Herzlich willkommen!

Dozent: Dipl.-Ing. Jürgen Wemheuer

Mail: wemheuer@ewla.de
Online: http://cpp.ewla.de/
https://ewla.de/dozent/cpp/

Disclaimer

- Diese Vorlesungs-/Unterrichtsfolien basieren auf Skripte meiner geschätzten Fachkollegen Prof. Dr. Dietrich Kuhn († 2010) und Prof. Dr. Stefan Enderle der Naturwissenschaftlich-Technischen Akademie (nta) in Isny.
- Das Skript wurde durch den Dozenten ausschließlich für die Gestaltung seines Unterrichts / seiner Vorlesung zusammengestellt bzw. verfasst und ist nicht als Referenz einer Programmiersprache gedacht.
- Dem Vorlesungsskript mangelt es an jeglichem Kontext.
  Dieser ist vielmehr der bestimmende Lehrinhalt in den Vorlesungen.
- Nicht alle Inhalte des Vorlesungsskripts sind prüfungsrelevant.
- Nicht alle prüfungsrelevanten Fakten sind im Vorlesungsskript enthalten.
- Ausschlaggebend für Prüfungen sind deshalb allein die im Unterricht bzw. in den Übungen und/oder Projektbeispielen vorgebrachten Inhalte.
- Aktuelle Änderungen des Vorlesungsskripts sind jederzeit vorbehalten.
- Mit allen auftretenden Fragen zum Fachgebiet und dem Vorlesungsskript sollten sich die SchülerInnen und StudentInnen stets an den Dozenten wenden.
- Das Vorlesungsskript wurde mit bestem Wissen und Gewissen und sorgfältig erarbeitet, jedoch können Irrtümer und Fehler nicht ausgeschlossen werden.
- Jegliche Haftung und Gewährleistung ist ausgeschlossen.

Inhalt:

Klassenattribute:

- static
- const
- friend

Aggregationen:

- Objekte als Klassenelemente

Statische Datenelemente (1)

Mehrere Objekte einer Klasse können Daten gemeinsam benutzen.

Diese Daten sind nur 1x vorhanden, gehören also nicht zu den einzelnen Objekten, sondern zur Klasse (*Klassenvariablen*).

Beispiel:

```
class Kunde
{
private:
  string name;
  static int anzahl; // nur 1x für alle Kunden
public:
  Kunde(string n);
...
};
```

• Deklaration durch *static* → „Statische Datenelemente“

Statische Datenelemente (2)

Statische Datenelemente existieren und belegen Speicherplatz, auch wenn noch kein Objekt existiert!

Die **Initialisierung** erfolgt **außerhalb** der Klasse.

Beispiel:

```
class Kunde
{
private:
  string name;
  static int anzahl;
public:
  Kunde(string n);
...
};
int Kunde::anzahl = 0;
```

Statische Datenelemente: Beispiel

Erhöhen des Zählers **anzahl** im Konstruktor:

```
class Kunde
{
private:
  string name;
  static int anzahl;
public:
  Kunde(string n);
...
};
int Kunde::anzahl = 0;

Kunde::Kunde(string n) {
  name = n;
  anzahl++;
}
```

Statische Datenelemente: Zugriff

Zugriff wie üblich:

- ein public-Element kann von überall angesprochen werden,
- ein private-Element nur innerhalb einer Methode der Klasse.

Bei statischen Elementen gibt es zwei public-Möglichkeiten:

- Über ein Objekt (sofern schon eines existiert…):

```
Kunde k("Mueller");
cout << k.anzahl;
```

- Über die Klasse (das geht immer…):

```
cout << Kunde::anzahl;
```

Statische Methoden

Um private statische Datenelemente abzufragen,
werden in der Regel auch statische Methoden benutzt.

Statische Datenelemente existieren unabhängig von Objekten
– so auch statische Methoden.

Beispiel:

```
class Kunde
{
private:
  string name;
  static int anzahl; // statisches privates Attribut
public:
  Kunde(string n);
  static int getAnzahl() { return anzahl; }
...
};
```

Statische Methoden: Aufruf

Wie statische Elemente kann auch eine statische Methode auf zwei Arten aufgerufen werden:

- Über ein Objekt (sofern schon eines existiert…):

```
Kunde k("Mueller");
cout << k.getAnzahl();
```

- Über die Klasse (das geht immer…):

```
cout << Kunde::getAnzahl();
```

Da statische Methoden unabhängig von Objekten der Klasse existieren, (also auch bereits bevor Objekte existieren) dürfen sie auch

- NUR statische Elemente benutzen,
- NUR (evtl. andere vorhandene) statische Methoden aufrufen.

