

# Programmieren in C++

---

Christian Lang (Lac)

25. Oktober 2019

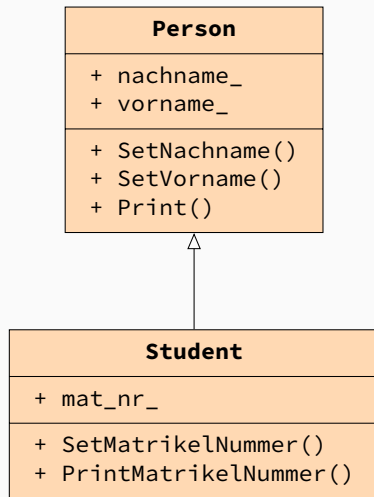
# Vererbung Teil 1

---

- Konzept: Vererbung
- Konstruktoren und Destruktoren in abgeleiteten Klassen
- Typkonvertierung von Pointer auf Klassen
- RTTI
- Typkonvertierung mit Smart-Pointer
- Verdecken von Attributen (shadowing)
- Polymorphie (Vielgestaltigkeit)
- statische und dynamische Bindung
- Überschreiben von Methoden
- Vererbung unterbinden
- Zugriffsrechte

# Konzept: Vererbung

- Superklasse (Basisklasse, Oberklasse, Vaterklasse)
  - Klasse Person mit Eigenschaften
  - Attribute:
    - Nachname
    - Vorname
  - Methoden:
    - SetNachname()
    - SetVorname()
    - Print()
- Subklasse (abgeleitete Klasse, Unterklasse, Sohnklasse)
  - erbt Eigenschaften der Superklasse
  - fügt eigene Eigenschaften hinzu:
    - Matrikelnummer
    - SetMatrikelnummer()
    - PrintMatrikelnummer()



## Beispiel: Klasse Person

```
1  class Person {                                // in h-Datei
2      public:
3          void SetNachname(const std::string& n) { name_ = n; }
4          void SetVorname(const std::string& v) { vorname_ = v; }
5          void Print() const;                    // keine inline-Implementierung
6
7      private:
8          std::string name_;                     // Aggregation: Nachnamen
9          std::string vorname_;                 // Aggregation: Vornamen
10 };
11
12 // in cpp-Datei
13 void Person::Print() const {
14     std::cout << "Nachname: " << name_ << std::endl;
15     std::cout << "Vorname:  " << vorname_ << std::endl;
16 }
```

## Beispiel: Klasse Student

```
1  // in h-Datei: Vererbung: ein Student ist eine Person
2  // und erbt alle Attribute und Methoden der Klasse Person
3  class Student : public Person {
4      public:
5          // neue Methoden der Klasse Student
6          void SetMatrikelNummer(int nr) { mat_nr_ = nr; }
7          void PrintMatrikelNummer() const;
8
9      private:
10         int mat_nr_;
11 };
12
13 // in cpp-Datei
14 void Student::PrintMatrikelNummer() const {
15     std::cout << "MatrikelNummer: " << mat_nr_ << std::endl;
16 }
```

# Verwendung der Klasse Student

```
1  void main () {
2      Person p1;
3      p1.SetNachname("Mueller");
4      p1.SetVorname("Peter");
5      p1.Print();
6
7      Student student;
8      student.SetNachname("Hasler");
9      student.SetVorname("Silvia");
10     student.SetMatrikelnummer(56123);
11     student.Print();           // gibt keine Matrikelnummer aus
12     student.PrintMatrikelnummer();
13
14     Person p2 = student;      // Projektion von Student auf Person
15                               // -> Object-Slicing: nur Attribute von Person werden kopiert
16     p2.Print();               // gibt keine Matrikelnummer aus
17 }
```

- Idee:
  - jeder abgeleitete Konstruktor initialisiert **nur die neuen** Attribute
  - vererbte Attribute werden vom **Basis**-Konstruktor initialisiert
- Umsetzung in C++:
  - Basis-Konstruktor in **Initialisierungs-Liste** aufrufen
  - sonst wird Default-Konstruktor **implizit** aufgerufen

## Aufgaben der Initialisierungsliste (**Reihenfolge** beachten)

- Aufrufen von **Basis**-Konstruktoren oder
- Aufrufen von **anderen** Konstruktoren der eigenen Klasse (Constructor delegation)
- Initialisieren der **eigenen** Attribute



