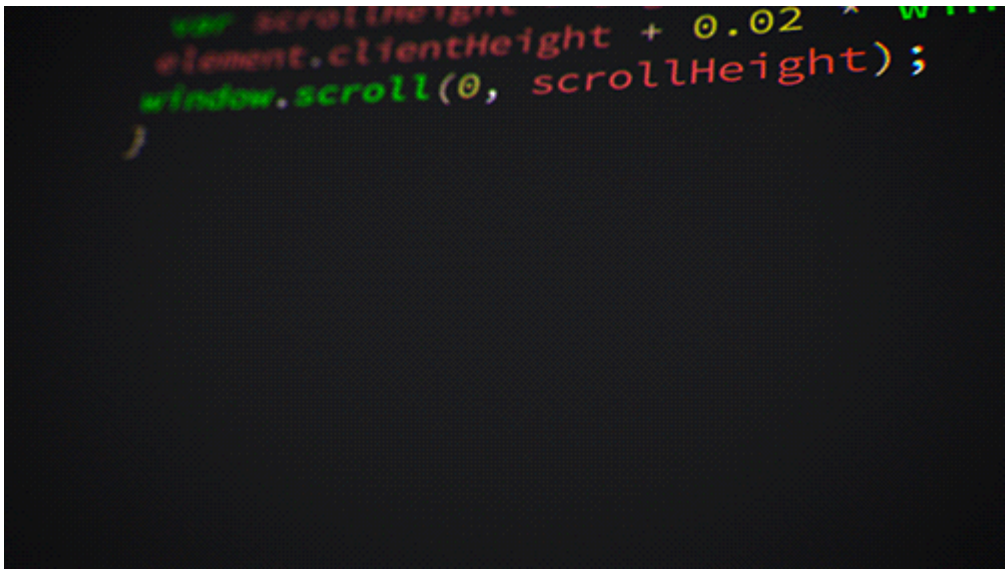
Objektorientierte Programmierung mit C++

Parameter	Kursinformationen
Veranstaltung:	`Prozedurale Programmierung / Einführung in die Informatik`
Semester	`Wintersemester 2022/23`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Klassen und Objekte`
Link auf Repository:	https://github.com/TUBAF-IfI-LiaScript/VL_ProzeduraleProgrammierung/blob/master/04_Funktionen.md
Autoren	Sebastian Zug & André Dietrich & Galina Rudolf



Fragen an die heutige Veranstaltung ...

- Wie strukturieren wir unseren Code?
- Was sind Objekte?
- Welche Aufgabe erfüllt ein Konstruktor?
- Was geschieht, wenn kein expliziter Konstruktor durch den Entwickler vorgesehen wurde?
- Wann ist ein Dekonstruktor erforderlich?
- Was bedeutet das "Überladen" von Funktionen?
- Nach welchen Kriterien werden überladene Funktionen differenziert?

Motivation

> Aufgabe: Statistische Untersuchung der Gehaltsentwicklung in Abhängigkeit vom Alter.

Welche zwei Elemente machen eine solche Untersuchung aus?

Daten	**Funktionen**
Name	Alter bestimmen
Geburtsdatum	Daten ausgeben
Gehalt	
...	

Wir entwerfen also eine ganze Sammlung von Funktionen wie zum Beispiel für `alterbestimmen()`:

```
int alterbestimmen(int tag, int monat, int jahr,
                   int akt_tag, int akt_monat, int akt_jahr)
{
  //TODO
}

int main(){
  int tag, monat, jahr;
  int akt_tag, akt_monat, akt_jahr;
  int alter=alterbestimmen(tag,monat,jahr,akt_tag,akt_monat,akt_jahr);
}
```

Was gefällt ihnen an diesem Ansatz nicht?

- lange Parameterliste bei der Funktionen
- viele Variablen
- der inhaltliche Zusammenhang der Daten schwer zu erkennen

Lösungsansatz 1: Wir fassen die Daten zusammen in einer übergreifenden Datenstruktur `struct`.

Ein struct vereint unterschiedliche Aspekte eines Datensets in einer Variablen.

```
struct Datum{           // hier wird ein neuer Datentyp definiert
    int tag, monat, jahr;
};

int alterbestimmen(Datum geb_datum, Datum akt_datum)
{
  //TODO
  return 0;
}

int main(){
  Datum geburtsdatum;
  Datum akt_datum;        // ... und hier wird eine Variable des
                          // Typs angelegt
  geburtsdatum.tag = 5;  // ... und "befüllt"
  geburtsdatum.monat = 10;
  geburtsdatum.jahr = 1923;
  int alter=alterbestimmen(geburtsdatum, akt_datum);
}
```

Mit `struct Datum` wurde ein Datentyp definiert, `Datum geburtstdatum;` definiert eine Variable (besser gesagt ein Objekt).

Was gefällt ihnen an diesem Ansatz nicht?

