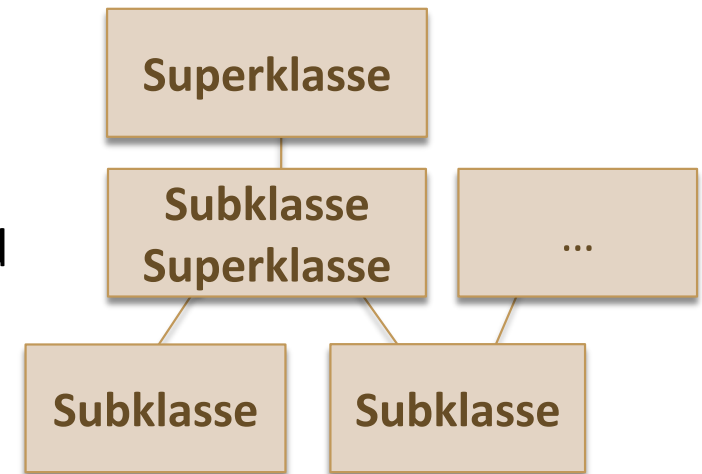


Vererbung & Polymorphie in C++

1. **Grundlegende Eigenschaften**
2. Redefinition
3. Polymorphie
4. Mehrfachvererbung

Vererbung in C++: Grundlagen

- Analog zu Java unterstützt C++ das Konzept der Vererbung:
 - **Superklasse** vererbt Zustand (Attribute) und Verhalten (Methoden) an **Subklasse**.
 - Die Begriffe Super-/Subklasse sind **transitive** Relationen.
- **Objektidentität** durch Speicheradresse definiert.
- **Objektgleichheit** kann durch Überladen von `==` definiert werden.
- Wesentliche Unterschiede zu Java:
 - Es gibt **keine allgemeine Klasse** von der alle Klassen (transitiv) erben.
 - **Mehrfachvererbung**: Klasse kann von mehr als einer Superklasse abgeleitet sein (nicht im Sinne von Interfaces in Java).
 - **Late Binding**: in Java immer; in C++ steuerbar durch Programmierer.



Definition einer Vererbungsbeziehung

```

        private           // analog auch für
class Sub : public       Sup, Sup1 { ... }; // Strukturen
        protected        // möglich

```

- Neben der Vererbung werden dabei **Rechte** definiert, unter denen alle Member der Superklasse in der abgeleiteten Klasse **sichtbar** sein sollen (engl. access specifier):
 - **public**: alle Member von **Sup** behalten ihre Sichtbarkeit in **Sub**
 - **protected**: öffentliche Member von **Sup** werden zu **protected** Members von **Sub**; private Member bleiben privat
 - **private**: alle Member von **Sup** werden zu privaten Members von **Sub**
 - Als Default – wenn ein Spezifizierer nicht angegeben ist (siehe **Sup1**) – wird **private** bei **class** und **public** bei **struct** verwendet.
- Java besitzt diese Rechte-Spezifikation nicht. Dort wird quasi immer mit **public** vererbt.

Redefinition von Attributen/Methoden

Was passiert wenn identische Bezeichner für Member in Super- und Subklasse verwendet werden?

- Bezeichner b (Attribut bzw. Methode) einer Klasse A ist **redefiniert** in Subklasse B gdw. b in A und B vorkommt.
 - Ein redefinierter Member in B **verdeckt** Member in A
 - Bei Methoden (egal ob virtuell oder nicht) ...
 - dürfen Parameter und Rückgabewert voneinander abweichen,
 - ist Kombination mit Überladen möglich → eine Methode von A kann in B mehrfach redefiniert sein (da in B überladen).



Beachte: Die *Redefinition* ist nicht zu verwechseln mit dem *Überladen* von Bezeichnern!

Beispiel: Verdeckung in Kombination mit Überladen

(i)

```
class Parent {  
    public:  
        void test() { cout << "test()" << endl; }  
};  
class Child : public Parent {  
    public:  
        void test(int i) { cout << "test(int)" << endl; }  
};  
int main() {  
    Child c;  c.test(1);  c.test();  
}
```

Völlig unerwartet wenn man das,
was Vererbung sagt, annimmt.



