

Vererbung

Diese Teil befasst sich mit einem wichtigen Bestandteil Objektorientierter Sprachen: der Vererbung von Eigenschaften und damit verbundenen Konzepten.

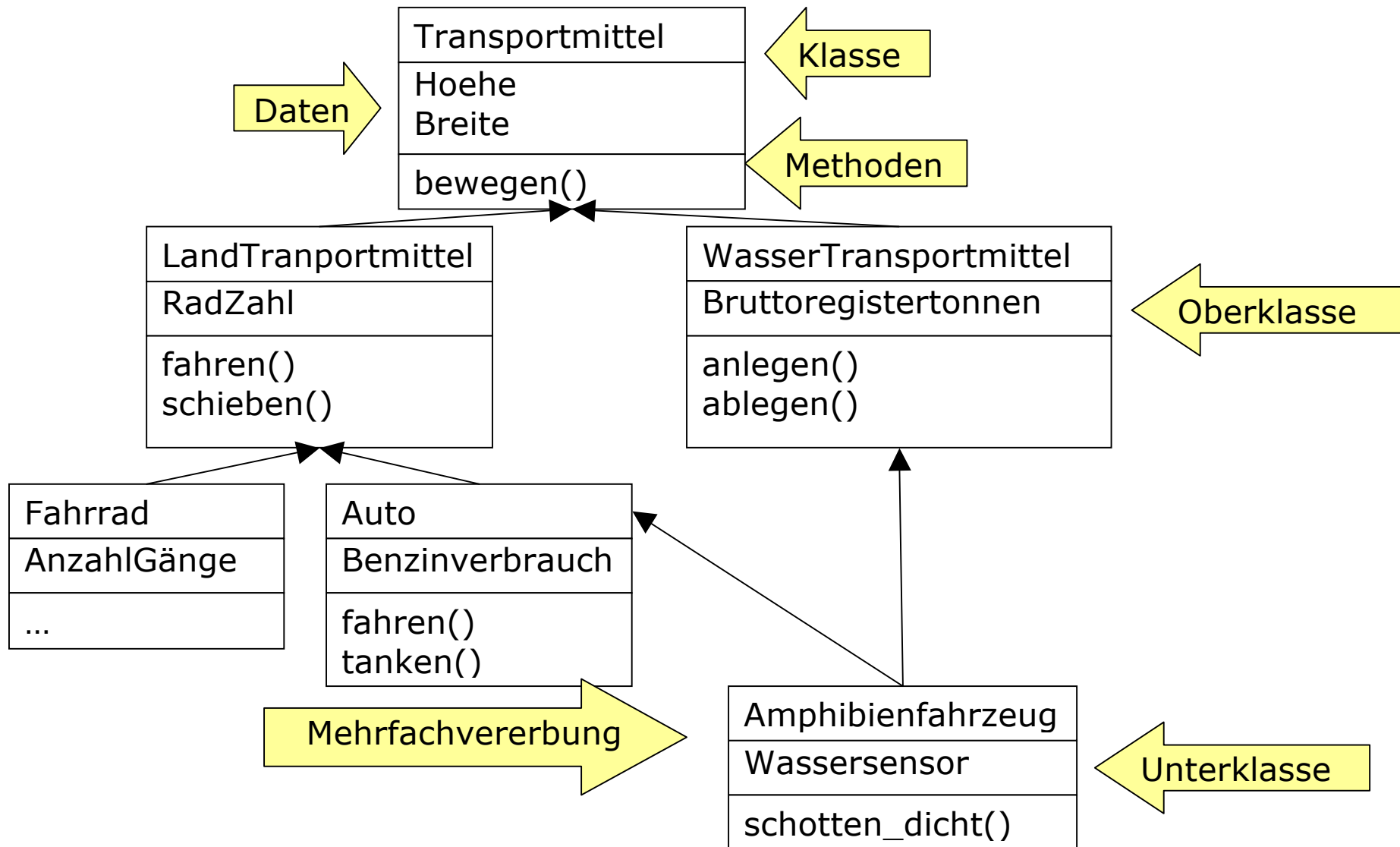
Inhalt

1	Überblick	3
1.1	Die Klasse Ort.....	8
1.2	Die Klasse GraphObj	12
1.3	Die Klasse Rechteck.....	15
1.4	Die Klasse Strecke.....	16
2	Vererbung und Initialisierung.....	19
3	Zugriffsschutz	20
3.1	Read-Only Daten.....	23
4	Typumwandlung Basisklasse – abgeleitete Klasse.....	26
5	Überschreiben von Funktionen	27
6	Polymorphismus	29

