp des this-Zeigers wäre in diesem Beispiel Car\*.

```
uint32_t Car::u32GetMaxSpeed(void)
{
    return u32MaxSpeed_; // this->u32MaxSpeed_
}

uint32_t Car::u32GetNumberOfInstances(void)
{
    return u32NumberOfInstances_c;
}
```

```
Car Car2{};
Car2.vSetMaxSpeed(200U);
std::cout << sizeof(Car2) << std::endl;
```

Gibt vier zurück. Ein Car-Objekt enthält nur ein Attribut u32MaxSpeed\_

### Objektmethode

this-Zeiger wird reincompiliert. Immer erster Parameter.

### Klassenmethode:

Kein this-Zeiger. Daher kann hier nicht auf Attribute des Objektes zugegriffen werden.

Speicher	
0x00	0x000000C8 ist 200 <sub>10</sub>
0x00	Little Endianess
0x00	
0xC8	← this

## 2. Klassen - Schnelleinstieg

12

### const

Mittels const lässt sich sicherer Code schreiben.

- Objekte können – wie Variablen in C – auch const sein.

```
const Car Car2{};
```

- Referenzen können ebenso const sein. Bei Zeigern gibt es sogar zwei const!

```
const Car& rCar2 = Car2{};
```

```
uint32_t u32V = 73U;
uint32_t const * const pu32V = &u32V;
```

Wert kann nicht verändert werden.

Zeiger kann nicht verschoben werden.

- Objektmethoden können auch als const definiert werden.

```
uint32_t u32GetMaxSpeed() const
{
    return u32MaxSpeed_;
}
```

Auf die Attribute kann nur lesend zugegriffen werden.

### 3. Instanziierung von Objekten

13

#### Dynamische und statische Allokation

Objekte können auf zwei grundlegende Arten allokiert (instanziert) werden:

##### Statische Allokation – Methodenaufruf mit Punkt-Operator

```
Car Car2{};  
Car2.vSetMaxSpeed(200U);
```

Wo liegt Car2 im Speicher?

- Lokales Objekt oder Übergabeparameter: Stack Segment (SS)
- Globales, modulglobales oder static lokales Objekt: Data Segment (DS)

##### Dynamische Allokation – Methodenaufruf mit -> Operator

```
Car* pCar3 = new Car;  
pCar3->vSetMaxSpeed(100U);
```

Wo liegt Car3 im Speicher?

- Heap Segment (HS)

```
delete pCar3;
```

Speicher muss wieder freigegeben werden, ansonsten entstehen Memory Leaks. Ähnliche Vorgehensweise ist von malloc/free bekannt.

Auf public Attribute könnte dann auch entsprechend zugegriffen werden.

### 3. Instanziierung von Objekten

14

#### Unterschiede: Allokation und Target

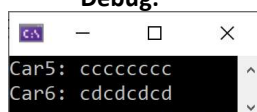
Bei der Instanziierung/Allokation ergeben sich Unterschiede bezüglich Debug/Release: Objekte im **Daten Segment** sind 0 initialisiert.

Bei Objekten im **Heap** und **Stack Segment** hängt dies vom Target (MSVS) ab:

- Debug: Lokales Objekt mit 0xCCCCCCCC – Objekt auf Heap mit 0xCDCDCDCD
- Release: Lokales Objekt mit 0 initialisiert bei leerem Default-Konstruktor – Objekt auf dem Heap zufällig

```
Car Car5; // Local  
std::cout << std::hex << "Car5: " << Car5.u32GetMaxSpeed() << std::endl;  
Car* pCar6 = new Car;  
std::cout << "Car6: " << pCar6->u32GetMaxSpeed() << std::hex << std::endl;  
delete pCar6;
```

Debug:



```
C:\>  
Car5: cccccccc  
Car6: cdcddcd
```

Release:



```
C:\>  
Car5: 0  
Car6: 8f00008f
```

Notwendigkeit einer standardisierten Initialisierung!

### 3. Instanziierung von Objekten

15

#### Konstruktor und Destruktor

Um eine einheitliche Initialisierung zu erreichen, können Konstruktoren verwendet werden. C++ stellt hier schon default-Varianten zur Verfügung. Eigene Konstruktoren überschreiben den Default-Konstruktor. Darin können die Attribute auf gestimmte Werte gesetzt werden.

```
class Car
{
private:
    uint32_t u32MaxSpeed_ = 130U; //C++11
};
```

äquivalent

```
class Car
{
public:
    Car() = default; //Default Constructor
    ~Car() = default; //Default Constructor

private:
    uint32_t u32MaxSpeed_ = 130U; //C++11
};
```

Es existieren hier Default-Konstruktor und ein Destruktor

Seit C++11 ist auch dies möglich.

Sobald eigene Implementierungen zur Verfügung gestellt werden, entfallen diese Default-Implementierungen. Konstruktor und Destruktor haben keine Rückgabewerte, da diese automatisch aufgerufen werden.