- die Funktionen sind von dem neuen Datentyp abhängig gehören aber trotzdem nicht zusammen
- es fehlt eine Überprüfung der Einträge für die Gültigkeit unserer Datumsangaben

Die objektorientierte Programmierung (OOP) ist ein Programmierparadigma, das auf dem Konzept der "Objekte" basiert, die Daten und Code enthalten können: Daten in Form von Feldern (oft als Attribute oder Eigenschaften bekannt) und Code in Form von Prozeduren (oft als Methoden bekannt).

```
struct Datum{
    int tag,monat,jahr;
    int alterbestimmen(Datum akt_datum)
    {
      //hier sind tag, monat und jahr bereits bekannt
      //TODO
    }
}

int main(){
  Datum geburtstdatum;
  Datum akt_datum;
  int alter=geburtstdatum.alterbestimmen(akt_datum);
}
```

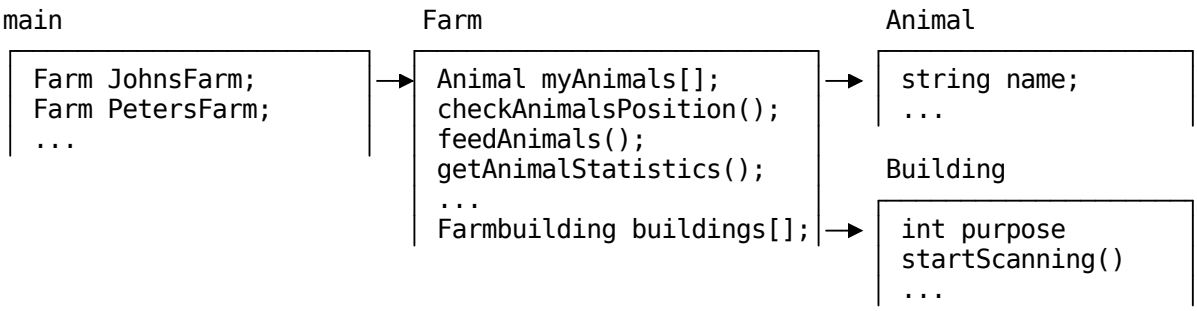
C++ sieht vor als Datentyp für Objekte `struct` - und `class` - Definitionen. Der Unterschied wird später geklärt, vorerst verwenden wird nur die `class` - Definitionen.

```
class Datum{
public:
    int tag,monat,jahr;
    int alterbestimmen(Datum akt_datum)
    {
      //hier sind tag, monat und jahr bereits bekannt
      //TODO
    }
}

int main(){
  Datum geburtstdatum;
  Datum akt_datum;
  int alter=geburtstdatum.alterbestimmen(akt_datum);
}
```

Ein Merkmal von Objekten ist, dass die eigenen Prozeduren eines Objekts auf die Datenfelder seiner selbst zugreifen und diese oft verändern können - Objekte haben somit eine Vorstellung davon oder von sich selbst 🙂.

Ein OOP-Computerprogramm kombiniert Objekt und lässt sie interagieren. Viele der am weitesten verbreiteten Programmiersprachen (wie C++, Java, Python usw.) sind Multi-Paradigma-Sprachen und unterstützen mehr oder weniger objektorientierte Programmierung, typischerweise in Kombination mit imperativer, prozeduraler Programmierung.



Wir erzeugen ausgehend von unserem Bauplan verschiedene Instanzen / Objekte vom Typ `Animals`. Jede hat den gleichen Funktionsumfang, aber unterschiedliche Daten.

> **Merke:** Unter einer Klasse versteht man in der objektorientierten Programmierung ein abstraktes Modell bzw. einen Bauplan für eine Reihe von ähnlichen Objekten.

C++ - Entwicklung

C++ eine von der ISO genormte Programmiersprache. Sie wurde ab 1979 von Bjarne Stroustrup bei AT&T als Erweiterung der Programmiersprache C entwickelt. Bezeichnenderweise trug C++ zunächst den Namen "**C with classes**". C++ erweitert die Abstraktionsmöglichkeiten erlaubt aber trotzdem eine maschinennahe Programmierung. Der Standard definiert auch eine Standardbibliothek, zu der wiederum verschiedene Implementierungen existieren.

Jahr	Entwicklung
197?	Anfang der 70er Jahre entsteht die Programmiersprache C
...	
1979	„C with Classes“ - Klassenkonzept mit Datenkapselung, abgeleitete Klassen, ein strenges Typsystem, Inline-Funktionen und Standard-Argumente
1983	"C++" - Überladen von Funktionsnamen und Operatoren, virtuelle Funktionen, Referenzen, Konstanten, eine änderbare Freispeicherverwaltung
1985	Referenzversion
1989	Version 2.0 - Mehrfachvererbung, abstrakte Klassen, statische Elementfunktionen, konstante Elementfunktionen
1998	ISO/IEC 14882:1998 oder C++98
...	Kontinuierliche Erweiterungen C++11, C++20, ...