Das Kompilieren mit g++ resultiert in folgender Ausgabe:

In function 'int main()':

error: no matching function for call to `Child::test()``

error: candidates are: void Child::test(int)

Beispiel: Verdeckung in Kombination mit Überladen

(ii)

- Grund: C++ Compiler/Linker geht bei der Namensauflösung wie folgt vor:
 1. Suche die erste Klasse, beginnend mit der eigenen Klasse, in welcher der **Bezeichner** der aufgerufenen Methode vorkommt
 - Im Beispiel: **test** wird in der Klasse **Child** gefunden
 2. Suche die Methode mit exakt gleicher **Signatur** innerhalb dieser Klasse
 - Im Beispiel: **Child** kennt **test(int)**, nicht aber **test()** → Kompilierfehler
- Abhilfe: Jede in Subklassen überladene Methode sollte dort neu definiert werden; üblicherweise unter Verwendung der Methode der Superklasse.

```
class Child : public Parent {  
    public:  
        void test() { Parent::test(); }  
        void test(int i) { cout << "test(int)" << endl; }  
};
```

Vererbung & Polymorphie in C++

1. Grundlegende Eigenschaften
2. Redefinition
3. **Polymorphie**
4. Mehrfachvererbung

Wiederholung: Polymorphie

- Polymorphie (griechisch „Vielgestaltigkeit“)
 - Einer Variable vom Typ eine Superklasse kann irgend ein Objekt vom Typ einer Subklasse zugewiesen sein.
 - Es ist zwischen dem **statischen** und **dynamischen Typ** einer Variablen zu unterscheiden:
 - Statischer Typ: wie deklariert
 - Dynamischer Typ: Typ des Objektes, das der Variable gerade zugewiesen ist
 - Polymorphie: Verhalten je nach dynamischen Typ.

```
LightBulb lb; NeonLight nl;  
Lamp *l = NULL; // statischer Typ: Lamp  
l = &lb;         // dyn. Typ: LightBulb  
l = &nl;         // dyn. Typ: NeonLight
```

sizeof(*l) liefert Wert
des statischen Typs.



Bild aus: B. Lahres, G. Rayman: Objektorientierte Programmierung.

Beispiel zur Polymorphie

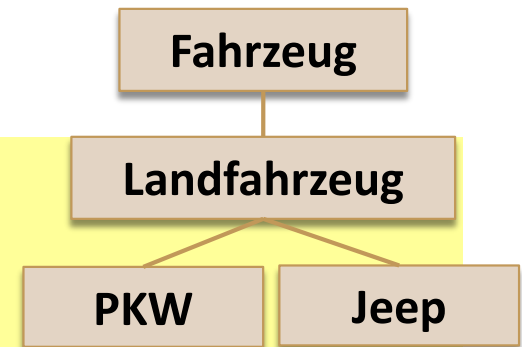
```
class Fahrzeug {
    public:
        virtual void mag() = 0;
};

class Landfahrzeug : public Fahrzeug {
    public:
        virtual void mag() {cout << "festen Boden" << endl;}
        void allrad() {cout << "mal so, mal so" << endl;}
};

class PKW : public Landfahrzeug {
    public:
        virtual void mag() {cout << "Strassen" << endl;}
};

class Jeep : public Landfahrzeug {
    public:
        virtual void mag() {cout << "offroad" << endl;}
        void allrad() {cout << "ja" << endl;}
};
```

Rein virtuelle Methode



Redefinierte Methode

Polymorphie in C++

(i)

- Bei Methodenzugriff über „Standard“-Variable wird immer die **Methode des statischen Typs** verwendet.
 - In diesem Fall **kein** polymorphes Verhalten!

```
Jeep j;  
j.mag();           // offroad  
j.allrad();        // ja  
Landfahrzeug lf = j;  
lf.mag();          // festen Boden  
lf.allrad();       // mal so, mal so
```