6.1	Virtuelle Funktionen.....	30
6.1.1	Verhalten nicht virtueller Funktionen.....	30
6.1.2	Verhalten virtueller Funktionen	32
6.2	Abstrakte Klassen.....	37
6.3	Virtuelle Destruktoren	45
7	Mehrfachvererbung	54
7.1	Namenskonflikte	58
7.2	Virtuelle Basisklassen.....	61
7.3	Virtuelle Basisklassen und Initialisierung.....	64

1 Überblick

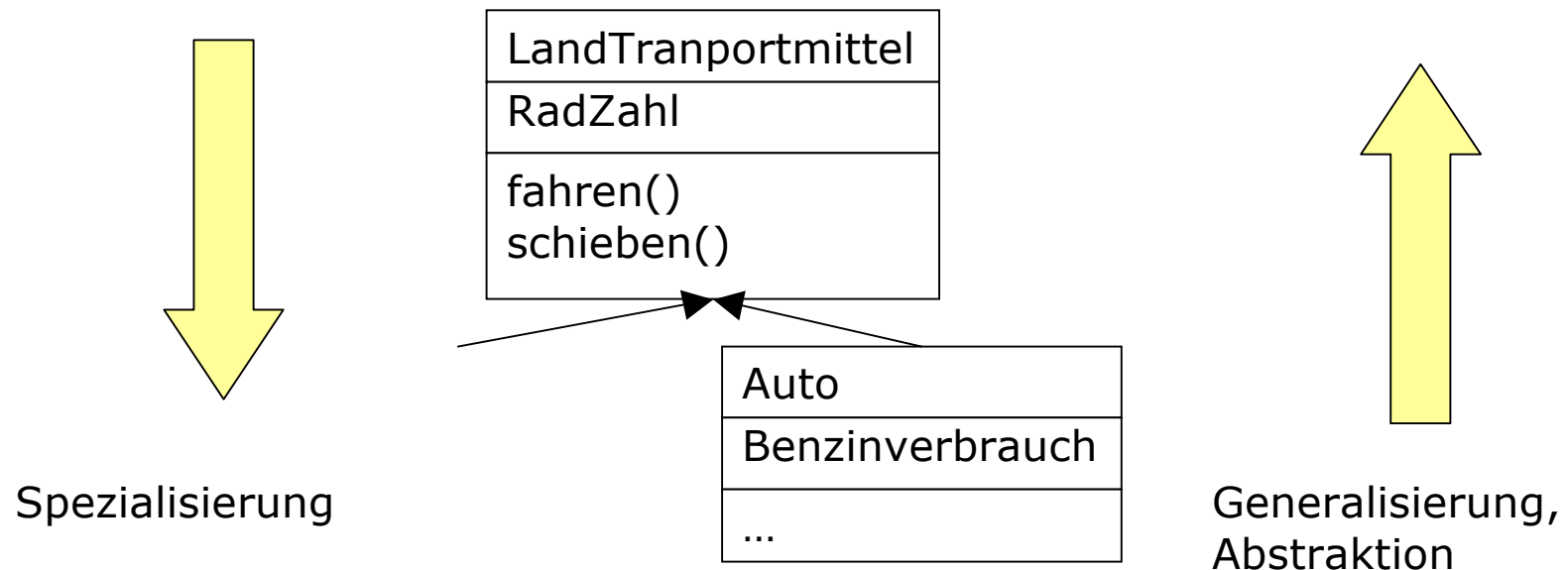
OOP ist ein Klassifizieren von Problemen. Ein Beispiel dafür ist die Einteilung von Transportmitteln:



Eine **Oberklasse** ist die **Abstraktion** oder **Generalisierung** von ähnlichen Eigenschaften der Unterklasse.

Die Unterklasse fügt zu den allgemeinen Eigenschaften der Oberklasse nur die unterklassenspezifischen Eigenheiten hinzu. Die allgemeinen Eigenschaften **erbt** die Unterklasse von der Oberklasse.

