

Inhalt

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

Vererbung

- Einfache Vererbung
- Zugriffskontrolle & Vererbungsart
- Vererbung verbieten
- Aufrufreihenfolge & Vererbung von Konstruktoren
- Mehrfache Vererbung
- Abstrakte Basisklassen

Vererbung

Einfache Vererbung

- Person ist die Basisklasse
- Student ist die abgeleitete Klasse
- Jede Instanz von Student kann als Person eingesetzt werden
 - ➔ Jeder Student ist eine Person
- Student erbt alle (public und protected) Eigenschaften (Membervariablen und Methoden) von Person

```
class Person
{
protected: // abgeleitete Funktionen haben Zugriff
    string _name;
    int _geburtsjahr;

public:
    Person(string name, int geburtsjahr)
        : _name(name), _geburtsjahr(geburtsjahr) {}
    void ausgabe()
    {
        cout << "Person: " << _name << endl;
    }
};
```

```
class Student : public Person
{
    int _matrikel_nr;

public:
    Student(string n, int j, int m) : Person(n, j)
    {
        _matrikel_nr = m;
    }
    void ausgabe()
    {
        cout << "Student:\nName: " << _name <<
            "\tmatrikel_nr: " << _matrikel_nr << endl;
    }
};
```

Vererbung

Zugriffskontrolle & Vererbungsart

- Die Vererbungsart zeigt an, ob beim Vererben der Zugriff auf Elemente der Basisklasse eingeschränkt wird.
 - Sie wird vor dem Namen der Basisklasse angegeben.
 - Wie bei Memberdeklarationen gibt es die Schlüsselwörter `public`, `protected` und `private` (Standard-Vererbungsart).
- „friend“-Beziehungen und `private` Variablen oder Methode werden nicht vererbt.

```
class Person { /* ... */ };  
class Student : public Person { /* ... */ };  
class StudentPro : protected Person { /* ... */ };  
class StudentPriv : private Person { /* ... */ }; // := class StudentPriv : Person { /*  
... */ };
```

Vererbung

Zugriffskontrolle & Vererbungsart

Ist ein Element in Person	public	protected	private
... wird es in Student	public	protected	nicht übergeben
... wird es in StudentPro	protected	protected	nicht übergeben
... wird es in StudenPriv	private	private	nicht übergeben

```
class Person { /* ... */ };  
class Student : public Person { /* ... */ };  
class StudentPro : protected Person { /* ... */ };  
class StudentPriv : private Person { /* ... */ }; // := class StudentPriv : Person { /*  
... */ };
```

Vererbung

Vererbung verbieten

- Das Schlüsselwort `final` sorgt dafür, dass ...
 - von einer Basisklasse nicht geerbt werden kann

```
class NoInheritance final {};  
  
class Derived: NoInheritance {}; // Error !
```

- eine Klasse der Endpunkt einer Ableitungshierarchie ist

```
class Base {};  
class LastClass final: Base {};  
class LastLastClass: LastClass {}; // Error !
```

Vererbung

Aufrufreihenfolge Konstruktoren

- Immer, wenn ein Objekt einer abgeleiteten Klasse deklariert wird, wird eine Kette von Konstruktoren ausgeführt.
- Dadurch ist gewährleistet, dass jedes Attribut der Ableitungskette initialisiert wird.
- Die Abarbeitung einer Kette von Konstruktoren beginnt mit der Basisklasse und endet mit der Klasse, von der nicht mehr weiter abgeleitet wird.

```
struct A{};
struct B:A{};
struct C:B{};

C c;           // A -> B -> C
```

Vererbung

Vererbung von Konstruktoren

- Durch die using-Deklaration erbt eine Klasse alle Konstruktoren ihrer direkten Basisklasse
- Der Default-Konstruktor, der Copy- und Move-Konstruktor wird nicht vererbt
- Die abgeleitete Klasse erbt alle Charakteristiken des Konstruktors (public, private, etc.)
- Default-Argumente für Parameter eines Basisklassenkonstruktors werden nicht vererbt
- Konstruktoren mit denselben Parametern wie die abgeleitete Klasse, werden nicht vererbt.
- **Achtung:** Das Vererben von Konstruktoren birgt die Gefahr, dass ein Attribut in der erbenden Klasse nicht initialisiert wird

