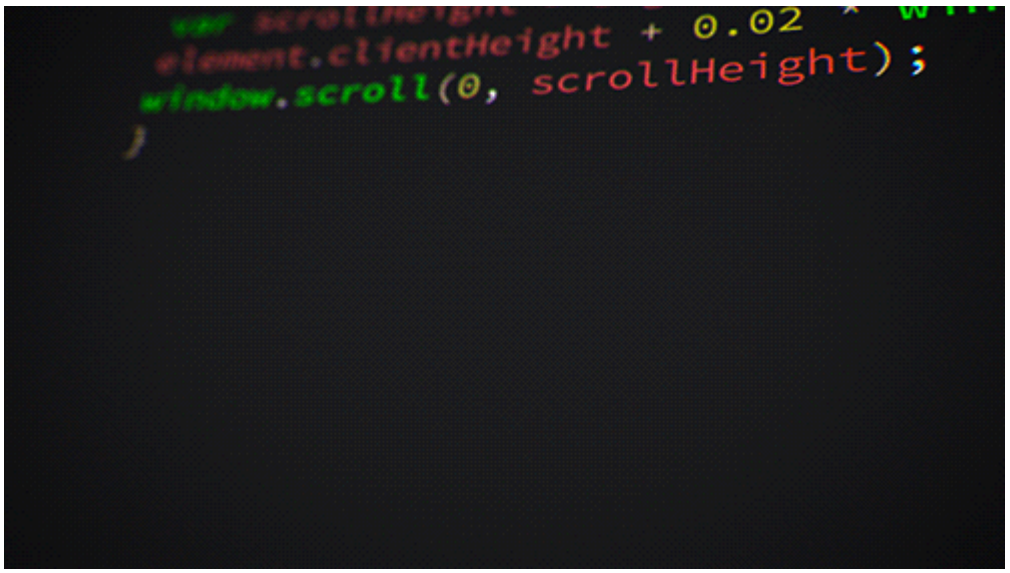
Objektorientierte Programmierung mit C++

Parameter	Kursinformationen
Veranstaltung:	`Prozedurale Programmierung / Einführung in die Informatik / Erhebung, Analyse und Visualisierung digitaler Daten`
Semester	`Wintersemester 2025/26`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Klassen und Objekte`
Link auf Repository:	https://github.com/TUBAF-IfI-LiaScript/VL_EAVD/blob/master/05_OOPI.md
Autoren	Sebastian Zug & André Dietrich & Galina Rudolf & Edda



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 Destruktor erforderlich?
- Was bedeutet das "Überladen" von Funktionen?
- Nach welchen Kriterien werden überladene Funktionen differenziert?

Reflexion Ihrer Fragen / Rückmeldungen

Zur Erinnerung ... Wettstreit zur partizipativen Materialentwicklung mit den Informatikern ...

Preis für das aktivste Auditorium

Format	Informatik Studierende
Verbesserungsvorschlag	2
Fragen	2
generelle Hinweise	0

Was davon kommt denn nun in der Klausur dran?

- Während der Klausur können Sie "alle Hilfsmittel aus Papier" verwenden!
- Im OPAL finden sich Klausurbeispiele.

Beispielhafte Klausuraufgabe

Die Zustimmung (in Prozent) für die Verwendung der künstlichen Intelligenz im Pflegebereich unter der Bevölkerung von Mauritius und Réunion soll vergleichend betrachtet werden. Die Ergebnisse der Umfragen für die Jahre 2010 bis 2020 (je ein Wert pro Jahr) sollen in zwei Arrays erfasst werden (je ein Array pro Insel) und in einem Programm ausgewertet werden.

- *Für beide Inseln soll aus den in Arrays erfassten Werten je ein Mittelwert berechnet werden. Schreiben Sie dazu eine Funktion, die ein Array übergeben bekommt und einen Mittelwert als ein Ergebnis an die main-Funktion zurück liefert. Rufen Sie die Funktion in der main-Funktion für jedes beider Arrays auf und geben Sie die Mittelwerte in der main-Funktion aus.*
- *Schreiben Sie eine weitere Funktion, die die korrespondierenden Werte beider Arrays miteinander vergleicht. Geben Sie für jedes Jahr aus, auf welcher Insel die Zustimmung größer war, bei den gleichen Werte ist eine entsprechende Meldung auszugeben. Rufen Sie die Funktion in der main-Funktion auf.*
- *In der main()-Funktion sind die Werte von der Console einzulesen und in die Arrays zu speichern.*

Denken Sie mal über eine Lösung nach, wie sprechen in der kommenden Veranstaltung darüber.