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



In C++ wird Abstraktion durch „: public“ ausgedrückt. Es wird als „**ist ein**“ oder „ist eine Art“ gelesen.

```
class Klassenname : public Oberklassenname
```

„Ein **Auto** ist ein **Landtransportmittel**“

Damit ist das Beispiel der Transportmittel in C++ wie folgt formulierbar:

```
class Transportmittel {
public:
    void bewegen();
private:
    double Hoehe, Breite;
};

class LandTransportmittel : public Transportmittel {
public: void fahren();
       void schieben();
private:
    int RadZahl;
};

class WasserTransportmittel : public Transportmittel {
public: void anlegen(); void ablegen();
private: double Bruttoregistertonnen;
};

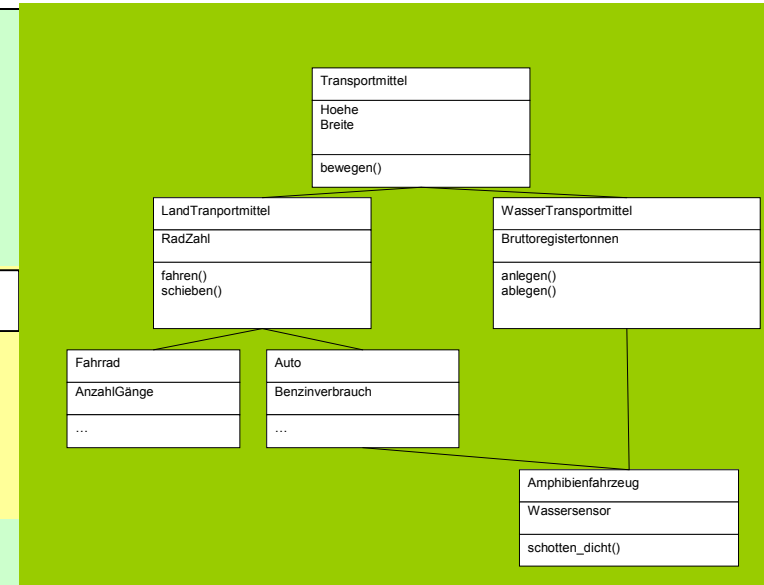
class Auto : public LandTransportmittel {
public:
    void fahren();
    void tanken();
private:
    int Benzinverbrauch;
};

class Amphibienfahrzeug : public Auto , public WasserTransportmittel {
public: void schotten_dichts();
private:
    String Wassensensor;
};
```

erben

überschreibt Methode Landtransportmittel::fahren()

