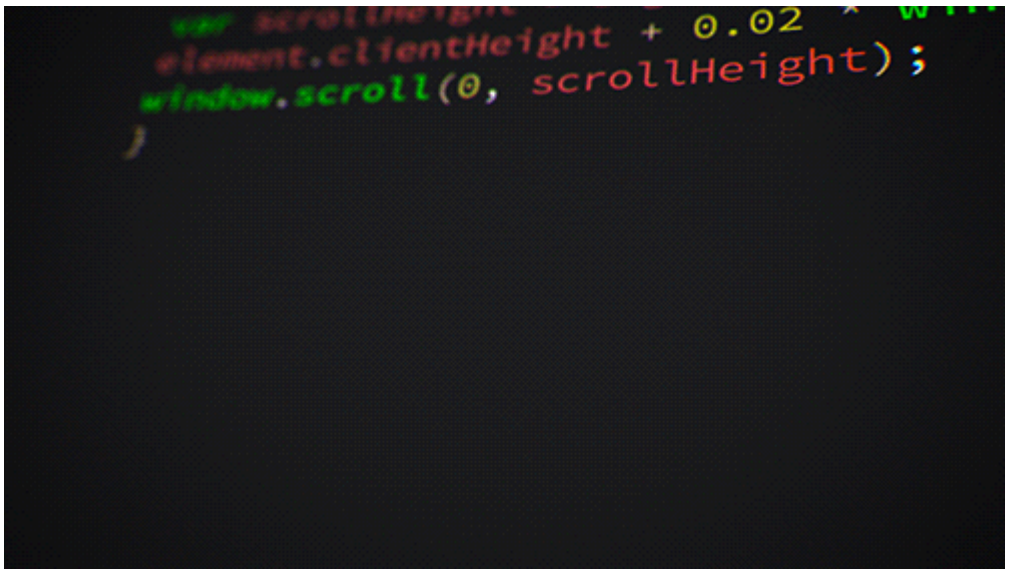
Objektorientierte Programmierung mit C++

Parameter	Kursinformationen
Veranstaltung:	`Prozedurale Programmierung / Einführung in die Informatik / Erhebung, Analyse und Visualisierung digitaler Daten`
Semester	`Wintersemester 2025/26`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Operatorenüberladung / Vererbung`
Link auf Repository:	https://github.com/TUBAF-IfI-LiaScript/VL_EAVD/blob/master/04_Funktionen.md
Autoren	Sebastian Zug & André Dietrich & Galina Rudolf