## Beispiel: Konstruktoren

```
1  Person::Person(const string& n, const string& v)
2      : name_(n), vorname_(v) {
3      std::cout << "Person: ctor" << std::endl;
4  }
5
6  Student::Student(const std::string& n, const std::string& v, int m)
7      : Person(n, v), mat_nr_(m) {
8      std::cout << "Student: ctor" << std::endl;
9  }
10
11  // im Testprogramm:
12  Student student("Pixie", "Hollow", 123456);
13
14  // Output:
15  Person: ctor
16  Student: ctor
```

# Destruktoren in abgeleiteten Klassen

- abgeleiteter Destruktor ruft **nach seinem Body** den Basis-Destruktor auf
- dynamische Attribute **müssen** im Destruktor gelöscht werden

```
1  Person::~~Person() {
2      std::cout << "Person: dtor" << std::endl;
3  }
4
5  Student::~~Student() {
6      std::cout << "Student: dtor" << std::endl;
7  }
8
9  // Output:
10     Student: dtor
11     Person: dtor
```

# Typkonvertierung von Pointer auf Klassen

- Typ des Pointers **muss nicht** dem der referenzierten Instanz entsprechen
- nur sinnvoll bei Klassen in gleicher **Klassen-Hierarchie**

```
1  Student* stud = new Student();  
2  
3  // impliziter Up-Cast  
4  Person* pers = stud;  
5  
6  // expliziter Down-Cast  
7  Student* s = static_cast<Student*>(pers);
```

- Problem: `static_cast` führt bei **ungültigem** Down-Cast zu Absturz
- RTTI speichert **genauen Typ** zu jeder Instanz
- RTTI kann **abgeschaltet** werden
- `dynamic_cast` gibt **`nullptr`** bei ungültigem Down-Cast zurück

```
1  Person* pers = new Person();
2  Person* stud = new Student();
3
4  // gültiger Down-Cast
5  Student* s1 = dynamic_cast<Student*>(stud);  // s1 == stud
6
7  // ungültiger Down-Cast
8  Student* s2 = dynamic_cast<Student*>(pers);  // s2 == nullptr
```

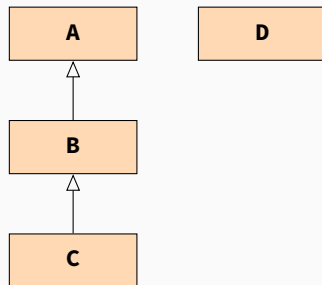
- Up-Cast
  - Konvertierung nach **oben** in der Vererbungs-Hierarchie
  - **implizite** Konvertierung möglich
- Down-Cast
  - Konvertierung nach **unten** in der Vererbungs-Hierarchie
  - nur **explizite** Konvertierung möglich

## Referenzen

- Auch Referenzen sind **Polymorph**
- Können für **Up-/Down-Casts** verwendet werden
- kein `nullptr` sondern **`std::bad_cast`**-Exception
- Syntax schwieriger und nicht so explizit → **nicht verwenden**

# Beispiele: Up- und Down-Casts

```
1 void main() {  
2     // implizite Up-Casts  
3     B* b = new C();  
4     A* a = b;  
5  
6     // gültige, explizite Down-Casts  
7     // ohne und mit Type-Check  
8     C* c1 = static_cast<C*>(b);  
9     C* c2 = dynamic_cast<C*>(a);  
10  
11    // ungültiger Down-Cast  
12    D* d = dynamic_cast<D*>(b);  
13    // d == nullptr  
14 }
```



# Typkonvertierung mit Smart-Pointer

- Funktioniert, bis auf wenige Ausnahmen, **genau gleich**

```
1 void main() {
2     // implizite Up-Casts
3     std::shared_ptr<B> b = std::make_shared<C>();
4     std::shared_ptr<A> a = b;
5
6     // gültige, explizite Down-Casts
7     // ohne und mit Type-Check
8     auto c1 = std::static_pointer_cast<C>(b);
9     auto c2 = std::dynamic_pointer_cast<C>(a);
10
11    // ungültiger Down-Cast
12    auto d = std::dynamic_pointer_cast<D>(b);
13    // d == nullptr
14 }
```

# Verdecken von Attributen (shadowing)

- nicht nur im Kontext von **Scopes**
- sondern auch bei **Vererbung** möglich
- sollte **nicht** verwendet werden

```
1  int a = 42;
2  {
3      double a = 3.14;    // verdeckt a in äusserem Scope
4      std::cout << a << std::endl;
5  }
```

```
1  struct A {
2      int a;
3  };
4  struct B : A {
5      int a;    // B::a verdeckt A::a
6  };
```