Mehrfachvererbung

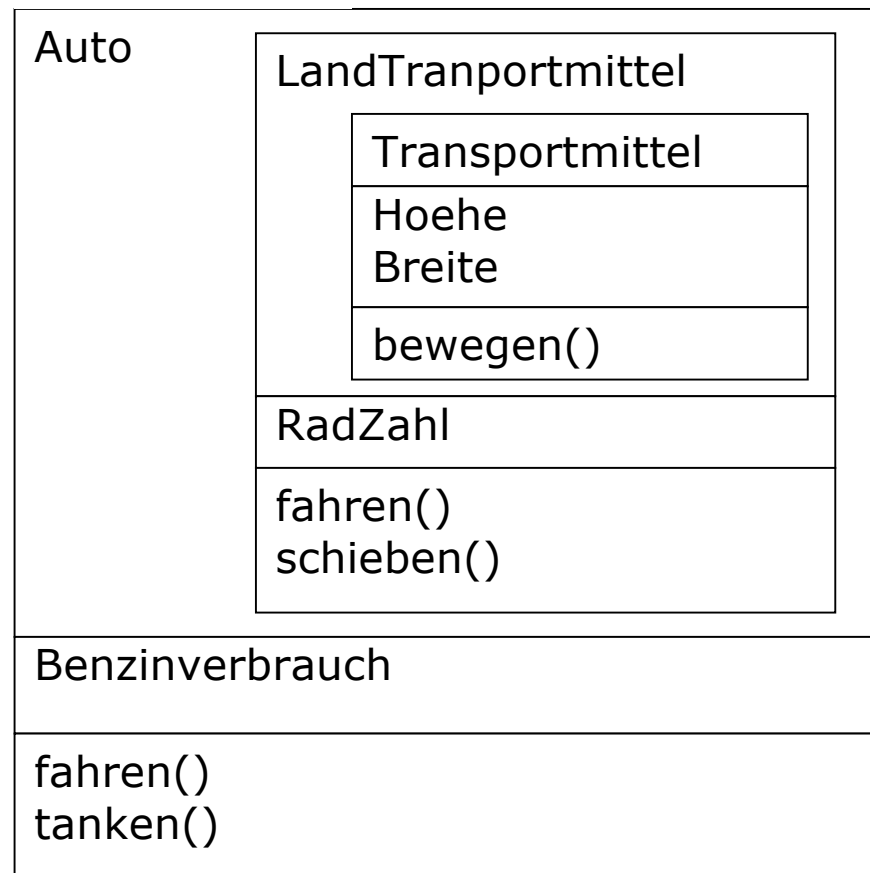


Vererbung (`class Abgeleitet : public Oberklasse`) bedeutet dabei in C++, dass:

- Jedes Objekt `ObjektAbgeleitet` vom Typ `Abgeleitet` beinhaltet ein Objekt vom Typ `Oberklasse` (Subobjekt genannt), das entsprechend Speicher belegt. Das Subobjekt wird noch vor der Erzeugung von `ObjektAbgeleitet` durch den Oberklassen-Konstruktor erzeugt.

Durch „`Auto meinAuto;`“ entsteht folgendes Konstrukt:

`meinAuto`



Somit existieren die Attribute:

- `meinAuto.Benzinverbrauch`
- `meinAuto.RadZahl`
- `meinAuto.Hoehe`
- `meinAuto.Breite`

- Jede (public-) **Elementfunktion** von Oberklasse kann auf ein Objekt vom Typ Abgeleitet angewendet werden. Am Aufruf einer Operation ist nicht erkennbar, ob es eine Methode der Oberklasse oder der Unterklasse ist. Somit ist im o.a. Beispiel z.B. `meinAuto.bewegen()` aufrufbar.
- Wenn eine Methode von Abgeleitet dieselbe Signatur wie eine Methode von Oberklasse hat, **überschreibt** die Abgeleitet-Methode die Oberklasse-Methode. Somit ist `meinAuto.fahren()` die speziell für Auto definierte Operation für das Fahren.
- In C++ ist eine Klasse ein Datentyp. Eine abgeleitete Klasse ist wie ein Subtyp der Oberklasse anzusehen (so, wie int ein Subtyp von double ist). Somit sind Zuweisungen der Form:

`ObjOberklasse = ObjAbgeleitet` **möglich**

```
LandTransportmittel meinLandtransportmitttel;
Auto meinAuto;
meinLandtransportmitttel = meinAuto; // 😊
```

Die nur zu ObjAbgeleiteten Attribute werden nicht kopiert, da in ObjOberklasse kein Platz dafür vorhanden ist.

`ObjAbgeleitet = ObjOberklasse` **nicht möglich**

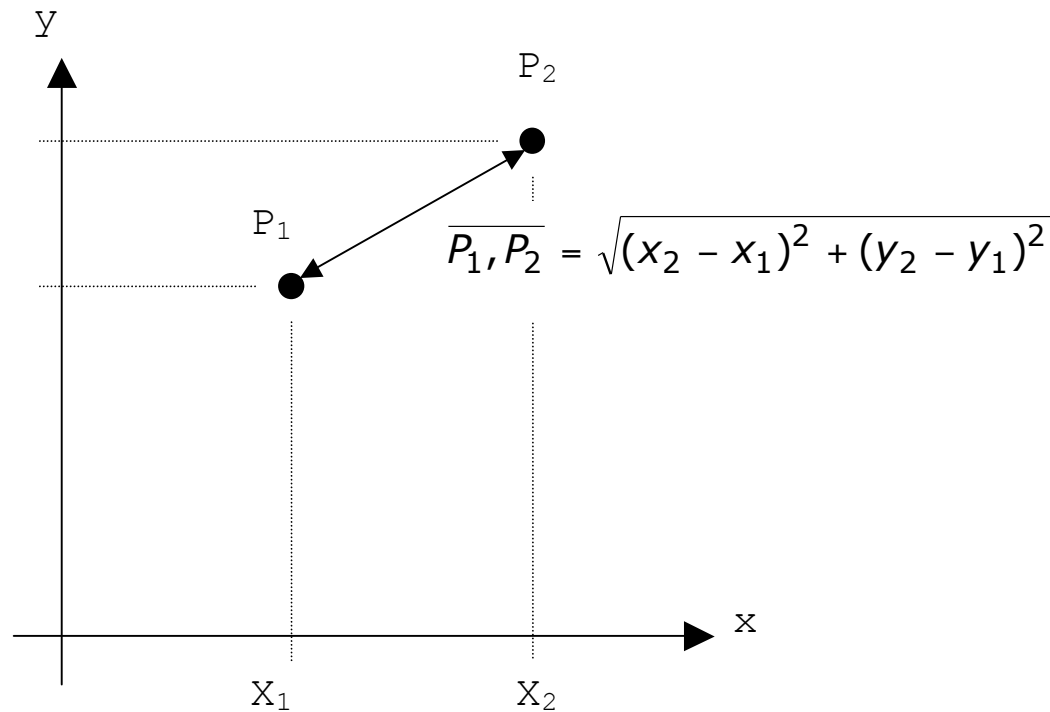
```
meinAuto = meinLandtransportmitttel; // 😞
```