Rückblick auf Funktionen

Bevor wir in die Objektorientierung einsteigen, wiederholen wir noch einmal die wichtigsten Konzepte von Funktionen mit einem geowissenschaftlichen Beispiel:

FunctionExample.cpp

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

// Funktion zur Klassifikation von Gesteinen nach Korngröße
std::string klassifiziereKorngroesse(double korngroesse_mm) {
    if (korngroesse_mm < 0.063) return "Tonstein";
    else if (korngroesse_mm < 2.0) return "Schluffstein";
    else if (korngroesse_mm < 64.0) return "Sandstein";
    else return "Konglomerat";
}

// Funktion zur Klassifikation nach Porosität
std::string klassifizierePorosität(double porosität_prozent) {
    if (porosität_prozent < 5.0) return "sehr gering";
    else if (porosität_prozent < 15.0) return "gering";
    else if (porosität_prozent < 25.0) return "mittel";
    else return "hoch";
}

// Funktion zur Ausgabe der Gesteinsanalyse
void ausgabeGesteinsdaten(double korngroesse, double porosität,
                          std::string gesteinstyp, std::string porosit
                              {
    std::cout << "=== Gesteinsanalyse ===\n";
    std::cout << "Korngröße: " << korngroesse << " mm -> " << gestein
      << "\n";
    std::cout << "Porosität: " << porosität << "% -> " << porositätsk
      "\n\n";
}

int main() {
    // Messdaten einer Gesteinsprobe
    double korngroesse = 0.8;  // mm
    double porosität = 12.5;   // %

    // Funktionsaufrufe
    std::string gesteinstyp = klassifiziereKorngroesse(korngroesse);
    std::string porositätskl = klassifizierePorosität(porosität);

    ausgabeGesteinsdaten(korngroesse, porosität, gesteinstyp, porosit
      );

    return 0;
}
```

```
=== Gesteinsanalyse ===
Korngröße: 0.8 mm -> Schluffstein
Porosität: 12.5% -> gering
```

Erklären Sie das Codefragment. Warum stecken nicht alle Klassifikationslogiken in der `main`-Methode?

- Funktionen klassifizieren Messdaten nach geologischen Kriterien
- Jede Funktion hat einen klaren Zweck (Korngröße, Porosität, Ausgabe) und kapselt ihre Logik.
- Parameter werden übergeben, Klassifikationen zurückgegeben
- Aber: Die Messdaten und die Funktionen sind getrennt - für jede Probe müssen alle Parameter einzeln verwaltet werden

Hinweis: Die Beispielaufgabe entstand nach einer kurzen Internetrecherche und ist nicht wissenschaftlich geprüft. Geben Sie uns gern Hinweise, wenn die Klassifikationen nicht korrekt sind.

Motivation

Aufgabe: Nehmen wir an, dass Sie eine statistische Untersuchung über einem Datensatz vornehmen und zum Beispiel für Mitarbeiter die Gehaltsentwicklung in Abhängigkeit vom Alter darstellen wollen.

Welche zwei Elemente machen eine solche Untersuchung aus?

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

Wir entwerfen also eine ganze Sammlung von Funktionen wie zum Beispiel `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 Funktion
- viele Variablen
- der inhaltliche Zusammenhang der Daten ist nur schwer zu erkennen

Wir brauchen eine neue Idee, um die Daten zu strukturieren!

Strukturen als Teillösung

Mit Strukturen oder `structs` werden zusammengehörige Variablen unterschiedlicher Datentypen und Bedeutung in einem Konstrukt zusammengefasst. Damit wird für den Entwickler der Zusammenhang deutlich. Die Variablen der Struktur werden als Komponenten (engl. members) bezeichnet.

Beispiele:

