Polymorphismus

- Virtuelle Funktionen
    - Auswirkungen des Polymorphismus auf die Programm-Entwicklung
    - Implementierung des Polymorphismus

- Rein virtuelle Funktionen

- Virtuelle Destruktoren

- Virtuelle Funktionen und Mehrfachvererbung
    - Mehrdeutigkeiten bei der Mehrfachvererbung
    - Virtuelle Basisklassen

Polymorphismus

Polymorphismus bedeutet **Vielgestaltigkeit**.

In der objektorientierten Programmierung beschreibt der Begriff Polymorphismus die Fähigkeit einer Variablen, *zur Laufzeit* eines Programms auf Objekte verschiedener Typen zu verweisen.

Erst *zur Laufzeit* eines Programms wird die zu dem jeweiligen Objekt passende Realisierung einer Operation ermittelt.

In C++ gilt die Einschränkung, dass die Objekte abgeleiteten Klassen zuzuordnen sind.

Wenn die Entscheidung, welche Funktion aufzurufen ist, erst *zur Laufzeit* getroffen wird, dann bezeichnet man dies als **dynamisches** oder **spätes Binden**.

Wenn schon *zur Übersetzungszeit* (also vom Compiler) festgelegt wird, welche Funktion aufzurufen ist, dann wird dies **statisches** oder **frühes Binden** genannt.

Eine zur Laufzeit ausgewählte Elementfunktion heißt **virtuelle Funktion**.

Polymorphismus

Virtuelle Funktionen

Wenn eine Elementfunktion über einen Zeiger (oder eine Referenz) auf ein Objekt aufgerufen wird, dann bestimmt normalerweise der Typ des Zeigers (oder der Referenz), welche Funktion aufgerufen wird (statisches Binden).

Dynamisches Binden wird in C++ durch *virtuelle Funktionen* erreicht.

Wenn eine Elementfunktion in einer Basisklasse als `virtual` deklariert wird, dann ist sie in allen abgeleiteten Klassen `virtual`.
Das Schlüsselwort `virtual` braucht in den abgeleiteten Klassen nicht wiederholt zu werden.

Ein Unterschied im Verhalten eines Objekts durch Aufruf einer virtuellen Funktion im Vergleich zu nicht-virtuellen Funktionen zeigt sich nur, wenn der Aufruf über

- **Basisklassenzeiger** oder
- **Basisklassenreferenzen**

geschieht, statt über den Objektnamen.

Virtuelle Funktionen

Beispiel:

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

    // Zugriffsfunktionen jetzt virtuell
    virtual int getweight() const;
    virtual void setweight(int wt);

  private:
    int weight;
};

// getweight()-Funktion der Klasse Vehicle
int Vehicle::getweight() const
{
  return weight;
}

class Land : public Vehicle
{
  // ...
};
```

Virtuelle Funktionen

```
class Auto : public Land
{
  // ...
};

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

    // Zugriffsfunktionen zum Schreiben
    // zweier weight Datenelemente
    void setweight(int engine_wt,
                   int trailer_wt);
    // ... und zum Lesen des Gesamtgewichts
    int getweight() const;

  private:
    int trailer_weight;
};

// getweight()-Funktion der Klasse Truck
int Truck::getweight() const
{
  return Auto::getweight() + trailer_weight;
}
```

Virtuelle Funktionen

Auswirkung des dynamischen Bindens

```
Vehicle v(1200);
// vehicle with weight 1200
Truck t(6000, 115, "Scania", 15000);
// truck with cabin weight 6000, speed 115,
// make Scania, trailer weight 15000

Vehicle *vp;
// generic vehicle pointer

int main()
{
  // Vehicle::getweight() wird aufgerufen.
  vp = &v;
  cout << vp->getweight() << endl;

  // Truck::getweight() wird aufgerufen.
  vp = &t;
  cout << vp->getweight() << endl;

  // Fehler: getspeed() ist keine
  // Elementfunktion der Klasse Vehicle!
  cout << vp->getspeed() << endl;

  return 0;
}
```

Auswirkungen des Polymorphismus auf die Programm-Entwicklung

Transportmittel-Klassifizierungssystem

Implementierung in C