Dies ist nicht möglich, da ansonsten in ObjektAbgeleitet nicht initialisierte Attribute vorhanden wären.

Der Vererbungsmechanismus wird am Beispiel einer Klasse `GraphObj` („graphisches Objekt“) und davon abgeleiteten Klassen für Rechtecke, Linien und Dreiecke demonstriert.

1.1 Die Klasse Ort

Zunächst wird eine Klasse `Ort` gebraucht, die einen Bezugspunkt `P` als Pixelkoordinaten (x,y) zur Verfügung stellt und einige Methoden, z.B. Entfernung implementiert.



```

$ cat ort.h
#ifndef ort_h
#define ort_h ort_h
#include<string>
#include<cmath>           // wegen sqrt()
#include<iostream>
using namespace std;

class Ort {
public:
    Ort(int einX = 0, int einY = 0) // Default: Nullpunkt
    : xKoordinate(einX), yKoordinate(einY) {

        int X() const { return xKoordinate;}
        int Y() const { return yKoordinate;}

        void aendern(int x, int y) {
            xKoordinate = x;
            yKoordinate = y;
        }
private:
    int xKoordinate,
        yKoordinate;
};

```

```

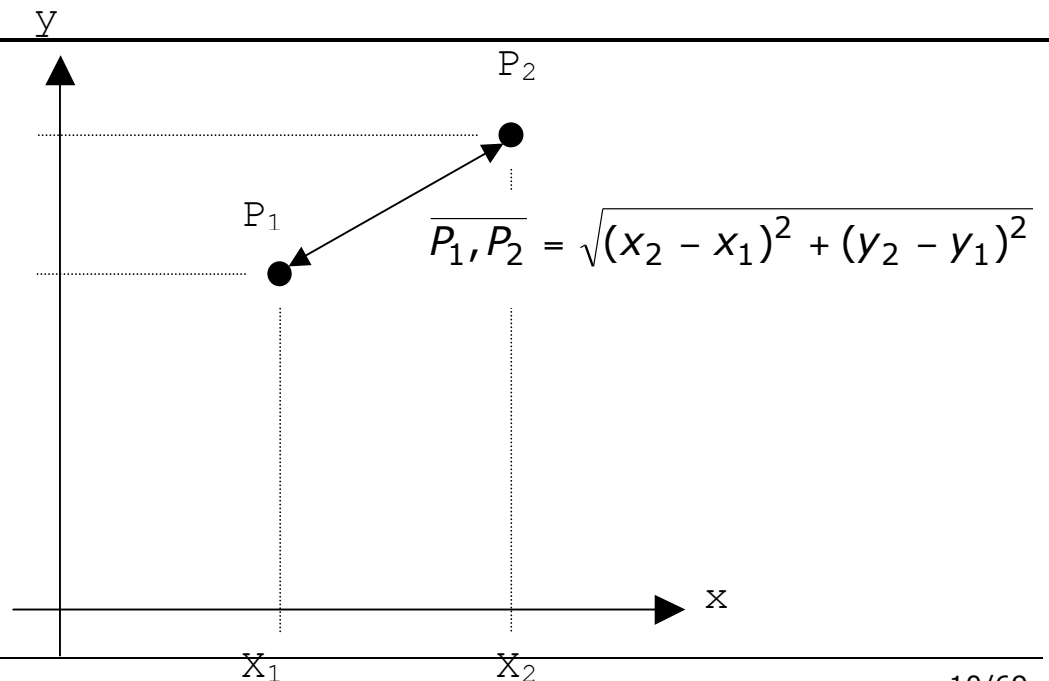
// globale Funktionen

// Berechnung der Entfernung zwischen zwei Orten
inline double Entfernung(const Ort &Ort1, const Ort &Ort2) {
    double dx = double(Ort1.X() - Ort2.X());
    double dy = double(Ort1.Y() - Ort2.Y());
    return sqrt(dx*dx + dy*dy);
}

// Anzeige auf der Standardausgabe
inline void anzeigen(const Ort &O) {
    cout << '(' << O.X() << ", " << O.Y() << ')';
}
#endif // ort_h
$