C++ kombiniert die Effizienz von C mit den Abstraktionsmöglichkeiten der objektorientierten Programmierung. C++ Compiler können C Code überwiegend kompilieren, umgekehrt funktioniert das nicht.

... das kennen wir doch schon

Ein- und Ausgabe

Bereits häufig verwendete std::cin und std::cout sind Objekte:

- `std::cin` ist ein Objekt der Klasse `istream`
- `std::cout` ist ein Objekt der Klasse `ostream`

```
extern std::istream cin;
```

```
extern std::ostream cout;
```

Sie werden in `<iostream>` als Komponente der Standard Template Library (STL) im Namensraum `std` definiert und durch `#include <iostream>` bekannt gegeben.

Die Streams lassen sich nicht nur für die Standard Ein- und Ausgabe verwenden, sondern auch mit Dateien oder Zeichenketten.

Mehr zu `iostream` unter https://www.c-plusplus.net/forum/topic/152915/ein-und-ausgabe-in-c-io-streams

Klasse string

C++ implementiert eine separate Klasse `string`-Datentyp (Klasse). In Programmen müssen nur Objekte dieser Klasse angelegt werden. Beim Anlegen eines Objektes muss nicht angegeben werden, wie viele Zeichen darin enthalten werden sollen, eine einfache Zuweisung reicht aus.

string.cpp

```cpp
#include <iostream>
#include <string>

int main(void) {
  std::string hallo;
  hallo="Hallo";
  std::string hanna = "Hanna";
  std::string anna("Anna");
  std::string alleBeide = anna + " + " + hanna;
  std::cout<<hallo<<" "<<alleBeide<<std::endl;
  return 0;
}
```

Arduino Klassen

Die Implementierungen für unseren Mikrocontroller sind auch Objektorientiert. Klassen repräsentieren unserer Hardwarekomponenten und sorgen für deren einfache Verwendung.

ArduinoDisplay.cpp

```cpp
#include <OledDisplay.h>
...
Screen.init();
Screen.print("This is a very small display including only 4 lines", true);
Screen.draw(0, 0, 128, 8, BMP);
Screen.clean();
...
```

Definieren von Klassen und Objekten

Eine Klasse wird in C++ mit dem Schlüsselwort `class` definiert und enthält Daten (member variables, Attribute) und Funktionen (member functions, Methoden).

Klassen verschmelzen Daten und Methoden in einem "Objekt" und deklarieren den individuellen Zugriff. Die wichtigste Eigenschaft einer Klasse ist, dass es sich um einen Typ handelt!

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
};   // <- Das Semikolon wird gern vergessen

class_name instance_name;
```

Bezeichnung	Bedeutung
`class_name`	Bezeichner für die Klasse - Typ
`instance_name`	Objekte der Klasse `class_name` - Instanz
`access_specifier`	Zugriffsberechtigung

Der Hauptteil der Deklaration kann *member* enthalten, die entweder Daten- oder Funktionsdeklarationen sein können und jeweils einem Zugriffsbezeichner . Ein Zugriffsbezeichner ist eines der folgenden drei Schlüsselwörter: `private`, `public` oder `protected`. Diese Bezeichner ändern die Zugriffsrechte für die *member*, die ihnen nachfolgen:

- `private` *member* einer Klasse sind nur von anderen *members* derselben Klasse (oder von ihren "Freunden") aus zugänglich.
- `protected` *member* sind von anderen *member* derselben Klasse (oder von ihren "Freunden") aus zugänglich, aber auch von Mitgliedern ihrer abgeleiteten Klassen.
- `public` *member* sind öffentliche *member* von überall her zugänglich, wo das Objekt sichtbar ist.

Standardmäßig sind alle Members in der Klasse `private`!

> **Merke:** Klassen und Strukturen unterscheiden sich unter C++ durch die Default-Zugriffsrechte und die Möglichkeit der Vererbung. Anders als bei `class` sind die *member* von `struct` per Default *public* . Die Veränderung der Zugriffsrechte über die oben genannten Zugriffsbezeichner ist aber ebenfalls möglich.

Im Folgenden fokussieren die Ausführungen Klassen, eine analoge Anwendung mit Strukturen ist aber zumeist möglich.

ClassExample.cpp

```cpp
#include <iostream>

class Datum
{
  public:
    int tag;
    int monat;
    int jahr;

    void print(){
        std::cout << tag << "." << monat <<"." << jahr <<std::endl;
    }
};

int main()
{
  // 1. Instanz der Klasse, Objekt myDatumA
  Datum myDatumA;
  myDatumA.tag = 12;
  myDatumA.monat = 12;
  myDatumA.jahr = 2000;
  myDatumA.print();

  // 2. Instanz der Klasse, Objekt myDatumB
  // alternative Initialisierung
  Datum myDatumB = {.tag = 12, .monat = 3, .jahr = 1920};
  myDatumB.print();
  return 0;
}
```

Und wie können wir weitere Methoden zu unserer Klasse hinzufügen?

AdditionalMethod.cpp