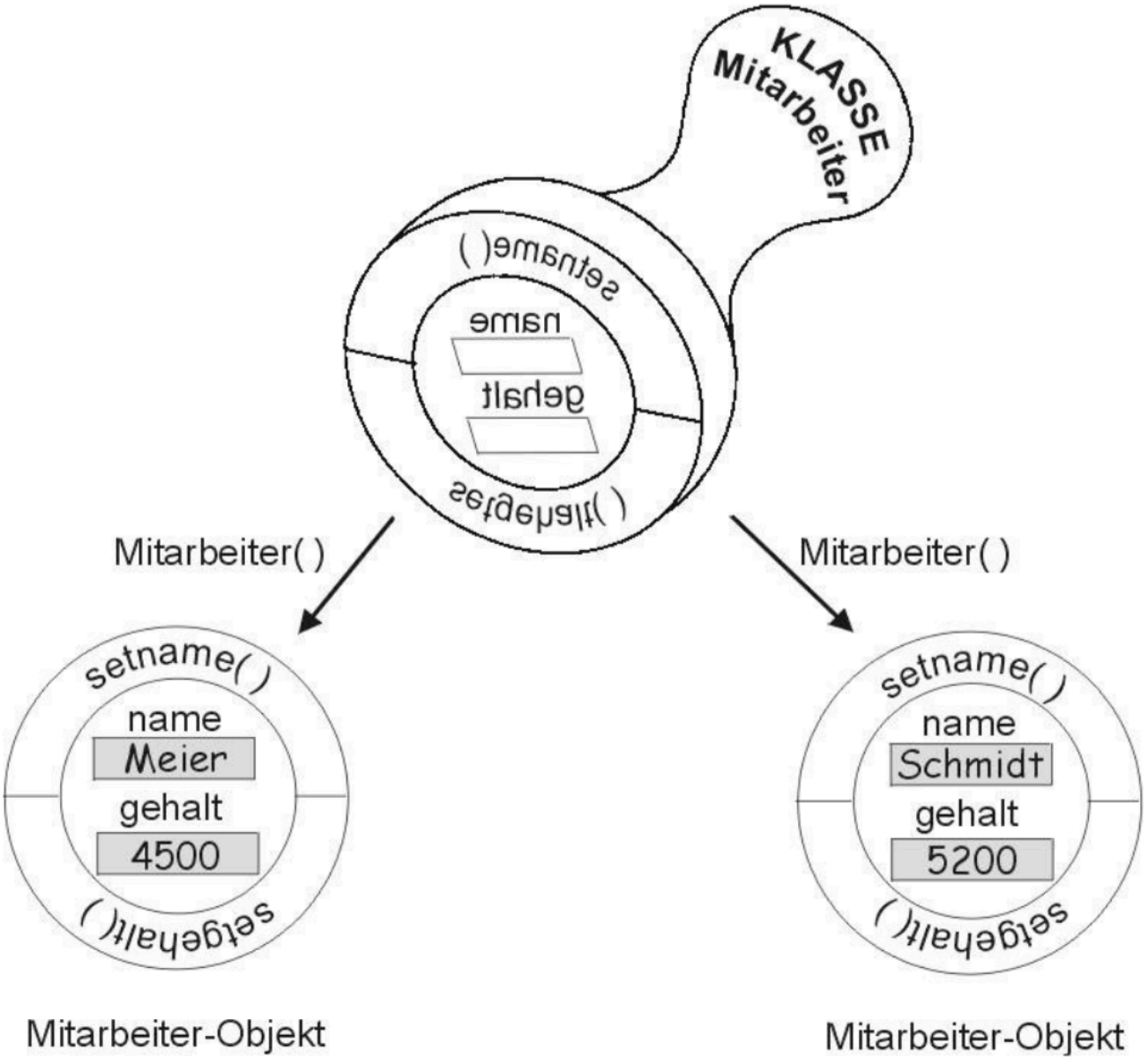


Fragen an die heutige Veranstaltung ...

- Was sind Operatoren?
- Warum werden eigene Operatoren für individuelle Klassen benötigt?
- Wann spricht man von Vererbung und warum wird sie angewendet?
- Welche Zugriffsattribute kennen Sie im Zusammenhang mit der Vererbung?

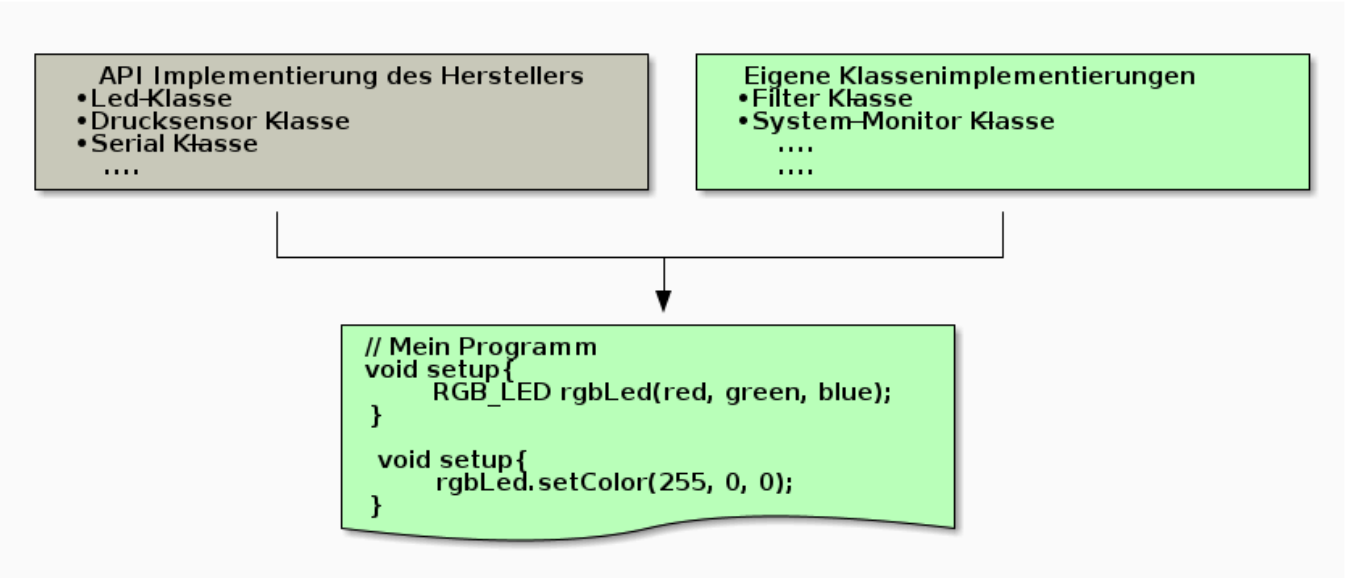
Rückblick

Unter einer Klasse (auch Objekttyp genannt) versteht man in der objektorientierten Programmierung ein abstraktes Modell bzw. einen Bauplan für eine Reihe von ähnlichen Objekten.