```
struct Datum {
  int tag;
  char monat[10];
  int jahr;
};


struct Student {
  int matrikel;
  char name[20];
  char vorname[25];
};  // <- Hier steht ein Semikolon!
```

Deklaration, Definition, Initialisierung und Zugriff

Und wie erzeuge ich Variablen dieses erweiterten Types, wie werden diese initialisiert und wie kann ich auf die einzelnen Komponenten zugreifen?

Das Bespiel zeigt, dass die Definition der Variable unmittelbar nach der `struct`-Definition oder mit einer gesonderten Anweisung mit einer vollständigen, partiellen oder ohne Initialisierung erfolgen kann.

Die nachträgliche Veränderung einzelner Komponenten ist über Zugriff mit Hilfe des Punkt-Operators möglich.

structExample.c

```
#include <iostream>
#include <cstring>

struct Datum                                        // <- Deklaration
{
  int tag;
  char monat[10];
  int jahr;
} geburtstag_1 = {18, "April", 1986};               // <- Initialisie
                                                    //    globale Var
                                          geburtstag_1

void print_date(struct Datum day) {
  std::cout << "Person A wurde am "
            << day.tag << ". "
            << day.monat << " "
            << day.jahr << " geboren."
            << "\n";
}

int main() {
  struct Datum geburtstag_2 = {};                   // <- Initialisie
                                                    //    Variable
                                          geburtstag_2
  geburtstag_2.tag = 13;                            // Zuweisungen
  geburtstag_2.jahr = 1803;
  strcpy(geburtstag_2.monat, "April");

  // Unvollstaendige Initialisierung
  struct Datum geburtstag_3 = {.monat = "September"}; // <- partielle
    Initialisierung

  print_date(geburtstag_1);
  print_date(geburtstag_2);
  print_date(geburtstag_3);
  return 0;
}
```

```
Person A wurde am 18. April 1986 geboren.
Person A wurde am 13. April 1803 geboren.
Person A wurde am 0. September 0 geboren.
Person A wurde am 18. April 1986 geboren.
Person A wurde am 13. April 1803 geboren.
Person A wurde am 0. September 0 geboren.
```

Vergleich von `struct`-Variablen

> Zusammengesetzte Datentypen machen die Anwendung von Vergleichsoperatoren schwieriger, bzw. wir müssen sie selbst definieren.

Im nachfolgenden Beispiel werden zur Überprüfung der Gleichheit die `tag` und `monat` verglichen. Der Vergleich wird dadurch vereinfacht, dass wir für die Repräsentation des Monats ein `int` verwenden. Damit entfällt aufwändigerer Vergleich der `char arrays`.

structExample.c

```
1  #include <iostream>
2  #include <cstring>
3
4  struct Datum                                          // <- Deklaration
5  {
6    int tag;
7    int monat;
8    int jahr;
9  };
10
11 int main() {
12   struct Datum person_1 =  {10, 1, 2013};
13   struct Datum person_2 =  {10, 3, 1956};
14
15   if ((person_1.tag == person_2.tag) && (person_1.monat == person_2.m
16     std::cout << "Oha, der gleiche Geburtstag im Jahr!\n";
17   else
18     std::cout << "Ungleiche Geburtstage!\n";
19   return 0;
20 }
```

```
Ungleiche Geburtstage!
Ungleiche Geburtstage!
```

Arrays von Strukturen

Natürlich lassen sich die beiden erweiterten Datenformate auf der Basis von `struct` und Arrays miteinander kombinieren.

ArraysOfStructs.cpp

```cpp
#include <iostream>
#include <cstring>

struct Datum {
  int tag;
  int monat;
  int jahr;
};

void print_date(struct Datum day) {
  std::cout << "Person A wurde am "
            << day.tag << "."
            << day.monat << "."
            << day.jahr << " geboren."
            << "\n";
}

int main() {

  struct Datum geburtstage[3] = {{18, 4, 1986},
                                 {12, 5, 1820}};

  geburtstage[2].tag = 5;
  geburtstage[2].monat = 9;
  geburtstage[2].jahr = 1905;

  long unsigned array_size = sizeof(geburtstage)/sizeof(struct Datum);

  for (long unsigned int i = 0; i < array_size; i++)
    print_date(geburtstage[i]);

  return 0;
}
```

```
Person A wurde am 18.4.1986 geboren.
Person A wurde am 12.5.1820 geboren.
Person A wurde am 5.9.1905 geboren.
Person A wurde am 18.4.1986 geboren.
Person A wurde am 12.5.1820 geboren.
Person A wurde am 5.9.1905 geboren.
```

Zurück zum Ausgangsproblem

Mit Blick auf unsere Eingangsfragestellung könnte die Lösung also wie folgt aussehen:

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

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

int main() {
  Datum geburtsdatum;
  Datum akt_datum;        // ... und hier wird eine Variable des
                          // Typs angelegt
  geburtsdatum.tag = 12;  // ... und "befüllt"
  geburtsdatum.monat = 3;
  geburtsdatum.jahr = 1920;
  int alter = alterbestimmen(geburtsdatum, akt_datum);
}
```

