Vererbung

- Der Konstruktor einer abgeleiteten Klasse
- Der Destruktor einer abgeleiteten Klasse
- Zugriffsschutz
- Überschreiben von Funktionen in abgeleiteten Klassen
- Mehrfachvererbung
- Typbeziehung zwischen Oberklasse und abgeleiteter Klasse
- Speichern von Basisklassen-Zeigern

Vererbung

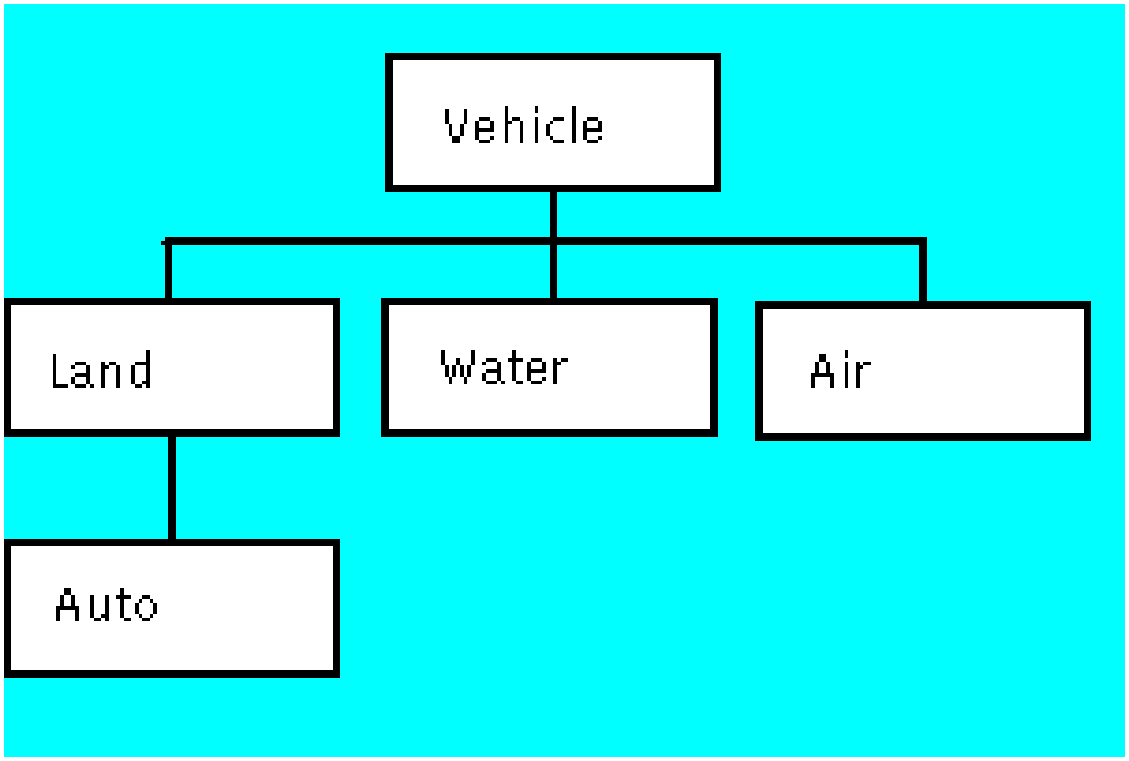
Der Vererbungsmechanismus zeichnet sich aus durch

- Eigenschaften, die einer Menge von Dingen gemeinsam sind, können als verallgemeinertes Konzept betrachtet werden, das besonders behandelt wird.

- Es gibt geringe Unterschiede zwischen diesen Dingen.

- Die Vererbung ist hierarchisch organisiert.

Beispiel: Klassifizierung von Transportmitteln



Vererbung

Die vererbende Klasse heißt **Oberklasse**.

Die erbende Klasse heißt **Unterklasse** oder **abgeleitete Klasse**.

Die oberste Klasse wird auch **Basisklasse** genannt.

Die Vererbung beschreibt eine **ist-ein**-Beziehung.

Die Vererbung ist eine **gerichtete** Beziehung.

Eine Oberklasse ist die **Abstraktion** oder **Generalisierung** von ähnlichen Eigenschaften und Verhaltensweisen der Unterklassen.

Die Unterklasse ist eine **Spezialisierung** der Oberklasse.

Die Unterklasse **erbt** von der Oberklasse

- die Eigenschaften (die Daten) und
- das Verhalten (die Methoden).