Von Binz - Own Creation, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=62707688

Und was bedeutet das angewandt auf unsere Vision mit dem Mikrocontroller Daten zu erheben?



Für die Implementierung einer Ausgabe auf dem Display des MXCHIP Boards nutzen wir die Klassenimplementierung der API.

Link

Operatorenüberladung

Motivation

Folgendes Beispiel illustriert den erreichten Status unserer C++ Implementierungen. Unsere Klasse `Student` besteht aus:

- 3 Membervariablen (Zeile 5-7)
- 2 Konstruktoren (Zeile 9-10)
- 1 Memberfunktion (Zeile 12)

Alle sind als `public` markiert.

example.cpp

```cpp
#include <iostream>

class Student {
  public:
    std::string name;
    int alter;
    std::string ort;

    Student(std::string n);
    Student(std::string n, int a, std::string o);

    void ausgabeMethode(std::ostream& os); // Deklaration der Methode
};

Student::Student(std::string n):
  name(n), alter(8), ort("Freiberg")
{}

Student::Student(std::string n, int a, std::string o):
  name(n), alter(a), ort(o)
{}

void Student::ausgabeMethode(std::ostream& os) {
  os << name << " " << ort << " " << alter << "\n";
}

int main()
{
  Student gustav = Student("Zeuner", 27, "Chemnitz");
  //Student gustav {"Zeuner", 27, "Chemnitz"};
  //Student gustav("Zeuner", 27, "Chemnitz");
  gustav.ausgabeMethode(std::cout);

  Student bernhard {"Cotta", 18, "Zillbach"};
  bernhard.ausgabeMethode(std::cout);

  Student nochmalBernhard {"Cotta", 18, "Zillbach"};
}
```

```
Zeuner Chemnitz 27
Cotta Zillbach 18
Zeuner Chemnitz 27
Cotta Zillbach 18
```

Aufgabe: Schreiben Sie

- eine Funktion `int vergleich(Student, Student)` und
- eine Methode `int Student::vergleich(Student)`,

die zwei Studenten miteinander vergleicht!

Was ist der konzeptionelle Unterschied zwischen beiden Implementierungen?

Frage: Was ist der Nachteil unserer Lösung?

Konzept

Das Überladen von Operatoren erlaubt die flexible klassenspezifische Nutzung von Arithmetischen- und Vergleichs-Symbolen wie `+`, `-`, `*`, `==`. Damit kann deren Bedeutung für selbstdefinierte Klassen mit einer neuen Bedeutung versehen werden. Ausnahmen bilden spezielle Operatoren, die nicht überladen werden dürfen (?: , :: , . , .* , typeid , sizeof und die Cast-Operatoren).

```
Matrix a, b;
Matrix c = a + b;     \\ hier wird mit dem Plus eine Matrixoperation ausgefü

String a, b;
String c = a + b;     \\ hier werden mit dem Plus zwei Strings konkateniert
```

Operatorüberladung ist Funktionsüberladung, wobei die Funktionen durch eine spezielle Namensgebung gekennzeichnet sind. Diese beginnen mit dem Schlüsselwort `operator`, das von dem Token für den jeweiligen Operator gefolgt wird.

```
class Matrix {
  public:
    Matrix operator+(Matrix zweiterOperand) { ... }
    Matrix operator/(Matrix zweiterOperand) { ... }
    Matrix operator*(Matrix zweiterOperand) { ... }
}

class String {
  public:
    String operator+(String zweiterString) { ... }
}
```

Operatoren können entweder als Methoden der Klasse oder als globale Funktionen überladen werden. Die Methodenbeispiele sind zuvor dargestellt, analoge Funktionen ergeben sich zu:

```
class Matrix {
  public:
    ...
}

Matrix operator+(Matrix ersterOperand, Matrix zweiterOperand) { ... }
Matrix operator/(Matrix ersterOperand, Matrix zweiterOperand) { ... }
Matrix operator*(Matrix ersterOperand, Matrix zweiterOperand) { ... }
```

> **Merke:** Funktion oder Methode - welche Version sollte wann zum Einsatz kommen? Einstellige Operatoren `++` sollten Sie als Methode, zweistellige Operatoren ohne Manipulation der Operanden als Funktion implementieren. Für zweistellige Operatoren, die einen der Operanden verändern (`+=`), sollte als Methode realisiert werden.