```cpp
#include <iostream>

class Datum{
  public:
    int tag;
    int monat;
    int jahr;

    const int monthDays[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31
      31};

    void print(){
        std::cout << tag << "." << monat <<"." << jahr <<std::endl;
    }

    int istSchaltJahr(){
      if (((jahr % 4 == 0) && (jahr % 100 != 0)) || (jahr % 400 == 0)
        return 1;
      else return 0;
    }

    int nterTagimJahr(){
      int n = tag;
      for (int i=0; i<monat - 1; i++){
          n += monthDays[i];
      }
      if (monat > 2) n += istSchaltJahr();
      return n;
    }
};

int main()
{
  Datum myDatumA;
  myDatumA.tag = 31;
  myDatumA.monat = 12;
  myDatumA.jahr = 2000;
  myDatumA.print();
  std::cout << myDatumA.nterTagimJahr() << "ter Tag im Jahr " << myDa
    .jahr << std::endl;
  return 0;
}
```

```
31.12.2000
366ter Tag im Jahr 2000
```

Objekte in Objekten

Natürlich lassen sich Klassen beliebig miteinander kombinieren, was folgende Beispiel demonstriert.

ClassOfFriends.cpp

```cpp
#include <iostream>
#include <ctime>
#include <string>

class Datum{
  public:
    int tag;
    int monat;
    int jahr;

    void print(){
        std::cout << tag << "." << monat <<"." << jahr <<std::endl;
    }
};

class Person{
  public:
    Datum geburtstag;
    std::string name;

    void print(){
        std::cout << name << ": ";
        geburtstag.print();
        std::cout <<std::endl;
    }

    int zumGeburtstagAnrufen() {
       time_t t = time(NULL);
       tm* tPtr = localtime(&t);
       if ((geburtstag.tag == tPtr->tm_mday) &&
           (geburtstag.monat == (tPtr->tm_mon + 1))) {
          std::cout << "\"Weil heute Dein ... \"" <<std::endl;
          return 1;
       }
       else return -1;
    }
};

int main()
{
  Person freundA = {.geburtstag = {1, 12, 2022}, .name = "Peter"};
  freundA.print();
  if (freundA.zumGeburtstagAnrufen() == 1){
     std::cout << "Zum Geburtstag gratuliert!" <<std::endl;
  }
  else{
     std::cout << freundA.name << " hat heute nicht Geburtstag" <<st
       ::endl;
```

```
48    }
49    return 0;
50  }
```

```
Peter: 1.12.2022

Peter hat heute nicht Geburtstag
```

Datenkapselung

Als Datenkapselung bezeichnet man das Verbergen von Implementierungsdetails einer Klasse. Auf die internen Daten kann nicht direkt zugegriffen werden, sondern nur über definierte Schnittstellen, die durch `public`-Methoden repräsentiert wird.

- get- und set-Methoden
- andere Methoden

getSetMethods.cpp

```cpp
#include <iostream>
using namespace std;
class Datum
{
    private:
      int tag;
      int monat;
      int jahr;
    public:
      void setTag(int _tag){
        tag=_tag;
      }
      void setMonat(int _monat){
        monat=_monat;
      }
      void setJahr(int _jahr){
        jahr=_jahr;
      }
      int getTag(){
        return tag;
      }
      //analog monat und jahr
      void ausgabe()
      {
          cout<<tag<<"."<<monat<<"."<<jahr<<endl;
      }
};

int main()
{
  Datum datum;
  datum.setTag(31);datum.setMonat(12);datum.setJahr(1999);
  datum.ausgabe();//jetzt geht es
}
```

```
31.12.1999
```

Memberfunktionen

Mit der Integration einer Funktion in eine Klasse wird diese zur *Methode* oder *Memberfunktion*. Der Mechanismus der Nutzung bleibt der gleiche, es erfolgt der Aufruf, ggf mit Parametern, die Abarbeitung realisiert Berechnungen, Ausgaben usw. und ein optionaler Rückgabewert bzw. geänderte Parameter (bei Call-by-Referenz Aufrufen) werden zurückgegeben.

Worin liegt der technische Unterschied?

ApplicationOfStructs.cpp

```cpp
#include <iostream>

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

    void ausgabeMethode(){
        std::cout << name << " " << ort << " " << alter  << std::endl
    }
};

void ausgabeFunktion(Student studentA){
    std::cout << studentA.name << " " << studentA.ort << " " << stude
      .alter  << std::endl;
}

int main()
{
  Student bernhard {"Cotta", 25, "Zillbach"};
  bernhard.ausgabeMethode();

  ausgabeFunktion(bernhard);

  return 0;
}
```

```
Cotta Zillbach 25
Cotta Zillbach 25
```

Methoden können auch von der Klassendefinition getrennt werden.

ApplicationOfStructs.cpp