Hinweis: Die angabe des Schlüsselworts `struct` bei der Deklaration von Variablen des Typs `Datum` ist zwingend in C und optional in C++.

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

Idee von OOP

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);
}
```

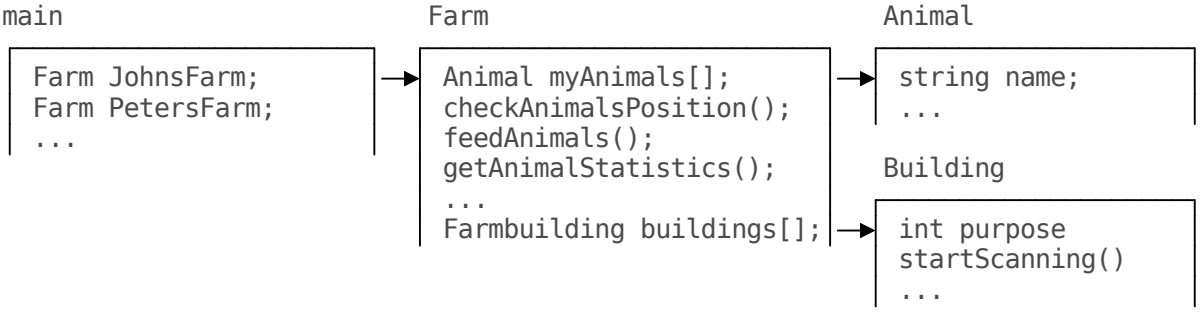
C++ sieht vor als Datentyp für Objekte `struct` - und `class` - Definitionen. Der Unterschied wird später geklärt, vorerst verwenden wird nur die `class` - Definitionen.

```
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 von sich selbst 🙂.

Ein OOP-Computerprogramm kombiniert Objekte und lässt sie interagieren. Viele der am weitesten verbreiteten Programmiersprachen (wie C++, Java, Python usw.) sind Multi-Paradigmen-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

Klasse string

C++ implementiert eine separate Klasse `string`, welche den Umgang mit Zeichenketten im Vergleich zu `char[]` vereinfacht. Beim Anlegen eines Objektes dieser Klasse muss nicht angegeben werden, wie viele Zeichen darin enthalten werden sollen, eine einfache Zuweisung reicht aus.

string.cpp

```
1  #include <iostream>
2  #include <string>
3
4  int main(void) {
5    std::string hallo_msg ="Hallo";
6    std::cout << hallo_msg << " hat eine Länge von " << hallo_msg.length
       " Zeichen.\n";
7    return 0;
8  }
```

```
Hallo hat eine Länge von 5 Zeichen.
Hallo hat eine Länge von 5 Zeichen.
```

> Schauen Sie auch in die Dokumentation der Klasse `string`
> http://www.cplusplus.com/reference/string/string/

Arduino Klassen

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

ArduinoDisplay.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 einen 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* und damit 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 << "\n";
    }
};

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;
}
```

```
12.12.2000
12.3.1920
```

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 << "\n";
    }

    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 << "\n";
  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 << "\n";
    }
};

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

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

    int zumGeburtstagAnrufen() {
      // bestimmt das aktuelle Datum in Sekunden seit 1970
      time_t aktuellerZeitstempel = time(NULL);
      // wandelt den Zeitstempel in eine lokale Zeitstruktur um
      tm* lokaleZeit = localtime(&aktuellerZeitstempel);
      // vergleicht Tag und Monat aus der Struktur mit dem
      // aktuellen Datum
      if ((geburtstag.tag == lokaleZeit->tm_mday) &&
          (geburtstag.monat == (lokaleZeit->tm_mon + 1))) {
        std::cout << "\"Weil heute Dein ... \"\n";
        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!\n";
```

```
49    else
50      std::cout << freundA.name << " hat heute nicht Geburtstag\n";
51
52    return 0;
53  }
```

```
Peter: 1.12.2022
Peter hat heute nicht Geburtstag
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>
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 print() {
        std::cout << tag << "." << monat << "." << jahr << "\n";
      }
};

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