Als Beispiel betrachten wir eine Klasse Rechteck und implementieren zwei Operatorüberladungen:

- eine Vergleichsoperation
- eine Additionsoperation die `A = A + B` oder abgekürzt `A+=B`

implementiert.

Comparison.cpp

```cpp
#include <iostream>

class Rectangle {
  private:
    float width, height;
  public:
    Rectangle(int w, int h):
      width{w}, height{h}
    {}
    float area() {
      return width * height;
    }
    Rectangle operator+=(Rectangle offset) {
      float ratio = (offset.area() + this->area()) / this->area();
      this->width = ratio *  this->width;
      return *this;
    }
};

bool operator>(Rectangle a, Rectangle b) {
  if (a.area() > b.area())
    return 1;
  else
    return 0;
}

int main () {
  Rectangle rect_a(3,4);
  Rectangle rect_b(5,7);
  std::cout << "Vergleich: " << (rect_a > rect_b) << "\n";

  std::cout << "Fläche a : " << rect_a.area() << "\n";
  std::cout << "Fläche b : " << rect_b.area() << "\n";
  rect_a += rect_b;
  std::cout << "Summe    : " << rect_a.area();

  return 0;
}
```

```
Vergleich: 0
Fläche a : 12
Fläche b : 35
Summe    : 47Vergleich: 0
Fläche a : 12
Fläche b : 35
Summe    : 47
```

Merke: Üblicherweise werden die Operanden, welche lediglich betrachtet werden, bei der Operatorüberladung als Referenzen übergeben. Damit wird eine Kopie vermieden. In Kombination mit dem Schlüsselwort `const` kann dem Compiler angezeigt werden. dass keine Veränderung an den Daten vorgenommen wird. Sie müssen also nicht gespeichert werden.

<!--- TODO: Aufgabe: Überlegen Sie sich, in welchen Situationen ein pass by reference eventuell doch sinnvoll wäre. -->

```
bool operator>(const Rectangle& a, const Rectangle& b) {
  if (a.area() > b.area())
    return 1;
  else
    return 0;
}
```

Stellen wir die Abläufe nochmals grafisch dar Pythontutor

Anwendung

Im folgenden Beispiel wird der Vergleichsoperator `==` überladen. Dabei sehen wir den Abgleich des Namens und des Alters als ausreichend an.

Comparison.cpp

```cpp
#include <iostream>

class Student {
  public:
    std::string name;
    int alter;
    std::string ort;

    Student(std::string n);
    Student(std::string n, int a, std::string o);

    void ausgabeMethode(std::ostream& os); // Deklaration der Methode
    bool operator==(const Student&);
};

Student::Student(std::string n):
  name(n), alter(8), ort("Freiberg")
{}

Student::Student(std::string n, int a, std::string o):
  name(n), alter(a), ort(o)
{}

void Student::ausgabeMethode(std::ostream& os) {
  os << name << " " << ort << " " << alter << "\n";
}

bool Student::operator==(const Student& other) {
  if ((this->name == other.name) && (this->alter == other.alter))
    return true;
  else
    return false;
}

int main()
{
  Student gustav = Student("Zeuner", 27, "Chemnitz");
  gustav.ausgabeMethode(std::cout);

  Student bernhard {"Cotta", 18, "Zillbach"};
  bernhard.ausgabeMethode(std::cout);

  Student NochMalBernhard {"Cotta", 18, "Zillbach"};
  NochMalBernhard.ausgabeMethode(std::cout);

  if (bernhard == NochMalBernhard)
    std::cout << "Identische Studenten\n";
  else
```

```
49      std::cout << "Ungleiche Identitäten\n";
50  }
```

```
Zeuner Chemnitz 27
Cotta Zillbach 18
Cotta Zillbach 18
Identische Studenten
Zeuner Chemnitz 27
Cotta Zillbach 18
Cotta Zillbach 18
Identische Studenten
```

Eine besondere Form der Operatorüberladung ist der `<<`, mit dem die Ausgabe auf ein Streamobjekt realsiert werden kann.