```cpp
#include <iostream>

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

    void ausgabeMethode();    // Deklaration der Methode
};

// Implementierung der Methode
void Student::ausgabeMethode(){
    std::cout << name << " " << ort << " " << alter  << std::endl;
}

int main()
{
  Student bernhard {"Cotta", 25, "Zillbach"};
  bernhard.ausgabeMethode();
  return 0;
}
```

```
Cotta Zillbach 25
```

Diesen Ansatz kann man weiter treiben und die Aufteilung auf einzelne Dateien realisieren.

Modularisierung unter C++

Der Konzept der Modularisierung lässt sich unter C++ durch die Aufteilung der Klassen auf verschiedene Dateien umsetzen. Unser kleines Beispiel umfasst Klassen, die von einer `main.cpp` aufgerufen werden.

Datum.h

```
#include <iostream>

#ifndef DATUM_H_INCLUDED
#define DATUM_H_INCLUDED
/* ^^ these are the include guards */

class Datum
{
  public:
    int tag;
    int monat;
    int jahr;

    void ausgabeMethode(){
        std::cout << tag << "." << monat <<"." << jahr <<std::endl;
    }
};

#endif
```

Student.h

```
#ifndef STUDENT_H_INCLUDED
#define STUDENT_H_INCLUDED

#include <iostream>
#include <string>
#include "Datum.h"

class Student{
  public:
    std::string name;  // "-"
    Datum geburtsdatum;
    std::string ort;

    void ausgabeMethode();    // Deklaration der Methode
};

#endif
```

Student.cpp

```cpp
#include "Student.h"

void Student::ausgabeMethode(){
    std::cout << name << " " << ort << " ";
    geburtsdatum.ausgabeMethode();
    std::cout << std::endl;
}

```

main.cpp

```cpp
#include <iostream>
#include "Student.h"

int main()
{
  Datum datum{1,1,2000};
  Student bernhard {"Cotta", datum, "Zillbach"};
  bernhard.ausgabeMethode();
  return 0;
}
```

```
Cotta Zillbach 1.1.2000
```

```
g++ -Wall main.cpp Student.cpp -o a.out
```

Definitionen der Klassen erfolgen in den Header-Dateien (.h), wobei für die meisten member-Funktionen nur die Deklarationen angegeben werden. In den Implementierungdateien (.cpp) werden die Klassendefinitionen mit include-Anweisungen bekannt gemacht und die noch nicht implementierten member-Funktionen implementiert.

> **Achtung:** Die außerhalb der Klasse implementierte Funktionen erhalten einen Namen, der den Klassenname einschließt.

Überladung von Methoden

C++ verbietet für die Variablen und Objekte die gleichen Namen, erlaubt jedoch die variable Verwendung von Funktionen in Abhängigkeit von der Signatur der Funktion. Dieser Mechanismus heißt "Überladen von Funktionen" und ist sowohl an die global definierten Funktionen als auch an die Methoden der Klasse

anwendbar.

> **Merke:** Der Rückgabedatentyp trägt nicht zur Unterscheidung der Methoden bei. Unterscheidet sich die Signatur nur in diesem Punkt, "meckert" der Compiler.

ostream.cpp

```cpp
#include <iostream>
#include <fstream>

class Seminar{
  public:
    std::string name;
    bool passed;
};

class Lecture{
  public:
    std::string name;
    float mark;
};

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

    void printCertificate(Seminar sem){
      std::string comment = " nicht bestanden";
      if (sem.passed)
        comment = " bestanden!";
      std::cout << "\n" << name << " hat das Seminar " << sem.name
                <<  comment;
    }

    void printCertificate(Lecture lect){
      std::cout << "\n" << name << " hat in der Vorlesung " <<
                   lect.name << " die Note " << lect.mark << " erreic
    }
};

int main()
{
  Student bernhard {"Cotta", 25, "Zillbach"};
  Seminar roboticSeminar {"Robotik-Seminar", false};
  Lecture ProzProg {"Prozedurale Programmierung", 1.3};

  bernhard.printCertificate(roboticSeminar);
  bernhard.printCertificate(ProzProg);

  return 0;
}
```

```
Cotta hat das Seminar Robotik-Seminar nicht bestanden
Cotta hat in der Vorlesung Prozedurale Programmierung die Note 1.3
erreicht
```

Achtung: Im Beispiel erfolgt die Ausgabe nicht auf die Console, sondern in ein ostream-Objekt, dessen Referenz an die print-Methoden als Parameter übergeben wird. Das ermöglicht eine flexible Ausgabe, z.B. in eine Datei, auf den Drucker etc.

Ein Wort zur Ausgabe

ostream.cpp

```cpp
#include <iostream>
#include <fstream>

class Seminar{
  public:
    std::string name;
    bool passed;
};

class Lecture{
  public:
    std::string name;
    float mark;
};