Polymorphie in C++

(ii)

- Bei Methodenzugriff über Zeiger oder Referenzen wird:
 - die **Methode des dynamischen Typs** verwendet, insofern die entsprechende Methode als **virtual** definiert wurde;
 - andernfalls wird die Methode des statischen Typs verwendet.

```
Jeep j;  
j.mag();           // offroad  
j.allrad();        // ja  
Landfahrzeug* lf = &j;  
lf->mag();          // offroad  
lf->allrad();       // mal so, mal so
```

Ergo: man kann gleiches Verhalten zu Java durch virtuelle Methoden und Verwendung von Zeigern bzw. Zeiger-Referenzen erreichen.

Early Binding versus Late Binding

- **Early binding** a.k.a. **static dispatch** (statisches Binden):
Es ist bereits zum Kompilierzeitpunkt bekannt, welche Methode ausgeführt wird (statischer Typ): `lf.mag()`;
- **Late binding** a.k.a. **dynamic dispatch** (dynamisches Binden):
Es ist erst zur Laufzeit bekannt (über den dynamischen Typ), welche Methode aufgerufen wird: `lf->mag()`;
 - Wird intern durch indirekte Adressierung implementiert: Zeiger auf (Hash-)Tabelle, die die Einsprungsadresse aller überschreibbaren Methoden enthält. Durch notwendigen Lookup in Tabelle besteht ein **geringer Overhead beim Aufruf virtueller Methoden**.



Virtuelle Methoden & abstrakte Klassen

Regeln zu virtuellen Methoden:

- Ist eine Methode in einer Superklasse nicht virtuell, dann kann man sie in Subklasse nicht virtuell machen.
- Eine einmal als **virtuell deklarierte Methode bleibt immer virtuell**, auch wenn in Subklassen das Schlüsselwort **virtual** nicht nochmals angegeben wird.

Abstrakte Klassen in C++:

- Können nicht instanziiert werden.
- Kriterium: **mindestens eine rein virtuelle Methode**, d.h. eine Methode die mit **virtual=0** gekennzeichnet ist; rein virtuelle Methoden haben keinen Body; (siehe **mag()** auf Folie 9).

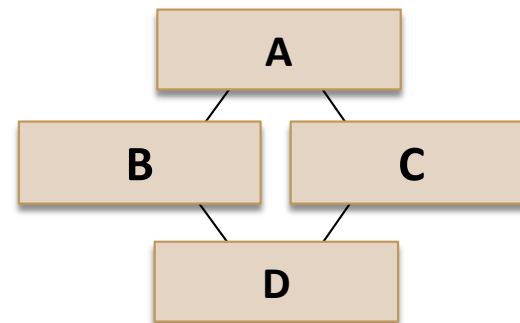
K06

Vererbung & Polymorphie in C++

1. Grundlegende Eigenschaften
2. Redefinition
3. Polymorphie
4. **Mehrfachvererbung**

Mehrfachvererbung

- Die Subklasse **erbt die Eigenschaften aller** Superklassen
- Prinzipiell ist Mehrfachvererbung problemlos, es sei denn:
 - Zwei oder mehr Superklassen haben eine **Methode mit identischer Signatur** oder ein **Attribut mit dem selben Name**
 - Zwei oder mehr Superklassen haben die **selbe Superklasse**



Beispiel: identische Methoden (i)

```
class Auto {  
    public:  
        void leistung() {cout << "90 PS" << endl;}  
};  
class Schiff {  
    public:  
        void tiefgang() {cout << "> 0.5m" << endl;}  
};  
class Amphibienfahrzeug: public Auto, public Schiff {};  
int main() {  
    Amphibienfahrzeug amph;  
    amph.leistung();    // 90 PS  
    amph.tiefgang();    // > 0.5m  
}
```