streamOperator.cpp

```cpp
#include <iostream>

class Student {
  public:
    std::string name;
    int alter;
    std::string ort;

    Student(const Student&);
    Student(std::string n);
    Student(std::string n, int a, std::string o);

    void ausgabeMethode(std::ostream& os); // Deklaration der Methode

    bool operator==(const Student&);
};

Student::Student(std::string n, int a, std::string o):
  name{n}, alter{a}, ort{o}
{}

std::ostream& operator<<(std::ostream& os, const Student& student) {
  os << student.name << '/' << student.alter << '/' << student.ort;
  return os;
}

int main() {
  Student gustav = Student("Zeuner", 27, "Chemnitz");
  Student bernhard = Student( "Cotta", 18, "Zillbach");
  std::cout << gustav << "\n";
}
```

```
Zeuner/27/Chemnitz
Zeuner/27/Chemnitz
```

Beachte: Innerhalb des `<<` Operanden wird normalerweise kein finales `"\n"` angehangen, sondern die Kontrolle über Zeilenumbrüche wird den Nutzern überlassen.

Eine umfangreiche Diskussion zur Operatorüberladung finden Sie unter https://www.c-plusplus.net/forum/topic/232010/%C3%BCberladung-von-operatoren-in-c-teil-1/2

Ebenso eine ausführliche Auskunft über Operatoren und ihre Eigenheiten: https://en.cppreference.com/w/cpp/language/operators.html

Vererbung

MultipleTypesOfPeople.cpp

```cpp
#include <iostream>

class Student {
  public:
    std::string name;
    std::string ort;
    std::string studiengang;

    Student(std::string n, std::string o, std::string sg):
      name{n}, ort{o}, studiengang{sg}
    {}
    void printCertificate(std::ostream& os) {
      os << "Studentendatensatz: " << name << " " << ort << " " <<
        studiengang << "\n";
    }
};

int main() {
  Student gustav = Student("Zeuner", "Chemnitz", "Mathematik");
  gustav.printCertificate(std::cout);

  // Professor winkler = Professor("Winkler", "Freiberg");
  // winkler.printCertificate(std::cout);
}
```

```
Studentendatensatz: Zeuner Chemnitz Mathematik
Studentendatensatz: Zeuner Chemnitz Mathematik
```

Aufgabe: Implementieren Sie eine neue Klasse `Professor`, die aber auf die Membervariable `Studiengang` verzichtet, aber eine neue Variable `Fakultät` einführt.

Motivation

Merke: Eine unserer Hauptmotivationen bei der "ordentlichen" Entwicklung von Code ist die Vermeidung von Codedopplungen!

In unserem Code entstehen Dopplungen, weil bestimmte Variablen oder Memberfunktionen usw. mehrfach für individuelle Klassen Implementiert werden. Dies wäre für viele Szenarien analog der Fall:

Basisklasse	abgeleitete Klassen	Gemeinsamkeiten
Fahrzeug	Flugzeug, Boot, Automobil	Position, Geschwindigkeit, Zulassungsnummer, Führerscheinpflicht
Datei	Foto, Textdokument, Datenbankauszug	Dateiname, Dateigröße, Speicherort
Nachricht	Email, SMS, Chatmessage	Adressat, Inhalt, Datum der Versendung

Merke: Die *Vererbung* ermöglicht die Erstellung neuer Klassen, die ein in existierenden Klassen definiertes Verhalten wieder verwenden, erweitern und ändern. Die Klasse, deren Member vererbt werden, wird Basisklasse genannt, die erbende Klasse als abgeleitete Klasse bezeichnet.

Implementierung in C++

In C++ werden Vererbungsmechanismen folgendermaßen abgebildet:

```
class Fahrzeug {
  public:
    int aktuellePosition[2];    // lat, long Position auf der Erde
    std::string zulassungsnummer;
    bool fuehrerscheinpflichtig
    ...
};

class Flugzeug: public Fahrzeug {
  public:
    int flughoehe;
    void fliegen();
    ...
};

class Boot: public Fahrzeug {
  public:
    void schwimmen();
    ...
};
```

Die generellere Klasse `Fahrzeug` liefert einen Bauplan für die spezifischeren, die die Vorgaben weiter ergänzen. Folglich müssen wir uns die Frage stellen, welche Daten oder Funktionalität übergreifend abgebildet werden soll und welche individuell realisiert werden sollen.