```
typedef enum    /* type of the vehicle */
{
  is_vehicle,
  is_land,
  is_auto,
  is_truck
} Vtype;

typedef struct  /* generic vehicle type */
{
  int weight;
} Vehicle;

typedef struct  /* land vehicle: adds speed */
{
  Vehicle v;
  int speed;
} Land;

typedef struct  /* auto: land vehicle + name */
{
  Land l;
  char *name;
} Auto;
```

Auswirkungen des Polymorphismus auf die Programm-Entwicklung

Transportmittel-Klassifizierungssystem
Implementierung in C

```
typedef struct   /* truck: auto + trailer */
{
  Auto a;
  int trailer_weight;
} Truck;

/* all sorts of vehicles in one union */
typedef union
{
  Vehicle v;
  Land l;
  Auto a;
  Truck t;
} AnyVehicle;

/* the data for all vehicles */
typedef struct
{
  Vtype type;
  AnyVehicle thing;
} Object;
```

Auswirkungen des Polymorphismus auf die Programm-Entwicklung

Transportmittel-Klassifizierungssystem

Implementierung in C

```
/* how to get the weight of a vehicle */
int getweight(Object *o)
{
  switch (o->type)
  {
    case is_vehicle:
      return o->thing.v.weight;
    case is_land:
      return o->thing.l.v.weight;
    case is_auto:
      return o->thing.a.l.v.weight;
    case is_truck:
      return o->thing.t.a.l.v.weight +
             o->thing.t.trailer_weight;
  }
}

void printweight(Object *any)
{
  printf("Weight: %d\n", getweight(any));
}
```

Auswirkungen des Polymorphismus auf die Programm-Entwicklung

Eine Erweiterung der C-Implementierung ist nicht einfach. Wenn ein Transportmittel `AirPlane` hinzugefügt werden soll, das z.B. zusätzlich die Anzahl der Passagiere speichert, muss der komplette C-Code geändert und neu übersetzt werden.

In C++ ermöglicht das dynamische Binden die unveränderte Wiederverwendung des "alten" Codes.

In C++ müssen wir zum Hinzufügen eines Transportmittels `AirPlane` lediglich eine neue Klasse `AirPlane` implementieren.

Bereits vorhandener Code wie z.B. die Funktion