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

    void printCertificate(std::ostream& os, Seminar sem);
    void printCertificate(std::ostream& os, Lecture sem);
};

void Student::printCertificate(std::ostream& os, Seminar sem){
  std::string comment = " nicht bestanden";
  if (sem.passed)
    comment = " bestanden!";
  os << "\n" << name << " hat das Seminar " << sem.name <<  comment;
}

void Student::printCertificate(std::ostream& os, Lecture lect){
  os << "\n" << name << " hat in der Vorlesung " << lect.name << " di
    " << lect.mark << " erreicht";
}

int main()
{
  Student bernhard {"Cotta", 25, "Zillbach"};
  Seminar roboticSeminar {"Robotik-Seminar", false};
  Lecture ProzProg {"Prozedurale Programmierung", 1.3};

  bernhard.printCertificate(std::cout, roboticSeminar);
  bernhard.printCertificate(std::cout, ProzProg);

  return 0;
}
```

```
Cotta hat das Seminar Robotik-Seminar nicht bestanden
Cotta hat in der Vorlesung Prozedurale Programmierung die Note 1.3
erreicht
```

Initalisierung/Zerstören eines Objektes

Die Klasse spezifiziert unter anderem (!) welche Daten in den Instanzen/Objekten zusammenfasst werden. Wie aber erfolgt die Initialisierung? Bisher haben wir die Daten bei der Erzeugung der Instanz übergeben.

Konstruktoren.cpp

```cpp
#include <iostream>

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

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

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

int main()
{
  Student bernhard {"Cotta", 25, "Zillbach"};
  bernhard.ausgabeMethode(std::cout);

  Student alexander { .name = "Humboldt" , .ort = "Berlin"  };
  alexander.ausgabeMethode(std::cout);

  Student unbekannt;
  unbekannt.ausgabeMethode(std::cout);

  return 0;
}
```

```
Cotta Zillbach 25
Humboldt Berlin 0
  0
```

Es entstehen 3 Instanzen der Klasse `Student`, die sich im Variablennamen `bernhard`, `alexander` und `unbekannt` und den Daten unterscheiden.

Im Grunde können wir unsere drei Datenfelder im Beispiel in vier Kombinationen initialisieren:

```
{name, alter, ort}
{name, alter}
{name}
{}
```

Elementinitialisierung beim Aufruf:

Umsetzung	**Beispiel**
vollständige Liste in absteigender Folge (uniforme Initialisierung)	`Student Bernhard {"Cotta", 25, "Zillbach"};`
unvollständige Liste (die fehlenden Werte werden durch Standard Defaultwerte ersetzt)	`Student Bernhard {"Cotta", 25};`
vollständig leere Liste, die zum Setzen von Defaultwerten führt	`Student Bernhard {};`
Aggregierende Initialisierung (C++20)	`Student alexander = { .ort = "unknown"};`

Wie können wir aber:

- erzwingen, dass eine bestimmte Membervariable in jedem Fall gesetzt wird (weil die Klasse sonst keinen Sinn macht)
- prüfen, ob die Werte einem bestimmten Muster entsprechen ("Die PLZ kann keine negativen Werte umfassen")
- automatische weitere Einträge setzen (einen Zeitstempel, der die Initialisierung festhält)
- ... ?

Konstruktoren

Konstruktoren dienen der Koordination der Initialisierung der Instanz einer Klasse. Sie werden entweder implizit über den Compiler erzeugt oder explizit durch den Programmierer angelegt.

```
class class_name {
  access_specifier_1:
    typ member1;
```

```
  access_specifier_2:
    typ member2;
    memberfunktionA(...)

  class_name (...) {            // <- Konstruktor
    // Initalisierungscode
  }
};

class_name instance_name (...);
```

Merke: Ein Konstruktor hat keinen Rückgabetyp!

Beim Aufruf `Student bernhard {"Cotta", 25, "Zillbach"};` erzeugt der Compiler eine Methode `Student::Student(std::string, int, std::string)`, die die Initialisierungsparameter entgegennimmt und diese der Reihenfolge nach an die Membervariablen übergibt. Sobald wir nur einen explizten Konstruktor integrieren, weist der Compiler diese Verantwortung von sich.

Entfernen Sie den Kommentar in Zeile 13 und der Compiler macht Sie darauf aufmerksam.

defaultConstructor.cpp

```
 1  #include <iostream>
 2
 3  class Student{
 4    public:
 5      std::string name;  // "-"
 6      int alter;
 7      std::string ort;
 8
 9      void ausgabeMethode(std::ostream& os){
10          os << name << " " << ort << " " << alter;
11      }
12
13      //Student();
14  };
15
16  // Implementierung der Methode
17
18
19  int main()
20  {
21    Student bernhard {"Cotta", 25, "Zillbach"};
22    bernhard.ausgabeMethode(std::cout);
23    return 0;
24  }
```

```
Cotta Zillbach 25
```

Dabei sind innerhalb des Konstruktors zwei Schreibweisen möglich:

```
//Initalisierung
Student(std::string name, int alter, std::string ort): name(name), alter(a
  ort(ort)
{
}

// Zuweisung innerhalb des Konstruktors
Student(std::string name, int alter, std::string ort):
    this->name = name;
    this->alter = alter;
    this->ort = ort;
}
```

Die zuvor beschriebene Methodenüberladung kann auch auf die Konstruktoren angewandt werden. Entsprechend stehen dann eigene Aufrufmethoden für verschiedene Datenkonfigurationen zur Verfügung. In diesem Fall können wir auf drei verschiedenen Wegen Default-Werte setzen:

- ohne spezfische Vorgabe wird der Standardinitialisierungswert verwendt (Ganzzahlen 0, Gleitkomma 0.0, Strings "")
- die Vorgabe eines indivduellen Default-Wertes (vgl. Zeile 7)

constructor.cpp

```cpp
#include <iostream>

class Student{
  public:
    std::string name;
    int alter;
    std::string ort; // = "Freiberg";

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

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

    Student(std::string name): name(name)
    {
    }
};

int main()
{
  Student bernhard {"Cotta", 25, "Zillbach"};
  bernhard.ausgabeMethode(std::cout);

  Student alexander = Student("Humboldt");
  alexander.ausgabeMethode(std::cout);

  return 0;
}
```

```
Cotta Zillbach 25
Humboldt  0
```

Delegierende Konstruktoren rufen einen weiteren Konstruktor für die teilweise Initialisierung auf. Damit lassen sich Codeduplikationen, die sich aus der Kombination aller Parameter ergeben, minimieren.

```cpp
Student(std::string n, int a, std::string o): name{n}, alter{a}, ort{o} { }
Student(std::string n) : Student (n, 18, "Freiberg") {};
Student(int a, std::string o): Student ("unknown", a, o) {};
```

Destruktoren

Destructor.cpp

```cpp
#include <iostream>

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

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

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

Student::~Student(){
  std::cout << "Destructing object of type 'Student' with name = '" <
    ->name << "'\n";
}

int main()
{
  Student max {"Maier", 19, "Dresden"};
  std::cout << "End...\n";
  return 0;
}
```

Destruktoren werden aufgerufen, wenn eines der folgenden Ereignisse eintritt:

- Das Programm verlässt den Gültigkeitsbereich (*Scope*, d.h. einen Bereich der mit `{...}` umschlossen ist) eines lokalen Objektes.
- Ein Objekt, das `new`-erzeugt wurde, wird mithilfe von `delete` explizit aufgehoben (Speicherung auf dem Heap)
- Ein Programm endet und es sind globale oder statische Objekte vorhanden.
- Der Destruktor wird unter Verwendung des vollqualifizierten Namens der Funktion explizit aufgerufen.

Einen Destruktor explizit aufzurufen, ist selten notwendig (oder gar eine gute Idee!).

Beispiel des Tages

Aufgabe: Erweitern Sie das Beispiel um zusätzliche Funktionen, wie die Berechnung des Umfanges. Überwachen Sie die Eingaben der Höhe und der Breite. Sofern diese negativ sind, sollte die Eingabe zurückgewiesen werden.

Rectangle

```cpp
#include <iostream>
using namespace std;

class Rectangle {
  private:
    int width, height;
  public:
    void set_values (int,int);          // Deklaration
    int area() {return width*height;}   // Deklaration und Defintion
};

void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}

int main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

```
area: 12
```

Anwendung

1. Ansteuern einer mehrfarbigen LED

   Die Auflistung der Memberfunktionen der entsprechenden Klasse finden Sie unter Link

2. Abfragen eines Sensors

   Die Auflistung der Memberfunktionen der entsprechenden Klassen finden Sie unter Link

Der Beispielcode für die Anwendungen ist in den `examples` Ordnern des Projektes enthalten.

Quiz

Definieren von Klassen und Objekten

Grundlegend

Welches Schlüsselwort wird benutzt um eine Klasse zu definieren?

Muss `[_____]` im unten aufgeführten Beispiel wirklich durch ein Semikolon ersetzt werden?

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
}[_____]

class_name instance_name;
```

- ◯ Ja
- ◯ Nein

Welche der folgenden Schlüsselwörter regeln die Zugriffsrechte bei Klassen und Klassen-Member?

- [ ] void
- [ ] private
- [ ] general
- [ ] open
- [ ] public
- [ ] protected
- [ ] encrypted

Welcher Zugriffsbezeichner gilt standardmäßig für alle Member einer Klasse?

- ( ) private
- ( ) protected
- ( ) public

Ersetzen Sie `[_____]` durch den Aufruf der Methode `print` des Objektes `Beispielauto`?