Vererbung

Virtuelle Methoden

- Um einer Methode in einer Basisklasse in einer abgeleiteten Klasse ein neues Verhalten zuzuordnen, wird sie in der abgeleiteten Klasse überschrieben.
- Zum Überschreiben muss die Methode in der Basisklasse *virtual* deklariert werden. Gerne wird die überschriebene Methode aus Dokumentationszwecken als *virtual* deklariert.

```
class Person {  
    virtual void ausgabe()  
    {  
        cout << "Person: " << _name << endl;  
    }  
}
```

```
class Student : public Person  
{  
    virtual void ausgabe()  
    {  
        cout << "Student: Name: " << _name << "\tmatrikel_nr: " << _matrikel_nr << endl;  
    }  
}
```

Vererbung

Virtuelle Methoden

- Wird eine virtuelle Methode über einen Basisklassenzeiger oder eine Referenz auf ein Objekt einer abgeleiteten Klasse aufgerufen, wird die Methode der abgeleiteten Klasse ausgeführt.

```
Person *p_ptr1;  
Student s1("Frank", 1995, 1234);  
p_ptr1 = &s1;           // Pointer Person zeigt auf Student  
p_ptr1->ausgabe();      // Student::ausgabe()
```

- **Achtung:** Die Entscheidung, welches Objekt verwendet wird, wird zur Laufzeit getroffen. Wird auch als dynamische oder späte Bindung bezeichnet.

Vererbung

Mehrfache Vererbung

- Ähnlich zur einfachen Vererbung, werden die Namen der Basisklassen in einer kommaseparierten Liste angegeben
- Die Mehrfachvererbung folgt den Regeln der Einfachvererbung:
 - Jeder Basisklasse können ihre Zugriffsrechte vorangestellt werden.
 - Klassen besitzen per Default private-; Strukturen public-Zugriffsrecht
- Enthält eine Instanz einer Klasse mehr als eine Instanz einer Basisklasse, ist der Aufruf ihrer Mitglieder mehrdeutig und führt zu einem Fehler ➔ [Diamond-Problem](#)
- Mehrdeutige Aufrufe bei Mehrfachvererbung lassen sich lösen, indem dem mehrdeutigen Mitglied den Namen der Basisklasse vorangestellt wird.

Vererbung

Mehrfache Vererbung: Virtuelle Basisklasse

- Mit einer virtuellen Basisklasse kann das Problem der Mehrfachvererbung behoben werden, bei der Objekte einer abgeleiteten Klasse mehrere Instanzen einer Basisklasse besitzen → [Diamond-Problem](#)
- Durch die Verwendung des Schlüsselwortes *virtual* bei der Vererbung von einer Basisklasse wird diese virtuell.
- Falls für virtuelle Basisklassen kein Default-Konstruktor verwendet wird, muss dieser explizit in der abgeleiteten Klasse aufgerufen werden.

Vererbung

Abstrakte Klassen

- Eine Klasse die über eine oder mehrere virtuelle Methoden verfügt
- Virtuelle Methode:
 - Wird mit `virtuell` deklariert
 - `=0` an die Klassendeklaration gehangen

```
virtual string getGeschlecht() = 0;
```

- Abstrakte Basisklassen Klassen dienen oft als Schnittstelle (Interface) für Klassenhierarchien, da sie konkret vorschreiben was Klassen implementieren müssen

Vererbung

Abstrakte Klassen

- Regeln:
 - Eine Klasse, die rein virtuelle Methoden enthält, kann nicht instanziiert werden.
 - Eine abgeleitete Klasse muss Definitionen für alle rein virtuellen Methoden bereitstellen, um instanziiert werden zu können.
 - Eine rein virtuelle Methode kann in einer Klasse definiert werden, die als rein virtuell deklariert ist.
 - Wenn der Destruktor einer Klasse als rein virtuell deklariert ist, muss trotzdem eine Definition dieser Methode in derselben Klasse angegeben werden.
 - ➔ Beliebtes Verfahren in C++, um eine Klasse zur abstrakten Basisklassen zu erklären

Vererbung Übungsprojekt

- Todo Liste als Gruppenarbeit
- Eventuell GitHub Classroom, link folgt ...