```
void printweight(Vehicle const *any)
{
  printf("Weight: %d\n", any->getweight());
}
```

würde nach wie vor korrekt arbeiten.

Wegen des dynamischen Bindens braucht die Funktion `printweight()` nicht einmal neu übersetzt zu werden.

Virtuelle Funktionen

Implementierung des Polymorphismus

Ein Objekt, das virtuelle Funktionen enthält, enthält einen versteckten Zeiger auf eine Tabelle, die Zeiger auf die virtuellen Funktionen enthält.

Alle Objekte einer Klasse verweisen auf die gleiche Tabelle.

Es ist möglich, dass zwei Klassen sich die gleiche Tabelle "teilen".

Durch virtuelle Funktionen entsteht also zusätzlicher Speicherbedarf:

- ein Zeiger **vptr** pro Objekt; dieser verweist auf
- eine Tabelle **vtbl** pro Klasse, welche die Zeiger auf die virtuellen Funktionen enthält

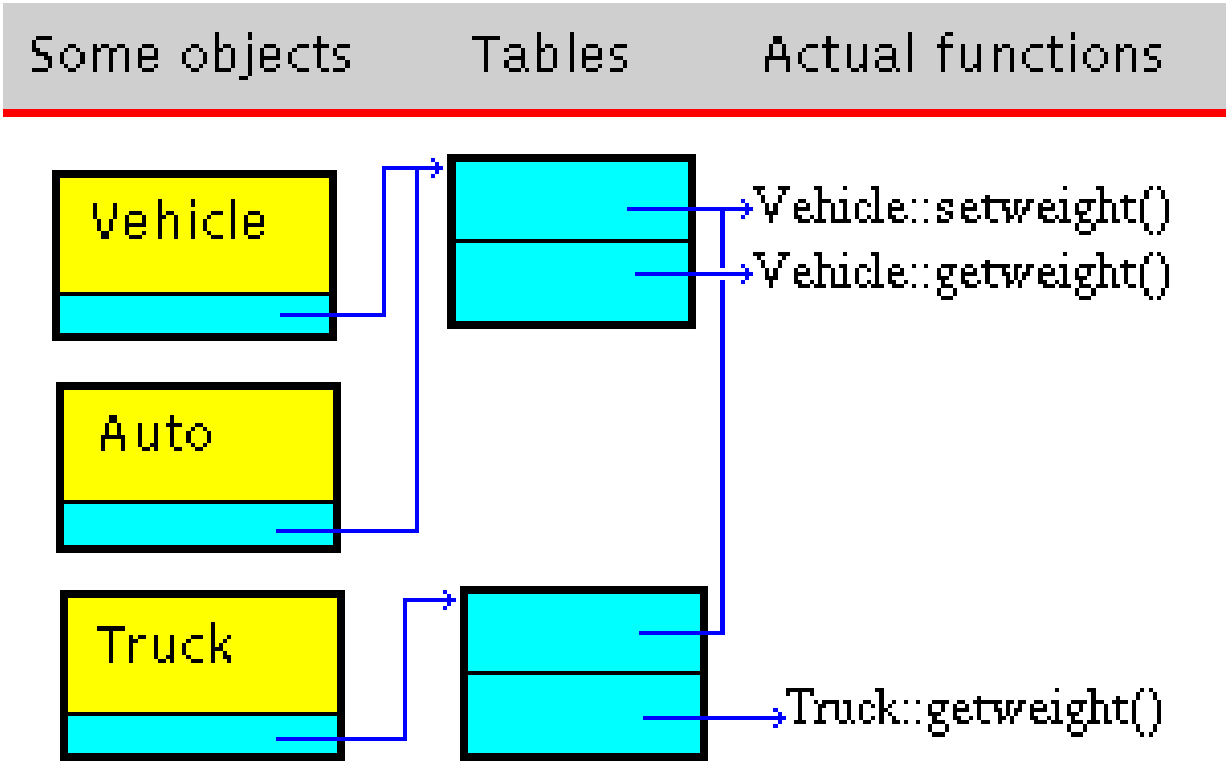
Implementierung des Polymorphismus

Beispiel:

Interne Organisation der Objekte des Transportmittel-Klassifizierungssystems

Objekte der Klassen `Vehicle` und `Auto` verweisen auf die gleiche Tabelle von Funktionszeigern.

Die Klasse `Truck` besitzt eine eigene Implementierung der Funktion `getweight()`; daher benötigt die Klasse eine eigene Tabelle von Funktionszeigern.



Virtuelle Funktionen

wesentliche Merkmale virtueller Funktionen:

- Virtuelle Funktionen dienen zum Überschreiben bei gleicher Signatur und gleichem Rückgabetyp.

- Der Aufruf einer nicht-virtuellen Funktion hängt vom Typ des Zeigers ab, über den die Funktion aufgerufen wird, während der Aufruf einer virtuellen Elementfunktion vom Typ des Objekts abhängt, auf das der Zeiger verweist.
  Das Gleiche gilt für Referenzen anstelle von Zeigern.

- Eine in einer Basisklasse als `virtual` deklarierte Funktion definiert eine Schnittstelle für alle abgeleiteten Klassen, auch wenn diese zum Zeitpunkt der Festlegung der Basisklasse noch unbekannt sind.

- Der vorstehende Punkt gilt auch für Destruktoren. Wenn es virtuelle Funktionen in einer Klasse gibt, sollte auch der Destruktor als `virtual` deklariert werden.

Wenn das Überschreiben einer Elementfunktion nötig erscheint, sollte die Funktion in der Basisklasse als `virtual` deklariert werden.

Polymorphismus

Rein virtuelle Funktionen

In C++ ist es möglich, virtuelle Funktionen in einer Basisklasse lediglich zu *deklarieren*, statt sie zu definieren.

Damit wird ein gemeinsames *Protokoll* (Name, Parameter und Rückgabetyp) für alle abgeleiteten Klassen festgelegt.

Die abgeleiteten Klassen müssen die Implementierung der Funktion vornehmen.

Funktionen, die in der Basisklasse deklariert, aber nicht definiert werden, heißen **rein virtuelle** Funktionen (engl. *pure virtual*).

Sie werden durch Anfügen von "`= 0`" deklariert.

Wenn eine Basisklasse eine rein virtuelle Funktion enthält, dann kann der C++-Compiler *kein* Objekt dieser Klasse anlegen.

Eine Klasse, die rein virtuelle Funktionen enthält, wird auch **abstrakte Klasse** genannt.

(Im Gegensatz zu **konkreten Klassen**, von denen Objekte erzeugt werden können.)

Rein virtuelle Funktionen

Beispiel:

Die Definition der Klasse `Sortable` legt fest, dass alle abgeleiteten Klassen eine Elementfunktion `compare()` enthalten müssen, deren Rückgabetyp `int` ist, und die als Argument eine Referenz auf ein `Sortable` -Objekt erwartet.