### 3. Instanziierung von Objekten

16

#### Konstruktor und Destruktor: Eigene Implementierungen

Eine eigene Implementierung überschreibt den bestehenden leeren Default-Konstruktor und den Default-Destruktor. Der leere Default-Konstruktor initialisiert alle Attribute mit 0.

```
class Car
{
public:
    Car(uint32_t u32NewMaxSpeed);
    ~Car();

private:
    uint32_t u32MaxSpeed_;
};
```

```
Car::Car(uint32_t u32NewMaxSpeed) :
    u32MaxSpeed_{ u32NewMaxSpeed }
{
    // Nothing to do
}
```

Initialisierungsliste

```
Car::~~Car()
{
    // Nothing to do
}
```

Statt einer Initialisierungsliste können die Attribute auch innerhalb des Konstruktors initialisiert werden. Manchmal wird sogar der gleiche Attributname verwendet. Unterschiedliche Sichtbarkeiten bei gleichnamigen Variablen (this-Zeiger).

```
Car::Car(uint32_t u32MaxSpeed_)
{
    this->u32MaxSpeed_ = u32MaxSpeed_;
}
```



### 3. Instanziierung von Objekten

17

#### Instanziierung bei mehreren Konstruktoren

```
class Car
{
public:
    Car(uint32_t u32NewMaxSpeed);
    Car(void);
    ~Car();

private:
    uint32_t u32MaxSpeed_;
};
```

```
//Constructor 1
Car::Car(uint32_t u32NewMaxSpeed) :
    u32MaxSpeed_{ u32NewMaxSpeed }
{
}
//Constructor 2
Car::Car(void) :
    u32MaxSpeed_{ 0U }
{
}
```

```
int main(void)
{
    Car Car1{ 130U }; → Instanziierung mit Constructor 1
    Car Car2(120U); → Instanziierung mit Constructor 1
    Car Car3{}; → Instanziierung mit Constructor 2
    Car Car4(); → Keine Instanziierung - Most Vexing Parse (nach Scott Meyers)
}
Brace yourself!
```

### 3. Instanziierung von Objekten

18

#### Allokation und Deallokation im Konstruktor/Destruktor

```
class Car
{
public:
    Car(uint32_t u32NewMaxSpeed);
    Car(void);
    ~Car();

private:
    uint32_t u32MaxSpeed_;
    char* pac_;
};
```

**RAII Resource Acquisition Is Initialization**  
Dabei wird die Belegung von Betriebsmitteln an den Konstruktoraufruf einer Variablen eines benutzerdefinierten Typs und die Freigabe der Betriebsmittel an dessen Destruktoraufruf gebunden.

Quelle: wikipedia.org.

```
Car::Car(uint32_t u32NewMaxSpeed) :
    u32MaxSpeed_{ u32NewMaxSpeed }
{
    pac_ = new char[100];
}
Car::Car(void) : u32MaxSpeed_{ 0U }
{
    pac_ = new char[100];
}
Car::~~Car()
{
    delete [] pac_; //Array of objects
}
```

```
int main(void)
{
    Car Car1{ 130U };
    return 0;
}
```

**Impliziter Aufruf:**  
Passender  
Konstruktor  
Destruktor

### 3. Instanziierung von Objekten

19

#### Kopierkonstruktor

```
class Car
{
public:
    Car(uint32_t u32NewMaxSpeed);
    Car(void);
    Car(const Car& rCar);
    ~Car();

private:
    uint32_t u32MaxSpeed_;
    char* pac_;
};
```

```
Car::Car(const Car& rCar)
{
    u32MaxSpeed_ = rCar.u32MaxSpeed_;
    pac_ = rCar.pac_;
}
```

Hinweis: Obwohl Attribute private sind, kann innerhalb einer Objektmethode der gleichen Klasse auf die privaten Elemente eines anderen Objektes der gleichen Klasse zugegriffen werden.

Kopierkonstruktor erhält eine Referenz auf ein Objekt der gleichen Klasse (rCar) und kopiert dessen Elemente: **const Car& rCar**

- Warum eine Referenz und kein Objekt als Übergabe? \_\_\_\_\_
- Warum kein Pointer statt einer Referenz? \_\_\_\_\_
- Warum eine konstante Referenz? \_\_\_\_\_

### 3. Instanziierung von Objekten

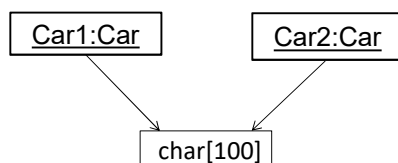
20

#### Kopierkonstruktor: Tiefe und flache Kopien

Der Kopierkonstruktor gibt es auch als Default-Variante. Hier werden die Attribute elementweise kopiert.