In einer Unterklasse brauchen nur die **Abweichungen** von der Oberklasse beschrieben zu werden. Alles andere kann **wiederverwendet** werden, weil es in der Oberklasse bereits vorliegt.

Vererbung

Komposition: `Land` **enthält ein** `Vehicle`.

```
class Vehicle
{
  public:
    Vehicle();
    Vehicle(int wt);

    int getweight() const;
    void setweight(int wt);

  private:
    int weight;
};

class Land
{
  public:
    void setweight(int wt);
  private:
    Vehicle v;      // Komposition
};

// keine neue Funktionalität, lediglich
// Aufruf von Vehicle::setweight()
void Land::setweight(int wt)
{
  v.setweight(wt);
}
```

Vererbung

Vererbung: Land **ist ein** Vehicle.

```
class Land : public Vehicle
{
  public:
    Land();
    Land(int wt, int sp);

    void setspeed(int sp);
    int getspeed() const;

  private:
    int speed;
};


Land veh(1200, 145);

int main()
{
  cout << "Vehicle weighs "
       << veh.getweight()
       << endl
       << "Speed is "
       << veh.getspeed()
       << endl;
}
```

Vererbung

Wenn eine Unterklasse `U` von einer Oberklasse `O` erbt, dann bedeutet das:

- Jedes Objekt u vom Typ `U` enthält ein (anonymes) Objekt vom Typ `O` (Subobjekt genannt).
  Dieses Subobjekt wird noch vor der Erzeugung von u durch impliziten Aufruf des Oberklassen-konstruktors gebildet.

- Jede Elementfunktion von `O` kann auf ein Objekt des Typs `U` angewandt werden, sofern die Elementfunktion öffentlich zugänglich (`public`) ist.

- Die Klasse `U` kann Erweiterungen der Daten und zusätzliche Methoden enthalten, die keinen Bezug zur Oberklasse haben.

Vererbung

Klassenhierarchien: Eine abgeleitete Klasse kann selbst wieder eine Oberklasse sein.

```
class Auto : public Land
{
  public:
    // Standardkonstruktor
    Auto();

    // Allgemeiner Konstruktor
    Auto(int wt, int sp, char const *nm);

    // Kopierkonstruktor
    Auto(Auto const &other);

    // Destruktor
    ~Auto();

    // Wertzuweisung
    Auto &operator=(Auto const &other);

    // Zugriffsfunktionen
    char const *getname() const;
    void setname(char const *nm);

  private:
    // Daten
    char const *name;
};
```

Vererbung

Der Konstruktor einer abgeleiteten Klasse

Konstruktor der Klasse `Land` (Version 1):

```
Land::Land (int wt, int sp)
{
  setweight(wt);
  setspeed(sp);
}
```

impliziter Aufruf des Standardkonstruktors der Basisklasse `Vehicle` (evtl. mit Initialisierung von `weight`); anschließend explizite Initialisierung der Datenelemente `weight` und `speed`

Konstruktor der Klasse `Land` (Version 2):

```
Land::Land(int wt, int sp)
  : Vehicle(wt)
{
  setspeed(sp);
}
```

expliziter Aufruf des allgemeinen Konstruktors der Basisklasse `Vehicle`

Vererbung

Der Destruktor einer abgeleiteten Klasse

impliziter Aufruf des Destruktors der Basisklasse
im Destruktor der abgeleiteten Klasse

```
#include <iostream>
using namespace std;

class Base
{
  public:
    Base() { cout << "const. Base" << endl; }
    ~Base() { cout << "dest. Base" << endl; }
};

class Derived : public Base
{
  public:
    Derived() { cout << "const. Derived" << endl; }
    ~Derived() { cout << "dest. Derived" << endl; }
};

int main()
{
  Derived derived;
}
```

Ausgabe:

Vererbung

Zugriffsschutz

Unter Zugriffsschutz ist die Abstufung von Zugriffsrechten auf Daten und Elementfunktionen zu verstehen.

Bisher sind zwei Fälle bekannt:

- **`public`**
  Elemente und Methoden unterliegen keiner Zugriffsbeschränkung

- **`private`**
  Elemente und Methoden sind ausschließlich innerhalb der Klasse zugreifbar, sowie für `friend`-Klassen und -Funktionen.

Die Zugriffsspezifizierer `public` und `private` gelten genauso in einer Vererbungshierarchie.

Es gibt einen weiteren Zugriffsspezifizierer:

- **`protected`**
  Elemente und Methoden sind in der eigenen und allen `public` abgeleiteten Klassen zugreifbar, nicht aber in anderen Klassen oder außerhalb der Klasse.