```
class Sortable
{
  public:
    virtual int compare
      (Sortable const &other) const = 0;
};
```

Rein virtuelle Funktionen

```
class Person : public Sortable
{
  public:
    // Konstruktoren, Destruktoren, etc.
    Person();
    Person(char const *nm, char const *add,
           char const *ph);
    Person(Person const &other);
    ~Person();
    Person const &operator=
      (Person const &other);

    // Zugriffsfunktionen
    char const *getname() const;
    char const *getaddress() const;
    char const *getphone() const;
    void setname(char const *nm);
    void setaddress(char const *add);
    void setphone(char const *ph);

    // Anforderungen, die von der Basisklasse
    // Sortable erzwungen werden
    int compare(Sortable const &other) const;

  private:
    // Datenelemente
    char *name, *address, *phone;
};
```

Rein virtuelle Funktionen

```
int Person::compare(Sortable const &o) const
{
  Person const &other = (Person const &)o;
  register int cmp;

  return

    // Erster Versuch: wenn die Namen
    // verschieden sind, sind wir fertig.
    (cmp = strcmp(name, other.name))

    ? cmp

    // Zweiter Versuch: Adressen vergleichen.
    : strcmp(address, other.address)

  ;
}
```

Rein virtuelle Funktionen

Evtl. soll die Funktion `Person::compare()` den Vergleich nur dann durchführen, wenn das Objekt `other` auch vom Typ `Person` ist.

Was würde die Anweisung

```
strcmp(name, other.name)
```

bewirken, wenn `other` kein `Person`-Objekt wäre und somit kein Datenelement `char *name` hätte?

Eine verbesserte Klasse `Sortable` erwartet von allen abgeleiteten Klassen, dass sie auch eine Funktion `getsignature()` definieren, mit der man den Typ eines Objekts einer abgeleiteten Klasse feststellen kann.

```
class Sortable
{
    ...
    virtual int getsignature() const = 0;
    ...
};
```

Rein virtuelle Funktionen

```
int Person::compare(Sortable const &o)
{
  register int cmp;

  // Zuerst wird die Signatur überprüft.
  if(cmp = getsignature() - o.getsignature())
    return cmp;

  Person const &other = (Person const &)o;

  return

    // Es sind zwei Person-Objekte,
    // daher müssen wir die Namen vergleichen.
    (cmp = strcmp(name, other.name))

    ? cmp

    // Letzter Versuch: Adressen vergleichen.
    : strcmp(address, other.address)

  ;
}
```

Rein virtuelle Funktionen

Wie kann die Funktion `getsignature()` implementiert werden?

