Inhalt

- Vorstellung
- Einleitung, Geschichte
- Grundlagen
- Klassen
- Vererbung
- **Zusätzliche Themen**
- Die Standardbibliothek

Zusätzliche Themen

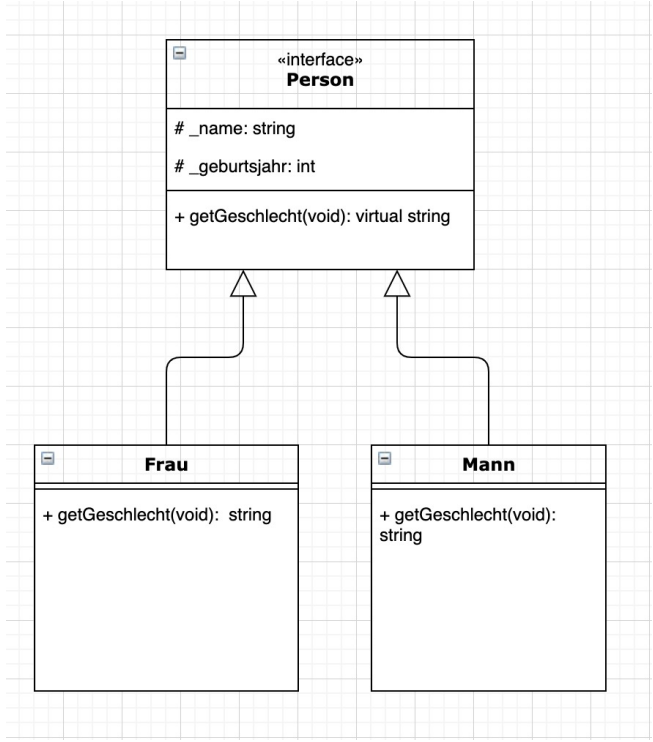
- UML
- Iteratoren – range based loops
- Templates
- Exception handling

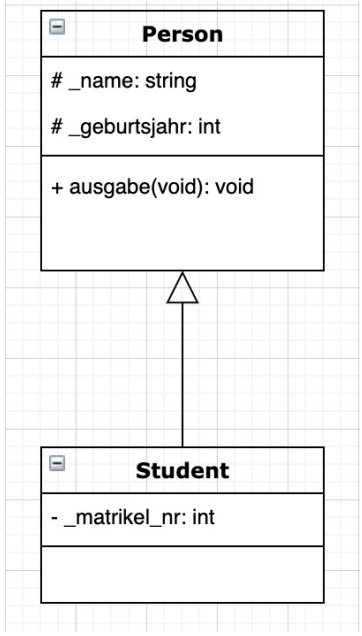
Zusätzliche Themen
UML

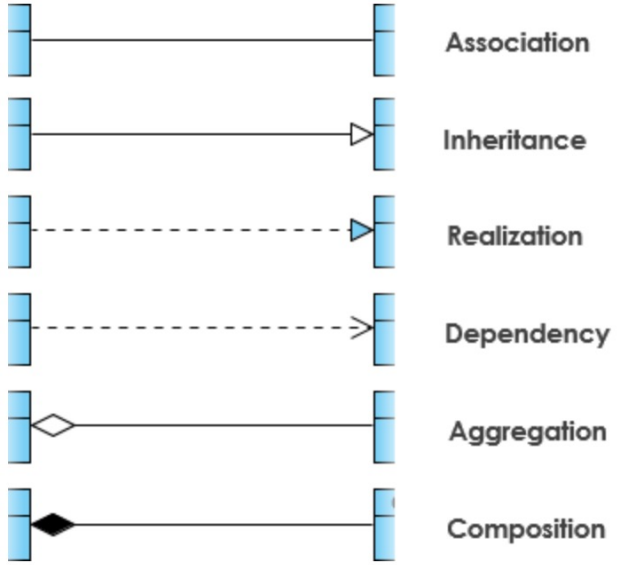
- Die **Unified Modeling Language (UML)** (dt. vereinheitlichte Modellierungssprache) eignet sich dazu Software Strukturen aber auch andere Systeme zu visualisieren, dokumentieren und designen.
- Die Entwicklung von UML begann in den 1990ern und wurde 2005 von ISO standardisiert.
- UML hat 14 Diagrammarten, welche sich grob in zwei Kategorien, Strukturdiagramme (z.B. die Beschreibung SW Klassen) und Verhaltensdiagramme (z.B. Beschreibung von Prozessen oder Programmabläufen), aufteilen lassen.
- Wir werden uns im Rahmen der Vorlesung auf Klassendiagramme zur Beschreibung der Klassenstruktur eines Programms konzentieren.