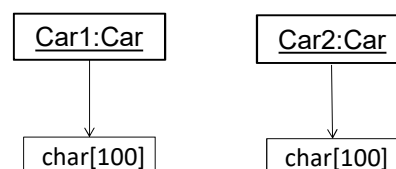
Verweist ein Element (Pointer, Referenz) auf ein anderes Objekt, so stellt sich die entscheidende Frage: **Will man eine flache oder eine tiefe Kopie?**

##### Flache Kopie



```
Car::Car(const Car& rCar)
{
    pac_ = rCar.pac_;
}
```

##### Tiefe Kopie



```
Car::Car(const Car& rCar)
{
    pac_ = new char[100];
    memcpy(pac_, rCar.pac_,
        100 * sizeof(char));
}
```

Car2 deleted, gibt Speicher char[100] im Destruktor wieder frei. Car1 zeigt dann auf **ungültigen** Speicher.

### 3. Instanziierung von Objekten

21

#### Kopierkonstruktor: Anwendung

```
class Car
{
public:
    Car(uint32_t u32NewMaxSpeed);
    Car(const Car& rCar);
    Car(void);
    ~Car();

private:
    uint32_t u32MaxSpeed_;
    char* pac_;
};
```

```
Car Car2(120U);
Car Car5(Car2);
```

Wenn in einer Deklaration eine Referenz angegeben wird, kann auch ein Objekt statt einer Referenz übergeben werden.

Der Default-Kopierkonstruktor erstellt nur flache Kopien.

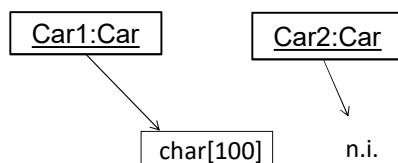
### 3. Instanziierung von Objekten

22

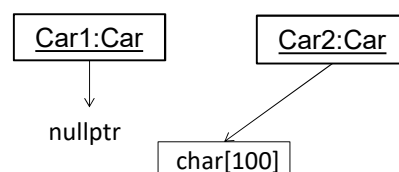
#### Move-Konstruktor

Ein Objekt kann auch seine dynamischen Objekte an ein anderes Objekt übergeben. Dafür ist der Move-Konstruktor gedacht. Oft sind diese dynamischen Objekte mit viel Aufwand (zeitlich, Ressourcen) erstellt worden, diese sollen dann nicht nochmals erstellt werden.

Car2 gerade in Konstruktion



Nach dem Aufruf des Move-Konstruktors



```
class Car
{
public:
    // ...
    Car(Car&& rCar) noexcept;
    // ...
};
```

```
Car::Car(Car&& rCar) noexcept
{
    u32MaxSpeed_ = rCar.u32MaxSpeed_;
    pac_ = rCar.pac_;
    rCar.pac_ = nullptr;
}
```

Aufruf: `Car Car6 = std::move(Car2);`

### 3. Instanziierung von Objekten

23

#### explicit

```
class Car
{
public:
    Car(uint32_t u32NewMaxSpeed);
    Car(const Car& rCar);
    Car(Car&& rCar) noexcept;
    Car(void);
    ~Car();

private:
    uint32_t u32MaxSpeed_;
    char* pac_;
};
```

Hier wird beim Aufruf von foo implizit ein Car Objekt angelegt.

```
{
    foo(1);
}

void foo(Car Car2)
{
}
```

Soll das implizite Erzeugen von Objekten verhindert werden, so ist **explicit** vor die Konstruktoren zu schreiben (Deklaration, nicht bei der Definition).

### 3. Instanziierung von Objekten

24

#### Tipps für den Anfang

Gerade beim Einstieg in C++ ist häufig unklar, wann welcher Konstruktor aufgerufen wird. Es ist daher zu empfehlen, dass anfänglich in alle Konstruktoren und in den Destruktor Konsolenausgaben erfolgen (am Funktionsanfang). Beispiele:

```
std::cout << "Constructor with uint32_t entered" << std::endl;

std::cout << "Empty Constructor of Car entered" << std::endl;

std::cout << "Copy Constructor of Car entered" << std::endl;

std::cout << "Move Constructor of Car entered" << std::endl;

std::cout << "Destructor of Car entered" << std::endl;
```

Mit bedingter Compilierung könnten die Ausgaben ein- und ausgeschaltet werden.

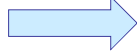
### 3. Instanziierung von Objekten

25

#### Empty class

```
class Empty {  
};
```

auto generated



`sizeof(Empty)` gibt 1 zurück.

Es wird im Speicher ein Dummy-Byte angelegt, damit eine gültige Objektadresse existiert.

```
class Empty  
{  
public:  
    Empty() = default;  
    Empty(const Empty&) = default;  
    Empty(Empty&&) noexcept = default;  
    ~Empty() = default;  
  
    // Copy-Assign und Move-Assign Operator  
    Empty& operator=(const Empty&) = default;  
    Empty& operator=(Empty&&) noexcept = default;  
};
```

Mit `default` wird dem Compiler mitgeteilt, dass die auto generated Version verwendet wird. Soll eine auto generated Methode/Operator gelöscht werden, so ist **delete** statt **default** zu verwenden.