```
class Person : public Sortable
{
    ...
    // getsignature() wird jetzt auch benötigt.
    int getsignature() const;
    ...
};

int Person::getsignature() const
{
  // tag ist eine funktions-statische Variable.
  // tag ist nur innerhalb der Funktion
  // Person::getsignature() sichtbar.
  // tag erhält einen festen Speicherplatz und
  // wird genau einmal - beim ersten Aufruf von
  // Person::getsignature() - initialisiert.
  static int tag;

  return (int) &tag;
}
```

15 Polymorphismus

Virtuelle Destruktoren

Im folgenden Beispiel wird ein Objekt der abgeleiteten Klasse (`Person`) über einen Basisklassenzeiger (`Sortable *`) zerstört. Daher wird statt des Destruktors der abgeleiteten Klasse der Destruktor der Basisklasse aufgerufen.

```
Sortable *sp;
Person *pp =
  new Person("Frank", "Ruwerblick 7", "12587");

sp = pp;      // sp zeigt jetzt auf eine Person
...
delete sp;    // Objekt wird zerstört
```

Abhilfe schafft ein **virtueller Destruktor**:

```
class Sortable
{
  public:
    virtual ~Sortable();
    virtual int
      compare(Sortable const &other) const = 0;
    ...
};
```

Virtuelle Destruktoren

Sollte der Destruktor der Basisklasse eine rein virtuelle Funktion sein?

Im Allgemeinen nicht.

Die Basisklasse sollte einen Destruktor definieren, der angewandt wird, falls abgeleitete Klassen keinen eigenen Destruktor definieren.

Ein solcher Destruktor der Basisklasse könnte einen leeren Rumpf haben.

```
Sortable::~Sortable()
{
}
```

Polymorphismus

Virtuelle Funktionen und Mehrfachvererbung

Eine Schwierigkeit bei der Mehrfachvererbung kann dann auftreten, wenn mehrere "Pfade" von der abgeleiteten Klasse zur Basisklasse führen.

Im folgenden Beispiel wird die Klasse `Derived` zweimal von der Klasse `Base` abgeleitet.

```
class Base
{
  public:
    void setfield(int val) { field = val; }
    int getfield() const { return field; }
  private:
    int field;
};