Zusätzliche Themen
UML

- Wiki Klassendiagramm
- Weitere Erklärungen zur Anwendung von Klassendiagrammen







Zusätzliche Themen
Funktions-Templates

- Templates dienen dem Compiler als sogenannte "*Kopiervorlagen*".
- Funktions-Templates werden durch die Verwendung der Schlüsselwortes *template* definiert. Darauf folgen Typ- oder Nichttyp-Paramter.
- Die Parameter werden durch die Schlüsselwörter *typename* oder *class* definiert.
- Für den ersten Typ-Parameter ist es üblich den Namen *T* zu verwenden.
- Die Parameter können wie gewohnt in der Funktion verwendet werden:

```
template <typename T>
void tausch(T &a, T &b)
{
   T temp = a;
   a = b;
   b = temp;
}
```

```
template <int N>
int nTimes(int n)
{
   return N * n;
}
```

Zusätzliche Themen
Funktions-Templates instanziieren

- Funktions-Templates werden instanziiert indem man die Template-Parameter durch konkrete Werte ersetzt.
- Der Compiler
  - Erzeugt automatisch eine Instanz des Funktions-Templates aufgrund der Argumente
  - Muss die Template-Argumente ableiten können um ein Funktions-Template zu erzeugen.
  - Wenn er die Template-Argumente nicht ableiten kann müssen diese konkret angegeben werden.

```
template <typename T>
void tausch(T &a, T &b)
{ ....

int x = 1, y = 2;
tausch(x,y);
```

```
template <int N>
int nTimes(int n)
{ ...

nTimes<10>(5)
```

Zusätzliche Themen
Funktions-Templates Überladen

- Funktions-Templates können überladen werden.
- Es gelten dabei die folgenden Regeln:
  - Templates unterstützen keine automatische Typkonvertierung.
  - Ist eine freie Funktion eine genauso gute oder bessere Wahl wie ein Funktions-Template, wird die freie Funktion vorgezogen.
  - Durch einen Aufruf der Form ***func<type>(...)*** mit einem Template-Argument type wird explizit ein Funktions-Template aufgerufen.
  - Durch einen Aufruf mit leerer Template-Argumentliste ***func<>(...)*** zieht der Compiler nur Funktions-Templates in Betracht.

Zusätzliche Themen
Klassen-Templates

- Klassen-Templates werden durch die Verwendung der Schlüsselwortes *template* definiert. Darauf folgen Typ- oder Nichttyp-Paramter.
- Die Parameter werden durch die Schlüsselwörter *typename* oder *class* definiert.
- Die Parameter können wie gewohnt in der Funktion verwendet werden

```
template <typename T, int N>
class Array{
  T el[N];
```

Zusätzliche Themen
Klassen-Templates instanziieren

- Klassen-Templates werden instanziiert indem man die Template-Parameter durch konkrete Werte ersetzt.
- Ein Klassen-Template kann im Gegensatz zu einem Funktions-Template seine Argumente nicht automatisch ableiten. ➔ Jedes Template-Argument muss explizit in spitzen Klammern angegeben werden

```
template <typename T>
void tausch(T &a, T &b)
{ ....

int x = 1, y = 2;
tausch(x,y);
```

```
template <typename T, int N>
class Array{
  .....

Array<double, 10> doubleArr;
Array<cKomplex, 10> cKomplexArr;
```

Zusätzliche Themen
Klassen-Templates – Methoden-Templates

- Methoden-Templates sind Funktions-Templates, die in Klassen oder Klassen-Templates verwendet werden.
- Methoden-Templates können innerhalb oder außerhalb der Klasse definiert werden.