```
#include <iostream>
#include <string>

class Auto
{
  public:
    std::string Hersteller;
    int Kilometerstand;
    int PS;

    void print(){
        std::cout << Hersteller << " - " << PS <<" PS - " << Kilometerstand
          Kilometer" << std::endl;
    }
};

int main()
{
  Auto Beispielauto;
  Beispielauto.Hersteller = "Hyundai";
  Beispielauto.Kilometerstand = 49564;
  Beispielauto.PS = 76;
  [_____]
}
```

```
}
```

Datenkapselung

Wodurch muss `[_____]` ersetzt werden um den Kilometerstand vom Objekt `Beispielauto` auf 40000 zu setzen?

```
#include <iostream>
#include <string>

class Auto
{
  private:
    std::string Hersteller;
    int Kilometerstand;
    int PS;

  public:
    void set_Hersteller(std::string _Hersteller){
        Hersteller = _Hersteller;
    }
    void set_Kilometerstand(int _Kilometerstand){
        Kilometerstand = _Kilometerstand;
    }
    void set_PS(int _PS){
        PS = _PS;
    }
};

int main()
{
  Auto Beispielauto;
  [_____]
}
```

Memberfunktion

Vervollständigen Sie die Implementierung der Methode `ausgabeMethode` in dem Sie `[_____]` durch noch fehlenden Teil ersetzen. Geben Sie die Antwort ohne Leerzeichen ein.

```
#include <iostream>

class Auto
{
  public:
    std::string Hersteller;
    int Kilometerstand;
    int PS;

    void ausgabeMethode();
};

// Implementierung der Methode
void [_____]{
    std::cout << Hersteller << " - " << Kilometerstand << " km " << PS << "
      << std::endl;
}

int main()
{
  Auto beispielauto {"Tesla", 25000, 283};
  beispielauto.ausgabeMethode();
  return 0;
}
```

Modularisierung unter C++

Im Programm `main.cpp` soll die in der Datei `Auto.h` deklarierte Klasse `Auto` verwendet werden. Welche Dateien werden dafür benötigt?

- [ ] `main.h`
- [ ] `main.cpp`
- [ ] `Auto.h`
- [ ] `Auto.cpp`

Im folgenden Programm soll die in der Datei `Auto.h` deklarierte Klasse `Auto` verwendet werden. Wodurch muss `[_____]` ersetzt werden um das zu ermöglichen? Es kann davon ausgegangen werden, dass alle benötigten Dateien im selben Ordner liegen.

```
#include <iostream>
#include [_____]

int main()
{
  Auto Beispielauto{"Tesla", 25000, 283};
  Beispielauto.ausgabeMethode();
  return 0;
}
```

Überladung von Methoden

Wie lautet die Ausgabe dieses Programms?

```
#include <iostream>

class Force
{
  public:
    double Newton(double mass){
      double F = mass * 9.81;
      return F;
    };

    double Newton(double mass, double acc){
      double F = mass * acc;
      return F;
    };
};
```

```
};

int main()
{
  Force Kraft;
  double G = Kraft.Newton(10);
  double F = Kraft.Newton(10, 10);
  double R = G + F;
  std::cout << R << std::endl;
}
```

Initialisierung/Zerstören eines Objektes

Konstruktoren

Wie lautet die Ausgabe dieses Programms?

```
#include <iostream>

class Student{
  public:
    std::string name;
    int alter;
    std::string lieblingstier = "Dikdik";

    void ausgabeMethode(std::ostream& os){
        os << name << " " << lieblingstier << " " << alter;
    }

    Student(std::string name, int alter, std::string lieblingstier): name(n
      alter(alter), lieblingstier(lieblingstier)
    {
    }

    Student(std::string name): name(name),alter(0)
    {
    }
};

int main()
{
  Student alexander = Student("Humboldt");
  alexander.alter = 19;
```

```
  alexander.ausgabeMethode(std::cout);

  return 0;
}
```

Wie lautet die Ausgabe dieses Programms?

```
#include <iostream>

class Student{
  public:
    std::string name;
    int alter;
    std::string lieblingstier = "Dikdik";

    void ausgabeMethode(std::ostream& os){
        os << name << " " << lieblingstier << " " << alter;
    }

    Student(std::string name, int alter, std::string lieblingstier): name(n
      alter(alter), lieblingstier(lieblingstier)
    {
    }

    Student(std::string name): name(name),alter(0)
    {
    }
};

int main()
{
  Student alexander = Student("Humboldt", 23, "Einhorn");
  alexander.ausgabeMethode(std::cout);

  return 0;
}
```

Destruktoren

Wie lautet die Ausgabe dieses Programms?

```
#include <iostream>
#include <string>

class Auto
{
  public:
    std::string Hersteller;
    int Kilometerstand;
    int PS;

    ~Auto(){
      std::cout << "!";
    }
};

int main()
{
  Auto Beispielauto{"Hyundai", 25000, 76};
  std::cout << Beispielauto.Hersteller << " ";
  std::cout << Beispielauto.Kilometerstand << " " << Beispielauto.PS;
}
```