# Polymorphie (Vielgestaltigkeit)

- Polymorphie von Operationen
  - gleiche Methodenaufrufe in verschiedenen Klassen führen zu klassenspezifischen Anweisungsfolgen
  - Beispiel: `pers->Print()`; vs. `stud->Print()`;
- Polymorphie von Objekten
  - an die Stelle eines Objektes in einem Programm kann auch ein Objekt einer abgeleiteten Klasse treten
  - ein abgeleitetes Objekt ist polymorph: es kann sich auch als Objekt einer Basisklasse ausgeben
  - Beispiel: ein Student verhält sich wie ein Student, kann sich aber auch wie eine Person verhalten

# statische und dynamische Bindung

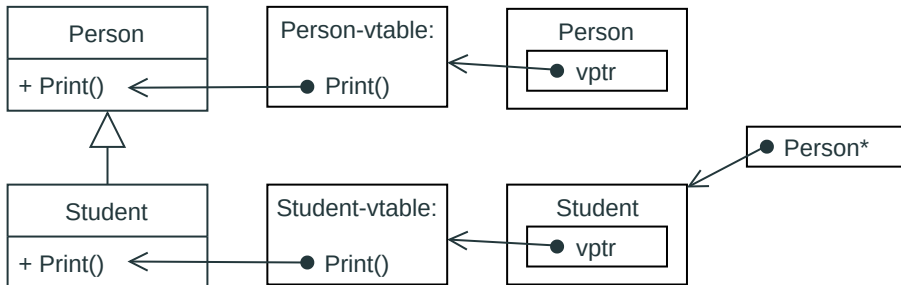
- in C++ kann die Art der Bindung **ausgewählt** werden
- in Java wird die Art der Bindung **automatisch** gewählt

## static Binding

- “früh” → zur **Compiletime**
- kann stark **optimiert** werden
- **default** Verhalten in C++

## dynamic Binding

- “spät” → zur **Runtime**
- erhöht **Wiederverwendbarkeit**
- aktivieren mit **virtual**



# Überschreiben von Methoden

- abgeleitete Klassen können virtuelle Methoden **überschreiben**
- muss gleichen **Namen**, **Return-Typ** und **Signatur** haben
- Pointer-Return-Typ kann auch auf **abgeleitete** Klasse zeigen
- **override** hilft Fehler zu erkennen

```
1  class Person {
2      virtual void Print();
3  };
4
5  class Student : public Person {
6      void Print() override;
7  }; // virtual nur in oberster Klasse nötig
8
9  std::shared_ptr<Person> stud = std::make_shared<Student>();
10 stud->Print(); // Student::Print()
```

## Beispiele: Überschreiben ohne virtual

- ohne virtual **entscheidet** Pointer-Typ
- override würde **Compile-Fehler** auslösen

```
1  class Person {
2      void Print();
3  };
4
5  class Student : public Person {
6      void Print();
7  };
8
9  std::shared_ptr<Person> pers = std::make_shared<Student>();
10 pers->Print(); // Person::Print()
11 std::shared_ptr<Student> stud = std::make_shared<Student>();
12 stud->Print(); // Student::Print()
```

## Beispiele: Überschreiben ohne Pointer

- dynamische Bindung funktioniert nur über **Pointer** oder **Referenzen**
- sonst kommt immer **statische Bindung** zum Zug
- ist mit “normaler” Syntax auch gar **nicht anders möglich**

```
1  Person pers;  
2  pers.Print();    // Person::Print()  
3  Student stud;  
4  stud.Print();    // Student::Print()
```

# Vererbung unterbinden

- Marker `final` für Klassen und Methoden

```
1  class B { ... };
2  class C final : B { ... };
3  class D : C { ... };    // Compile-Fehler
```

```
1  struct B {
2      virtual void f(int) {}
3  };
4  struct C : B {
5      void f(int) final override {}
6  };
7  struct D : C {
8      void f(char) {}    // neue Methode, weil andere Signatur
9      void f(int) {}     // Compile-Fehler
10 };
```

# Zugriffsrechte

Zugriffsrechte der Basisklasse	Basisklasse <b>geerbt</b> als	Zugriffsrechte bei der <b>Benutzung</b> der abgeleiteten Klasse
public protected private	public <sup>1</sup>	public protected No Access <sup>2</sup>
public protected private	protected	protected protected No Access <sup>2</sup>
public protected private	private	private private No Access <sup>2</sup>

<sup>1</sup>Für OOP wird meistens **public**-Inheritance verwendet. Default in Java.

<sup>2</sup>Ausser **friend**-Deklaration in Basisklasse erlaubt den Zugriff explizit.