class Derived : public Base, public Base
{
};
```

Die Funktionalität der Klasse `Base` ist nun zweimal in `Derived` enthalten. Dies führt zu Mehrdeutigkeiten:

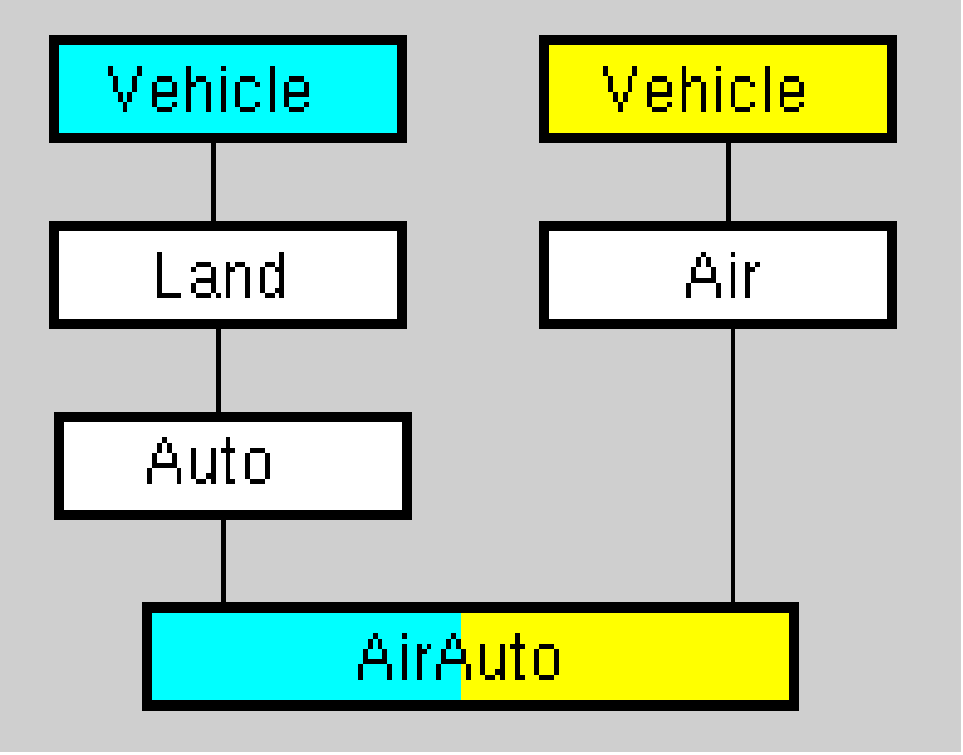
Wenn die Funktion `setfield()` für ein `Derived`-Objekt aufgerufen wird, welche Funktion ist dann gemeint?
Es gibt zwei `setfield()`-Funktionen!

Virtuelle Funktionen und Mehrfachvererbung

Mehrdeutigkeiten bei der Mehrfachvererbung

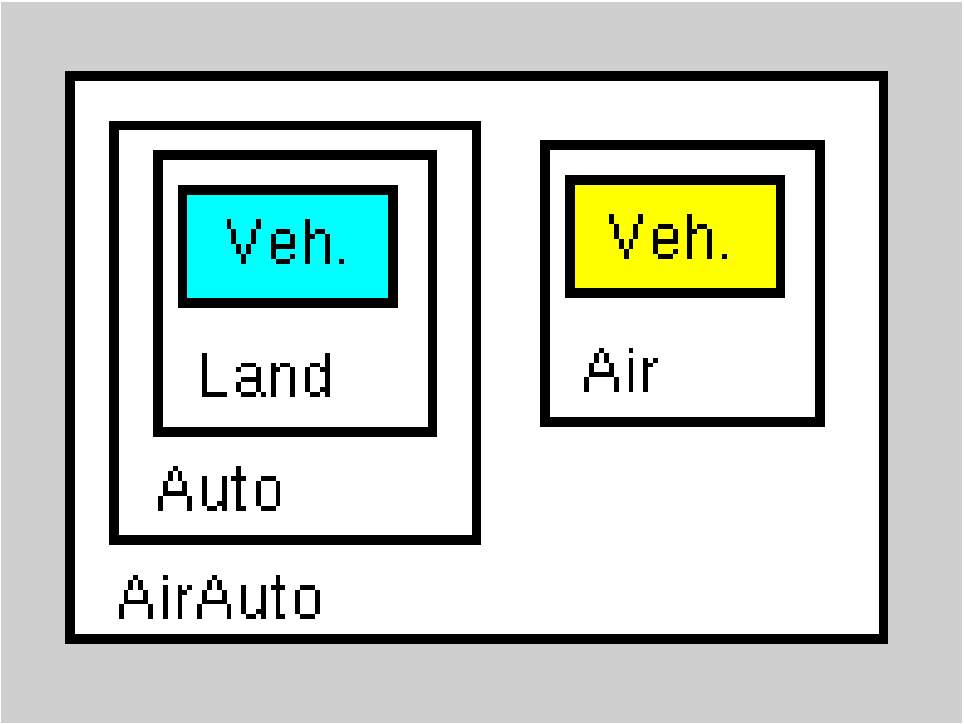
Ein `AirAuto`-Objekt z.B. würde zwei `Vehicles` enthalten und damit auch zwei `weight`-Datenelemente sowie zwei `setweight()`-Funktionen und zwei `getweight()`-Funktionen.

Die folgende Abbildung zeigt das Duplizieren eines Basisklassen-Objekts durch die Mehrfachvererbung.



Mehrdeutigkeiten bei der Mehrfachvererbung

Die folgende Abbildung zeigt den internen Aufbau eines Objekts vom Typ `AirAuto`.



Mehrdeutigkeiten bei der Mehrfachvererbung

Der C++-Compiler entdeckt die Mehrdeutigkeit in einem `AirAuto`-Objekt und weist den folgenden Code daher als fehlerhaft ab.