Inhalt

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

Zusätzliche Themen

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

Zusätzliche Themen

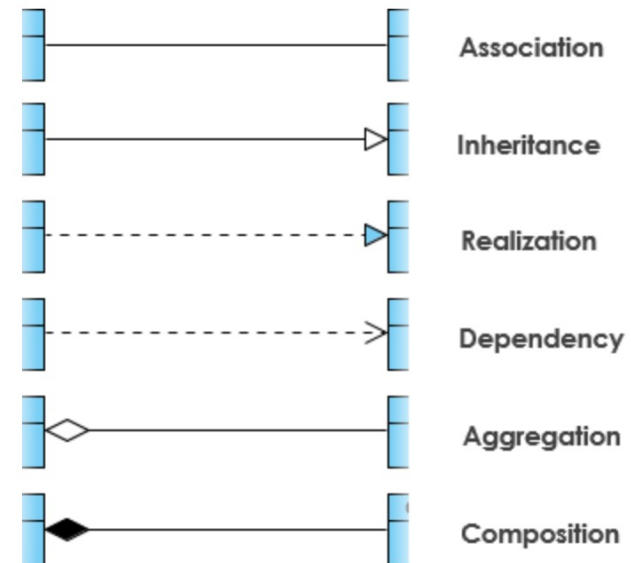
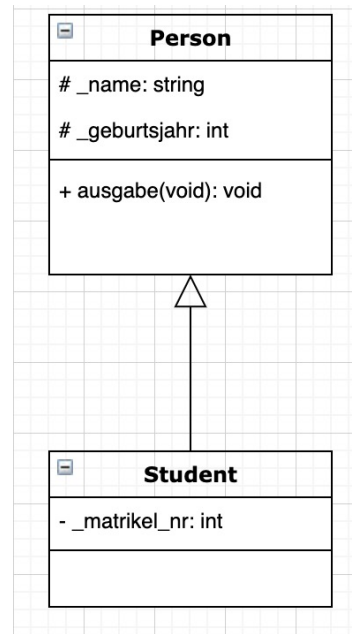
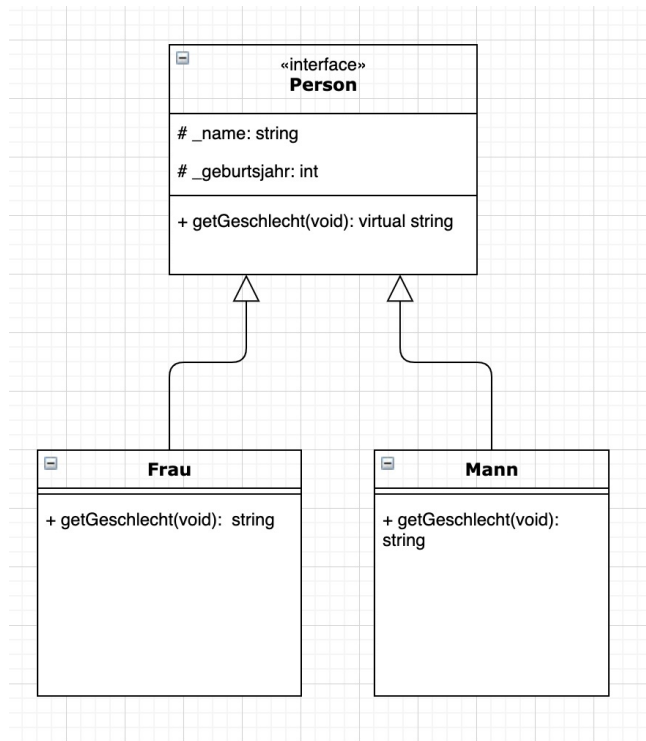
UML

- 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 konzentrieren.

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-Parameter.
- 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 instanziiieren

- 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 instanziiieren

- 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:
 - .begin(): 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 die Argumente als konstante Referenz annehmen (const Exception& e).