Vererbung

Zugriffsschutz

Für die Vererbung der Zugriffsrechte bei **`public`**-Vererbung gelten folgende Regeln:

Zugriffsrecht in der Basisklasse	Zugriffsrecht in einer abgeleiteten Klasse
`private`	kein Zugriff
`protected`	`protected`
`public`	`public`

Vererbung

Überschreiben von Funktionen in abgeleiteten Klassen

```
class Truck : public Auto
{
  public:
    // Konstruktoren
    Truck();
    Truck(int engine_wt, int sp,
          char const *nm, int trailer_wt);

    // Zugriffsfunktionen
    void setweight(int engine_wt,
                   int trailer_wt);
    int getweight() const;

  private:
    // Daten
    int trailer_weight;
};

// Bsp.: der allgemeine Konstruktor
Truck::Truck(int engine_wt, int sp,
             char const *nm, int trailer_wt)
  : Auto(engine_wt, sp, nm)
{
  trailer_weight = trailer_wt;
}
```

Überschreiben von Funktionen in abgeleiteten Klassen

```
void Truck::setweight
  (int engine_wt, int trailer_wt)
{
  trailer_weight = trailer_wt;
  Auto::setweight(engine_wt);
  // hier wird Auto::setweight() verwendet
}
```

`Truck::setweight(int, int)` überdeckt `Auto::setweight(int)`
Daher müssen wir den Gültigkeitsbereich explizit angeben.

```
int Truck::getweight() const
{
  return
    Auto::getweight() + trailer_weight;
  // Gewicht des Motorwagens plus
  // Gewicht des Anhängers
}
```

Auch im Rumpf von `getweight()` müssen wir den Gültigkeitsbereich explizit angeben.

```
return getweight() + trailer_weight;
```

würde zu einer nicht abbrechenden Rekursion führen.

Vererbung

Mehrfachvererbung

```
class Engine
{
  public:
    // Konstruktoren
    Engine();
    Engine(char const *serial_nr, int power,
           char const *fuel_type);

    // Die Klasse enthält Zeiger!
    ~Engine();
    Engine(Engine const &other);
    Engine &operator=(Engine const &other);

    // Zugriffsfunktionen
    void setserial(char const *serial_nr);
    void setpower(int power);
    void setfueltype(char const *type);

    char const *getserial() const;
    int getpower() const;
    char const *getfueltype() const;

  private:
    // Daten
    char const *serial_number, *fuel_type;
    int power;
};
```

Mehrfachvererbung

```
class MotorCar : public Auto, public Engine
{
  public:
    // Konstruktoren
    MotorCar();
    MotorCar(int wt, int sp, char const *nm,
             char const *ser, int pow,
             char const *fuel);
};


MotorCar::MotorCar
  (int wt, int sp, char const *nm,
   char const *ser, int pow, char const *fuel)
  : Engine (ser, pow, fuel), Auto (wt, sp, nm)
{}
```

Mehrfachvererbung

Die Mehrfachvererbung ist bei diesem Beispiel ein wenig seltsam:

- Ein `MotorCar` **ist ein** `Auto`, aber
- ein `MotorCar` *ist keine* `Engine`.

- Ein `MotorCar` **hat eine** Engine

wäre die korrekte Beziehung: also Komposition statt Vererbung.

Komposition führt jedoch zum Duplizieren von Code:
Alle Zugriffsfunktionen von `Engine` müssen für `MotorCar` implementiert werden.

```
void MotorCar::setpower(int pow)
{
  engine.setpower(pow);
}

int MotorCar::getpower() const
{
  return engine.getpower();
}

// usw., usw., wiederholt für
// set-/getserial() und set-/getfueltype()
```

Vererbung

Typbeziehung zwischen Oberklasse und abgeleiteter Klasse

Eine abgeleitete Klasse kann als *Subtyp* der Oberklasse aufgefaßt werden.

Ein Objekt der abgeleiteten Klasse ist zuweisungskompatibel zu einem Objekt der Oberklasse.

```
Vehicle v(900);
// Fahrzeug mit einem Gewicht von 900 kg

Auto a(1200, 130, "Ford");
// Automobil mit Gewicht 1200 kg,
// Höchstgeschwindigkeit 130 km/h,
// Marke Ford

v = a;
```

Ein `Auto` ist auch ein `Vehicle`, daher ist die Zuweisung erlaubt. Ein `Vehicle` hat weder `speed`- noch `name`-Feld, diese werden daher nicht zugewiesen.

```
a = v;
```

Die Zuweisung eines Objekts der Oberklasse an ein Objekt der Unterklasse ist verboten. Welche Werte sollen den Datenelementen `speed` und `name` zugewiesen werden?