```
31.12.1999
31.12.1999
```

<!--- TODO: Wären die Setter nicht der perfekte Zeitpunkt, um `this` einzuführen/anzuteasern? -->

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  << "\n";
    }
};

void ausgabeFunktion(Student studentA) {
  std::cout << studentA.name << " " << studentA.ort << " " << student
    .alter  << "\n";
}

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

  ausgabeFunktion(bernhard);

  return 0;
}
```

```
Cotta Zillbach 25
Cotta Zillbach 25
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  << "\n";
}

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

```
Cotta Zillbach 25
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

```cpp
#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 << "\n";
    }
};

#endif // !defined(DATUM_H_INCLUDED)
```

Student.h

```cpp
#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 // !defined(STUDENT_H_INCLUDED)
```

Student.cpp

```cpp
#include "Student.h"

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

```

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
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 Variablen und Objekte die gleichen Namen, erlaubt jedoch eine 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 verdict;
      if (sem.passed)
        verdict = " bestanden!";
      else
        verdict = " nicht bestanden";
      std::cout << name << " hat das Seminar " << sem.name << verdict
        "\n";
    }

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

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

Ein Wort zur Ausgabe

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.

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 verdict;
  if (sem.passed)
    verdict = " bestanden!";
  else
    verdict = " nicht bestanden";
  os << name << " hat das Seminar " << sem.name << verdict << "\n";
}

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

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;
```

```
49 }
```

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

Initalisieren/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
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 ergibt)
- 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 expliziten 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

```cpp
#include <iostream>

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

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

    //Student();
};

// Implementierung der Methode


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

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

Dabei sind innerhalb des Konstruktors zwei Schreibweisen möglich:

```cpp
//Initalisierung
Student(std::string name, int alter, std::string ort):
  name(name), alter(alter), 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

```
1  #include <iostream>
2
3  class Student{
4    public:
5      std::string name;
6      int alter;
7      std::string ort; // = "Freiberg";
8
9      void ausgabeMethode(std::ostream& os) {
10       os << name << " " << ort << " " << alter << "\n";
11     }
12
13     Student(std::string name, int alter, std::string ort):
14       name(name), alter(alter), ort(ort)
15     {}
16
17     Student(std::string name):
18       name(name)
19     {}
20 };
21
22 int main() {
23   Student bernhard {"Cotta", 25, "Zillbach"};
24   bernhard.ausgabeMethode(std::cout);
25
26   Student alexander = Student("Humboldt");
27   alexander.ausgabeMethode(std::cout);
28
29   return 0;
30 }
```

```
Cotta Zillbach 25
Humboldt  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.

```
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{a}, ort{o}
{}

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

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

```
End...
Destructing object of type 'Student' with name = 'Maier'
End...
Destructing object of type 'Student' with name = 'Maier'
```

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>

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

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

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

```
area: 12
area: 12
```

Quiz

Definieren von Klassen und Objekten

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 leistung;

    void print() {
      std::cout << hersteller << " - " << leistung << " PS - " << kilometer
        << " Kilometer\n";
    }
};

int main() {
  Auto beispielauto;
  beispielauto.hersteller = "Hyundai";
  beispielauto.kilometerstand = 49564;
  beispielauto.leistung = 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 leistung;

  public:
    void set_Hersteller(std::string _hersteller) {
      hersteller = _hersteller;
    }
    void set_Kilometerstand(int _kilometerstand) {
      kilometerstand = _kilometerstand;
    }
    void set_Leistung(int _leistung) {
      leistung = _leistung;
    }
};

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 leistung;

    void ausgabeMethode();
};

// Implementierung der Methode
void [_____]{
    std::cout << hersteller << " - " << kilometerstand << " km " << leistur
      PS\n";
}

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 << "\n";
}
```

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(name), 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(name), 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 leistung;

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

int main() {
  Auto beispielauto{"Hyundai", 25000, 76};
  std::cout << beispielauto.hersteller << " ";
  std::cout << beispielauto.kilometerstand << " " << beispielauto.leistung;
}
```