Dabei können ganze Ketten von Vererbungen entstehen, wenn aus einem sehr allgemeinen Objekt über mehrere Stufen ein spezifischeres Set von Membern umgesetzt wird.

```
class Fahrzeug {
  public:
    int aktuellePosition[2];    // lat, long Position auf der Erde
    std::string zulassungsnummer;
    bool fuehrerscheinpflichtig
    ...
};

class Automobil: public Fahrzeug {
  public:
    void fahren();
    int zahlderRaeder;
    int sitze;
    ...
};

class Hybrid: public Automobil {
  public:
    void fahreElektrisch();
    ...
};
```

Was bedeutet das für unsere Implementierung von Studenten und Professoren?

Inheritance.cpp

```cpp
#include <iostream>

class Student {
  public:
    std::string name;
    std::string ort;
    std::string studiengang;

    Student(std::string n, std::string o, std::string sg):
      name{n}, ort{o}, studiengang{sg}
    {};
    void printCertificate(std::ostream& os) {
      os << "Studentendatensatz: " << name << " " << ort << " " <<
        studiengang << "\n";
    }
};

int main()
{
  Student gustav = Student("Zeuner", "Chemnitz", "Mathematik");
  gustav.printCertificate(std::cout);

  //Professor winkler = Professor("Winkler", "Freiberg");
  //winkler.printCertificate(std::cout);
}
```

```
Studentendatensatz: Zeuner Chemnitz Mathematik
Studentendatensatz: Zeuner Chemnitz Mathematik
```

Ein weiteres Beispiel greift den Klassiker der Einführung objektorientierter Programmierung auf, den Kanon der Haustiere 🙂 Das Beispiel zeigt die Initialisierung der Membervariablen :

- der Basisklasse beim Aufruf des Konstruktors der erbenden Klasse
- der Member der erbenden Klasse wie gewohnt

Animals.cpp

```cpp
#include <iostream>

// Vererbende Klasse Animal
class Animal {
  public:
    Animal():
      name{"Animal"}, weight{0.0}
    {}
    Animal(std::string _name, double _weight):
      name{_name}, weight{_weight}
    {};
    void sleep () {
      std::cout << name << " is sleeping!\n";
    }

    std::string name;
    double weight;
};

// Erbende Klasse Dog - Dog übernimmt die Methoden und Attribute von
class Dog: public Animal {
public:
    Dog(std::string name, double weight, int id):
      Animal(name, weight), id{id}
    {};

    // Dog spezifische Methoden: bark() und top_speed()
    void bark() {
      std::cout << "woof woof\n";
    }
    double top_speed() {
      if (weight < 40) return 15.5;
      else if (weight < 90) return 17.0;
      else return 16.2;
    }

    int id;
};

int main() {
    Dog dog = Dog("Rufus", 50.0, 2342);
    dog.sleep();
    dog.bark();
    std::cout << dog.top_speed() << "\n";
}
```

```
Rufus is sleeping!
woof woof
17
Rufus is sleeping!
woof woof
17
```

Vererbungsattribute

Die Zugriffsattribute `public` und `private` kennen Sie bereits. Damit können wir Elemente unserer Implementierung vor dem Zugriff von außen Schützen.

Wiederholungsaufgabe: Verhindern Sie, dass

- die Einträge von `id` im Nachhinein geändert werden können und
- das diese außerhalb der Klasse sichtbar sind.

Welche zusätzlichen Methoden benötigen Sie dann?

Animals.cpp

```cpp
#include <iostream>

class Animal {
public:
  Animal(std::string name, int id):
    name{name}, id{id}
  {}

  std::string name;
  int id;
};

int main() {
  Animal fish = Animal("Nemo", 234242);
  fish.id = 12345;
  std::cout << fish.id << "\n";
}
```

```
12345
12345
```

Wie wirkt sich das Ganze aber auf die Vererbung aus? Hierbei muss neben dem individuellen Zugriffsattribut auch der Status der Vererbung beachtet werden. Damit ergibt sich dann folgendes Bild:

```
+
```

```
class A {
public:
  int x;
protected:
  int y;
private:
  int z;
};

class B : public A {
  // x is public
  // y is protected
  // z is not accessible from B
};

class C : protected A {
  // x is protected
  // y is protected
  // z is not accessible from C
};

class D : private A {   // 'private' is default for classes
  // x is private
  // y is private
  // z is not accessible from D
};
```

Das Zugriffsattribut `protected` spielt nur bei der Vererbung eine Rolle. Innerhalb einer Klasse ist `protected` gleichbedeuted mit `private`. In der Basisklasse ist also ein Member geschützt und nicht von außen zugreifbar. Bei der Vererbung wird der Unterschied zwischen `private` und `protected` deutlich: Während `private` Member in erbenden Klassen nicht direkt verfügbar sind, kann auf die als `protected` deklariert zugegriffen werden.