Konstante Datenelemente

Sollen Attribute einer Klasse nicht verändert werden, können sie als *const* deklariert werden:

Beispiel:

```
class Person
{
private:
  string name;
  const int geburtsjahr; // keine Veränderung
public:
  Person(string n, int gj);
  ...
};
```

Konstante Elemente initialisieren

Wir erinnern uns: *const*-Variablen müssen bei der Deklaration sogleich initialisiert werden.
Ein späterer schreibender Zugriff ist nicht mehr möglich!

```
const int meinInt;          // das geht so
meinInt = 5;                // leider nicht!

const int meinInt = 5;      // das geht!
```

Genauso bei const-Elementen in Klassen-Konstruktoren:

```
(im Beispiel: const int geburtsjahr;)
Person(string n, int gj) {
  name = n;
  geburtsjahr = gj;         // geht so nicht!
}
```

Elementinitialisierung

Deshalb gibt es die Elementinitialisierung
über eine Initialisierungsliste direkt bei der Instanziierung:

```
class Person
{
private:
  string name;
  const int geburtsjahr; // keine Veränderung
public:
  Person(string n, int j);
  ...
};
Person::Person(string n, int j) :
       name(n),geburtstjahr(j) {}
```

Initialisierungsliste (1)

Die Initialisierungsliste geht aber immer, auch ohne const-Elemente, und zusätzlich zu anderen Konstruktoranweisungen:

```
class Datum {
private:
  int Tag, Monat, Jahr;
public:
  Datum(int t, int m, int j);
};

Datum::Datum(int t, int m, int j) :
              Tag(t),Monat(m),Jahr(j)
{ /* weitere Anweisungen */ }
```

Initialisierungsliste (2)

Die Initialisierungsliste funktioniert auch mit literalen und berechneten Initialisierungswerten:

```
class Datum {
  int Tag, Monat, Jahr;
  Datum();
};

int a = 2014;

Datum::Datum() : Tag(1),Monat(1),Jahr(++a) {}
```

(Die Initialisierungsliste geht natürlich nicht für normale Zuweisungen in anderen Funktionen und set-Methoden…)

Objekte als Klassenattribute

Eine Klasse kann als Attribute Objekte von anderen Klassen besitzen (haben) / (Hat-Beziehung, Aggregation).
Zuerst werden die inneren Objekte konstruiert, dann das äußere Objekt.

Beispiel:

```
class Datum {
  int Tag, Monat, Jahr;
public:
  Datum(int t, int m, int j): Tag(t),Monat(m),Jahr(j) {}
};
class Person {
  string name; // Klasse string
  Datum geburtstag; // Klasse Datum
public:
  Person(string n, int t, int m, int j);
...
};
```

Initialisierung mit Objekten


```
class Datum {
  int Tag, Monat, Jahr;
public:
  Datum(int t, int m, int j) : Tag(t),Monat(m),Jahr(j) {}
};
class Person {
  string name; // Klasse string
  Datum geburtstag; // Klasse Datum
public:
  Person(string n, int t, int m, int j);
...
};
```

Initialisierung mit Elementinitialisierung / Initialisierungsliste:

```
Person::Person(string n, int t, int m, int j) :
                name(n), geburtstag(t,m,j) {}
```

Man kann es zwar auch anders machen, aber nur mit einem spürbaren Verlust an Effizienz, Eleganz und Sicherheit! Siehe Tafelbeispiele…

Friend-Funktionen

Funktionen, die keine Methoden einer Klasse sind, haben normalerweise keinen Zugriff auf private Datenelemente der Klasse. Manchmal ist es (z.B. aus Effizienz-Gründen) jedoch sinnvoll, dass eine Funktion Zugriff auf private Datenelemente bekommt. Beispiel:

```
class Kunde
{
private:
  string name;
public:
  Kunde(string n);
  ...
};

void kundenliste()
{
  for (int i=0; i<anzahlKunden; i++)
  cout << kunde[i].name;              // Fehler: Kein Zugriff
}
```

Du bist mein Freund: friend

Eine Klasse kann den Zugriff auf ihre privaten Daten explizit erlauben.
Dies geschieht durch **friend** und Angabe der Funktion:

```
class Kunde
{
private:
  string name;
public:
  Kunde(string n);
  ...
  friend void kundenliste(); // Darf zugreifen
};

void kundenliste()
{
  for (int i=0; i<anzahlKunden; i++)
  cout << kunde[i].name;              // OK!
}
```

Friend-Klassen (1)

Manchmal arbeiten zwei Klassen so eng zusammen, dass es sinnvoll ist, einer Klasse sämtliche Rechte zum Zugriff auf die privaten Datenelemente der anderen Klasse zu geben.

```
class Kunde
{
private:
  string name;
public:
  Kunde(string n);
  ...
};

class Kundendatenbank
{
      // Kein direkter Zugriff auf Kunde::name
};
```

Friend-Klassen (2)

Die Klasse Kunde erlaubt der Klasse Kundendatenbank Zugriff auf ihre privaten Datenelemente:

```
class Kunde
{
private:
  string name;
public:
  Kunde(string n);
  ...
  friend class Kundendatenbank;
};

class Kundendatenbank {
      // Für die Kundendatenbank werden alle privaten
      // Daten und Methoden von Kunde sichtbar.
};
```

Ende