Schiff

Auto

Amphibienfahrzeug

Beispiel: identische Methoden (ii)

```
class Auto {
    public:
        void leistung() {cout << "90 PS" << endl;}
};
class Schiff {
    public:
        void tiefgang() {cout << "> 0.5m" << endl;}
        void leistung() {cout << "500 PS" << endl;}
};
class Amphibienfahrzeug: public Auto, public Schiff {};
int main() {
    Amphibienfahrzeug amph;
    amph.leistung();           // Kompiler-
    amph.Auto::leistung();     // 90 PS
    amph.Schiff::leistung();   // 500 PS
}
```

Schiff

Auto

Amphibienfahrzeug

Werden mehrere Methoden (oder Attribute) mit der gleichen Signatur geerbt, so muss die Methode beim Aufruf **eindeutig** spezifiziert werden, also: **Klasse::**

Beispiel: 2x selbe Superklasse

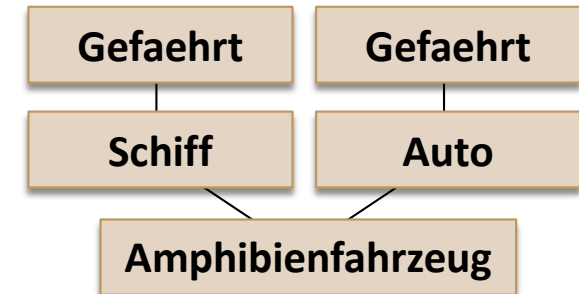
```
class Gefaehrt {
protected:
    int i; // Anzahl der Inhaber
public:
    Gefaehrt() : i(0) {};
    void verkaufen() {i++};
};

class Auto : public Gefaehrt {
};

class Schiff: public Gefaehrt {
};

class Amphibienfahrzeug :
    public Auto, public Schiff {};

```



```
int main() {
    Amphibienfahrzeug af;
    af.verkaufen(); // Kompilerfehler!
    af.Auto::verkaufen();
    af.Schiff::verkaufen();

    Gefaehrt* gf = &af; // K'fehler!
    Gefaehrt* at =
        static_cast<Auto*>(&af); // OK
}

```

Auch hier gilt: es muss eindeutig spezifiziert werden!

Virtuelle Mehrfachvererbung

- Bewirkt, dass eine gemeinsame Superklasse nur einmal vererbt wird.
- Dazu wird das Keyword **virtual** verwendet (bei der Vererbungsdeklaration, nicht bei einer Methode!).
- Qualifizierte Bezeichnung der verwendeten Methoden und Attribute der gemeinsamen Superklasse(n) nicht mehr nötig (in unserem Beispiel: **Gefahrt**).
- Auf Einfachvererbung oder Mehrfachvererbung ohne gemeinsame Basisklasse hat das Keyword **virtual** keinen Einfluss.

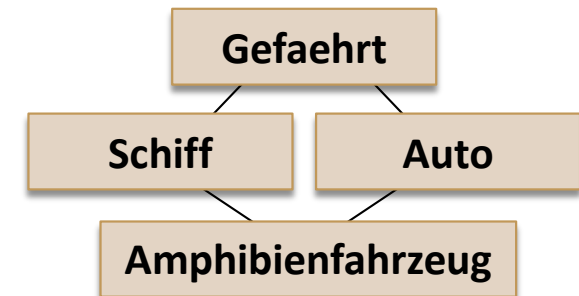
Beispiel: Virtuelle Mehrfachvererbung

```
class Gefaehrt {
protected:
    int i; // Anzahl der Inhaber
public:
    Gefaehrt() : i(0) {};
    void verkaufen() {i++;}
};

class Auto : public virtual Gefaehrt {
};

class Schiff: public virtual Gefaehrt {
};

class Amphibienfahrzeug :
    public Auto, public Schiff{};
```



Eine **virtuelle Superklasse**

- wirkt sich erst bei Mehrfachvererbung aus
- bleibt auch bei weiteren Ableitungen virtuell

```
int main() {
    Amphibienfahrzeug af;
    af.verkaufen(); // jetzt möglich
}
```