Zusätzliche Themen
Iteratoren

- Werden verwendet, um die Elemente eines Containers der Reihe nach zu durchlaufen.
- Werden als Template-Klasse implementiert.
- Verhalten sich ähnlich zu Zeigern:
  - Dateninhalt ausgeben: *p
  - Zeiger auf nächstes Element verschieben: ++p
- Algorithmus zur Ausgabe des Inhalts eines Container auf die Konsole:

```
template <class P>
void ausgabe(P a_beginn, P a_end) {
   for (; a_beginn != a_end; ++a_beginn)
   {
      std::cout << *a_beginn << std::endl;
   }
}
```

Zusätzliche Themen
Iteratoren

- Die Container-Klassen der C++ Standardbibliothek (z.B. vector, map, list, etc) verfügen schon über Iteratoren.
- Die Fähigkeiten des Iterators hängen dabei von der Struktur des Container ab.
- Die Container-Klassen unterstützen normalerweise folgende Methoden:
  - .beginn(): Iterator auf das erste Element des Containers
  - .end(): Iterator der Hinter das letzte Element zeigt
- Weitere Infos

Zusätzliche Themen
Exception Handling

- Werden aus try- und catch-Blöcken zusammengesetzt

```
try {
// Bad File Name
// or missing file handles
}
catch(const BadFileName& e) {
   // handle exception
}
catch(const MissingFileHandle& e) {
   // handle exception
}
```

Zusätzliche Themen
Exception Handling: try

- **try**:
  - Grenzt den Bereich ab in dem eine Ausnahme geworfen werden kann
- **Vorgehen nach einer geworfenen Ausnahme:**
  - Die Programmausführung springt zum passenden catch-Block, der unmittelbar dem try-Block folgt.
  - Wird kein passender catch-Block gefunden, wird der Aufruf Stack gegebenenfalls bis zur main- Funktion zurückverfolgt.
  - Die Funktion terminate ruft den Default-Terminatehandler abort auf.
  - Die Funktion abort bricht den aktuellen Prozess ab.

Zusätzliche Themen
Exception Handling: throw

- **throw**
  - Löst eine Ausnahme aus (throw e)
  - Der Typ der Ausnahme entscheidet, welcher catch-Block ausgeführt wird.
  - Die Ausnahme e wird als Argument an den catch-Block übergeben, um sie bei der Ausnahmebehandlung zu verwenden.
  - Die Methode e.what() der Ausnahme e gibt Informationen zu dieser zurück.
  - In einem catch-Block kann die Ausnahme durch throw wieder ausgelöst werden.
- **Ausnahmen**
  - Sind im Header exception definiert.
  - C++ bietet bereits eigene Ausnahmen an (z.B. std::out_of_range) → Weitere Infos
  - Eigene Ausnahmen sollten vom Typ std::exception abgeleitet werden.

Zusätzliche Themen
Exception Handling: catch

- **catch**
  - Auf einen try-Block folgen eine oder mehrere catch-Blöcke.
  - Die catch-Blöcke geben an, wie bestimmte Typen von Ausnahmen behandelt werden.
  - Die catch-Blöcke werden in der Reihenfolge ihres Auftreten geprüft.
  - Der erste passende catch-Block wird ausgeführt.
  - Eine Ellipse (...) fängt alle Ausnahmen ab (catch(...) {)
- **catch-Blöcke**
  - Sollen vom Speziellen zum Allgemeinen geordnet sein.
  - Sollen dir Argumente als konstante Referenz annehmen (const Exception& e).