Entsprechend muss man auch die Vererbungskonstellation berücksichtigen, wenn man festlegen möchte ob ein Member gar nicht (`private`), immer (`public`) oder nur im Vererbungsverlauf verfügbar sein (`protected`) soll.

Überschreiben von Methoden der Basisklasse (Polymorphie)

Die grundsätzlicher Idee bezieht sich auf die Implementierung "eigener" Methoden gleicher Signatur in den abgeleiteten Klassen. Diese implementieren dann das spezifische Verhalten der jeweiligen Objekte.

Inheritance.cpp

```cpp
#include <iostream>

class Person {
  public:
    std::string name;
    std::string ort;

    Person(std::string n, std::string o):
      name{n}, ort{o}
    {}
    void printData(std::ostream& os) {
      os << "Datensatz: "  << name << " " << ort << "\n";
    }
};

class Student: public Person {
  public:
    std::string studiengang;

    Student(std::string n, std::string o, std::string sg):
      Person(n, o), studiengang{sg}
    {}
    void printData(std::ostream& os) {
      os << "Student Datensatz: "  << name << " " << ort << "\n";
    }
};

class Professor: public Person {
  public:
    int id;

    Professor(std::string n, std::string o, int id):
      Person(n, o), id{id}
    {}
    void printData(std::ostream& os) {
      os << "Prof. Datensatz: "  << name << " " << ort << "\n";
    }
};

int main()
{
  Student *gustav = new Student("Zeuner", "Chemnitz", "Mathematik");
  gustav->printData(std::cout);
  delete gustav;

  Professor *winkler = new Professor("Winkler", "Freiberg", 234234);
  winkler->printData(std::cout);
  delete winkler;
```

```
49 }
```

```
Student Datensatz: Zeuner Chemnitz
Prof. Datensatz: Winkler Freiberg
Student Datensatz: Zeuner Chemnitz
Prof. Datensatz: Winkler Freiberg
```

Entwerfen Sie eine Klasse, die das Verhalten einer Ampel mit den notwendigen Zuständen modelliert. Welche Methoden sollten zusätzlich in die Klasse aufgenommen werden?

Simulation time: 00:31.843 (154%)

TrafficLight.cpp

```
class Ampel {
private:
    int redPin, yellowPin, greenPin;
    int state = 0;

public:
    Ampel(int red, int yellow, int green):
      redPin{red}, yellowPin{yellow}, greenPin{green}
    {
      pinMode(red, OUTPUT);
      pinMode(yellow, OUTPUT);
      pinMode(green, OUTPUT);
    };
    void activateRed() {
      digitalWrite(redPin, HIGH);
    }
    void startOnePeriod(int waitms) {
      digitalWrite(redPin, HIGH);
      delay(waitms);
      digitalWrite(yellowPin, HIGH);
      delay(waitms);
      digitalWrite(redPin, LOW);
      digitalWrite(yellowPin, LOW);
      digitalWrite(greenPin, HIGH);
    }
};

void setup() {
  Ampel trafficLight = Ampel(13, 12, 11);
  trafficLight.activateRed();
  trafficLight.startOnePeriod(1000);
}

void loop() {
  delay(100);
}
```

```
Sketch uses 1016 bytes (3%) of program storage space. Maximum is 32256
bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes
for local variables. Maximum is 2048 bytes.


Sketch uses 1016 bytes (3%) of program storage space. Maximum is 32256
bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes
for local variables. Maximum is 2048 bytes.
```

Quiz

Operatorenüberladung

Konzept

Welches Schlüsselwort wird bei der Operatorüberladung verwendet?

Für welche Operatoren sollten die Methoden einer Klasse und für welche die globalen Funktionen bevorzugt zum Einsatz kommen?