```
AirAuto cool;

cout << cool.getweight() << endl;
```

Der Compiler kann nicht entscheiden, welche der `getweight()`-Elementfunktionen aufgerufen werden soll.

Der Programmierer hat zwei Möglichkeiten, die Mehrdeutigkeit explizit aufzulösen.

- Der Funktionsaufruf, in dem die Mehrdeutigkeit auftritt, wird modifiziert:

```
// Wir hoffen, dass das Gewicht im Auto-
// Bestandteil des Objekts gespeichert ist.
cout << cool.Auto::getweight() << endl;
```

- Für die Klasse `AirAuto` wird eine spezielle Funktion `getweight()` bereitgestellt:

```
int AirAuto::getweight() const
{
  return Auto::getweight();
}
```

Virtuelle Funktionen und Mehrfachvererbung

Virtuelle Basisklassen

Wenn bei Mehrfachvererbung nicht erwünscht ist, dass mehrere Basisklassenobjekte erzeugt werden, können **virtuelle Basisklassen** verwendet werden.

Von diesen virtuellen Basisklassen erzeugt der C++-Compiler nur *ein* Subobjekt, auf das über verschiedene Vererbungswege zugegriffen werden kann.

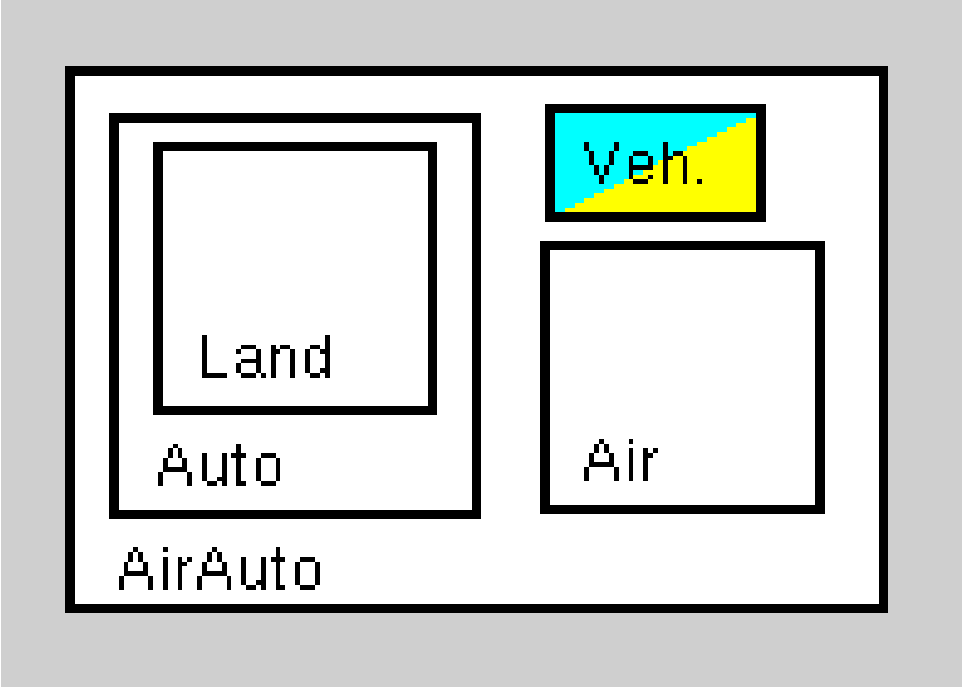
Für die Klasse `AirAuto` bedeutet das, dass die Ableitung der Klassen `Land` und `Air` geändert werden muss.

```
class Land : virtual public Vehicle
{
  ...
};

class Air : virtual public Vehicle
{
  ...
};
```

Virtuelle Basisklassen

Die folgende Abbildung zeigt den internen Aufbau eines Objekts vom Typ `AirAuto` bei virtueller Basisklasse `Vehicle`.



Virtuelle Basisklassen

Einige Anmerkungen zur virtuellen Ableitung:

- Virtuelle Ableitung geschieht zur Übersetzungszeit, nicht zur Laufzeit.

- Im obigen Beispiel würde es ausreichen, **entweder** `Land` **oder** `Air` als virtuell von `Vehicle` abgeleitet zu definieren.

- Der Konstruktor eines `AirAuto` ruft direkt den Konstruktor eines `Vehicle` auf - ohne den Umweg über den Konstruktor eines `Land` oder `Air`.