Typbeziehung zwischen Oberklasse und abgeleiteter Klasse

Die Konversion eines Objekts einer Oberklasse in ein Objekt einer abgeleiteten Klasse kann explizit definiert werden.

Für die Wertzuweisung

```
a = v;
```

benötigt die Klasse `Auto` eine überladene Wertzuweisung, die ein `Vehicle`-Objekt als Argument hat.

```
Auto &Auto::operator=(Vehicle const &veh)
{
  setweight (veh.getweight());

  // Hier sollte der Code eingefügt werden,
  // der die restlichen Datenelemente
  // initialisiert.

}
```

Typbeziehung zwischen Oberklasse und abgeleiteter Klasse

```
Land land(1200, 130);
Auto auto1(500, 75, "Daf");
Truck truck(2600, 120, "Mercedes", 6000);
Vehicle *vp;
```

Wir können `vp` die Adressen der Objekte abgeleiteter Klassen zuweisen.

```
vp = &land;
vp = &auto1;
vp = &truck;
```

Bei der Zuweisung findet eine implizite Konversion vom Typ der abgeleiteten Klasse nach `Vehicle` statt, weil `vp` als Zeiger auf `Vehicle` vereinbart ist.

Bei Verwendung von `vp` können nur die Zugriffsfunktionen aufgerufen werden, die das Datenelement `weight` manipulieren.

Nach der Zuweisung

```
vp = &truck;
```

liefert

```
vp->getweight()
```

nur das Gewicht des Motorwagens (2600) statt des Gesamtgewichts (8600)!

Typbeziehung zwischen Oberklasse und abgeleiteter Klasse

Vermeidung der impliziten Konversion durch eine explizite Typanpassung

```
Truck truck;
Vehicle *vp;

vp = &truck;
// vp zeigt jetzt auf ein Objekt vom Typ Truck

Truck *trp;

trp = (Truck *) vp;
cout << "Make: " << trp->getname() << endl;
```

Vererbung

Speichern von Basisklassen-Zeigern

Klassen, die Objekte anderer Klassen speichern sollen, werden Behälterklassen (engl. *container class*) genannt.

```
class VStorage
{
  public:
    VStorage();
    VStorage(VStorage const &other);
    ~VStorage();
    VStorage &operator=(VStorage const &other);

    // Vehicle * speichern
    void add(Vehicle const *vp);
    // ersten Vehicle * aus dem Speicher holen
    Vehicle const *getfirst();
    // nächsten Vehicle * aus dem Speicher holen
    Vehicle const *getnext();

  private:
    // Daten
    Vehicle const **storage;
    int nstored, current;
};
```

Speichern von Basisklassen-Zeigern

Verwendung der Klasse VStorage:

```
Land land(200, 20);
// weight 200, speed 20
Auto auto1(1200, 130, "Ford");
// weight 1200, speed 130, make Ford
VStorage garage;
// der Behälter

garage.add(&land);
// in den Behälter stecken
garage.add(&auto1);

Vehicle const *anyp;
int total_wt = 0;

for (anyp = garage.getfirst();
     anyp;
     anyp = garage.getnext())
  total_wt += anyp->getweight();

cout << "Total weight: " << total_wt << endl;
```

Speichern von Basisklassen-Zeigern

Durch die Vererbung und die Konversion zwischen Objekten der Basisklasse und Objekten abgeleiteter Klassen ermöglicht C++ die Verarbeitung aller abgeleiteten Typen in einer generischen Klasse.

`VStorage` könnte auch benutzt werden, um Objekte zu speichern, die zukünftig noch von `Vehicle` abgeleitet werden. `VStorage` verwendet ein *Protokoll*, das von `Vehicle` definiert wird und obligatorisch ist für alle abgeleiteten Klassen.

`VStorage` hat nur einen Nachteil.

Wenn wir ein `Truck`-Objekt in den Behälter stecken, dann gibt der folgende Code

```
Vehicle *any;
VStorage garage;

any = garage.getnext();
cout << any->getweight() << endl;
```

nicht das Gesamtgewicht von Motorwagen und Anhänger aus.

```
any->getweight()
```

liefert lediglich das Gewicht, das im `Vehicle`-Anteil des LKWs gespeichert ist.