Methode	Funktion	
◯	◯	++
◯	◯	+
◯	◯	*
◯	◯	%
◯	◯	--
◯	◯	+=

Wie lautet die Ausgabe dieses Programms? (Hinweise können über das Feld mit der Glühbirne angezeigt werden.)

```
#include<iostream>

class Vektor {
  public:
    int x = 0, y = 0;

    Vektor(int x, int y);

    Vektor operator+(const Vektor& vek_tmp);

    void printVektor() {
      std::cout << x << ", " << y << "\n";
    }
};

Vektor::Vektor(int x, int y):
  x(x), y(y)
{}

Vektor Vektor::operator+(const Vektor& vek_tmp) {
```

```
  Vektor vek_loe(0, 0);
  vek_loe.x = this->x + vek_tmp.x;
  vek_loe.y = this->y + vek_tmp.y;
  return vek_loe;
}

int main() {
  Vektor v1(4, 7), v2(10, 9);
  Vektor v3 = v1 + v2;
  v3.printVektor();
}
```

Anwendung

Wie lautet die Ausgabe dieses Programms?

```
#include <iostream>

class Ort {
  public:
    std::string name;
    std::string bundesland;
    int einwohner;

    Ort(const Ort&);
    Ort(std::string n);
    Ort(std::string n, std::string b, int e);
};

Ort::Ort(std::string n, std::string b, int e):
  name{n}, bundesland{b}, einwohner{e}
{}

std::ostream& operator<<(std::ostream& os, const Ort& ort)
{
  os << ort.name << ", " << ort.bundesland << ", " << ort.einwohner;
  return os;
}

int main()
{
  Ort Freiberg = Ort("Freiberg", "Sachsen" , 41823);
  std::cout << Freiberg << "\n";
}
```

Vererbung

Erbt die erbende Klasse immer alle Attribute und Methoden der Basisklasse?

- ◯ Ja
- ◯ Nein

Implementierung in C++

Wodurch muss `[_____]` ersetzt werden um eine Klasse `Flugzeug` zu definieren, die von der Klasse `Fahrzeug` erbt?

```
#include <iostream>

class Fahrzeug {
  public:
    int aktuellePosition[2];
    std::string Zulassungsnummer;
    bool Fuehrerscheinpflichtig;
};

[_____]{
  public:
    int Flughoehe;
    void fliegen();
};
```

Kann eine abgeleitete Klasse als Basis-Klasse für eine weitere Klasse verwendet werden?

- ◯ Ja
- ◯ Nein

Wie lautet die Ausgabe dieses Programms?

```
#include <iostream>

class Fahrzeug {
  public:
    Fahrzeug():
      name{"Fahrzeug"}
    {}
    Fahrzeug(std::string _name):
      name{_name}
    {}
    void defekt() {
      std::cout << name << " muss in die Werkstatt.\n";
    }

    std::string name;
};

class Auto: public Fahrzeug {
  public:
    Auto(std::string name, int ps):
      Fahrzeug(name), ps{ps}
    {};

    int ps = 0;
};

int main() {
  Auto auto1 = Auto("Peters Auto", 100);
  auto1.defekt();
  return 0;
}
```

Vererbungsattribute

Welche Zugriffspezifizierer sind für die Mitglieder einer Basisklasse zu verwenden damit die folgenden Aussagen zutreffen?

`private`	`protected`	`public`	
◯	◯	◯	Zugriff ist aus einer beliebigen Klasse möglich.
◯	◯	◯	Zugriff ist nur innerhalb der Basisklasse möglich.
◯	◯	◯	Zugriff ist in der Basisklasse und in erbenden Klassen möglich.

Polymorphie

Was ist Polymorphie?

- ◯ Eine Technik, die es verhindert, bestehende Methoden in den ableiteten Klassen aufzurufen
- ◯ Eine Technik, die es ermöglicht, bestehende Methoden zu überschreiben
- ◯ Eine Technik, die es ermöglicht, Datenfelder einer Klasse in den abgeleiteten Klassen zu überschreiben

Welche Aussagen treffen im Bezug auf Polymorphie zu?

- ◯ Polymorphie soll beim Erstellen einer abgeleiteten Klasse immer durch Vergabe eines anderen Namens umgangen werden.
- ◯ Polymorphie ermöglicht die Methoden der Basisklase in der abgeleiteten Klasse mit einer anderen Funktionalität zu versehen.
- ◯ Polymorphie wird verwendet um die Methoden der Basisklasse zu löschen.

Wie lautet die Ausgabe dieses Programms?

```
#include <iostream>

class Basisklasse {
  public:
    void ausgabe() {
      std::cout << "Ausgabe1\n";
    }
```

```
};

class Ableitungsklasse : public Basisklasse {
  public:
    void ausgabe() {
      std::cout << "Ausgabe2\n";
    }
};

int main() {
  Basisklasse b = Basisklasse();
  b.ausgabe();

  Ableitungsklasse a = Ableitungsklasse();
  a.ausgabe();
  return 0;
}
```