```

Damit kann ein Ort angezeigt und verändert werden:



```

$ cat ortmain.cpp
#include"ort.h"
using namespace std;

// Funktion zum Verschieben des Orts um dx und dy
Ort Ortsverschiebung(Ort derOrt, int dx, int dy) {
    derOrt.aendern(derOrt.X() + dx, derOrt.Y() + dy);
    return derOrt;           // Rückgabe des neuen Orts
}

int main() {
    Ort einOrt(10, 300);
    Ort verschobenerOrt = Ortsverschiebung(einOrt, 10, -90);
    cout << " alter Ort: ";
    anzeigen(einOrt);
    cout << "\n neuer Ort: ";
    anzeigen(verschobenerOrt);
    cout << endl;
}
$

```

Mit dieser einfachen Klasse, die Orte im Koordinatensystem abbildet, wird nun die Klasse `Graph-obj` definiert.

1.2 Die Klasse GraphObj

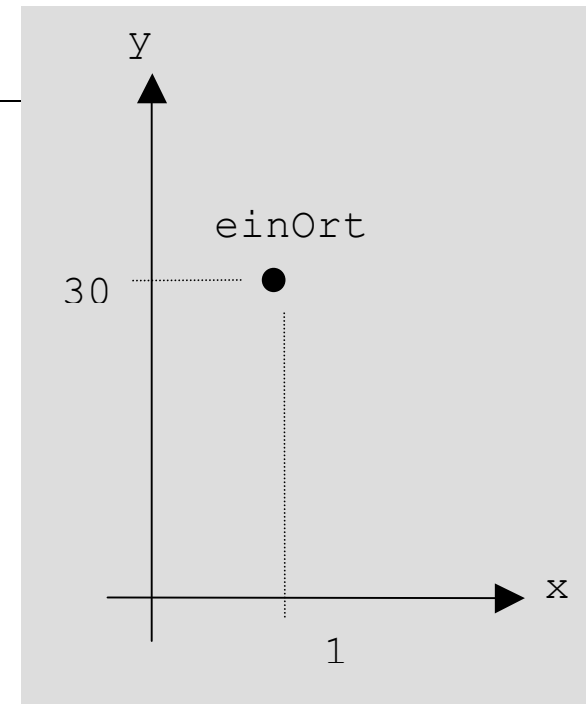
```
$ cat erben/graphobj.h
#include"ort.h"
class GraphObj { // Version 1
private:
    Ort Referenzkoordinaten;
public:
    GraphObj(const Ort &einOrt) // allg. Konstruktor
    : Referenzkoordinaten(einOrt) {}

    // Bezugspunkt ermitteln
    const Ort& Bezugspunkt() const { return Referenzkoordinaten; }

    // alten Bezugspunkt ermitteln und gleichzeitig neuen wählen
    Ort Bezugspunkt(const Ort &nO) {
        Ort temp = Referenzkoordinaten;
        Referenzkoordinaten = nO; // neuen Bezugspunkt setzen
        return temp; // alten Bezugspunkt zurückgeben
    }

    // Koordinatenabfrage
    int X() const { return Referenzkoordinaten.X(); }
    int Y() const { return Referenzkoordinaten.Y(); }

    // Standardimplementierung:
    double Flaechen() const {return 0.0;}
};
```



```

/* Die Entfernung zwischen 2 GraphObj-Objekten ist hier als
   Entfernung ihrer Bezugspunkte (überladene Funktion) definiert.
*/
inline double Entfernung(const GraphObj &g1, const GraphObj &g2) {
    return Entfernung(g1.Bezugspunkt(), g2.Bezugspunkt()); // Entfernung zwischen 2 Orten
}
$

```

Damit lassen sich Objekte erzeugen und ausgeben:

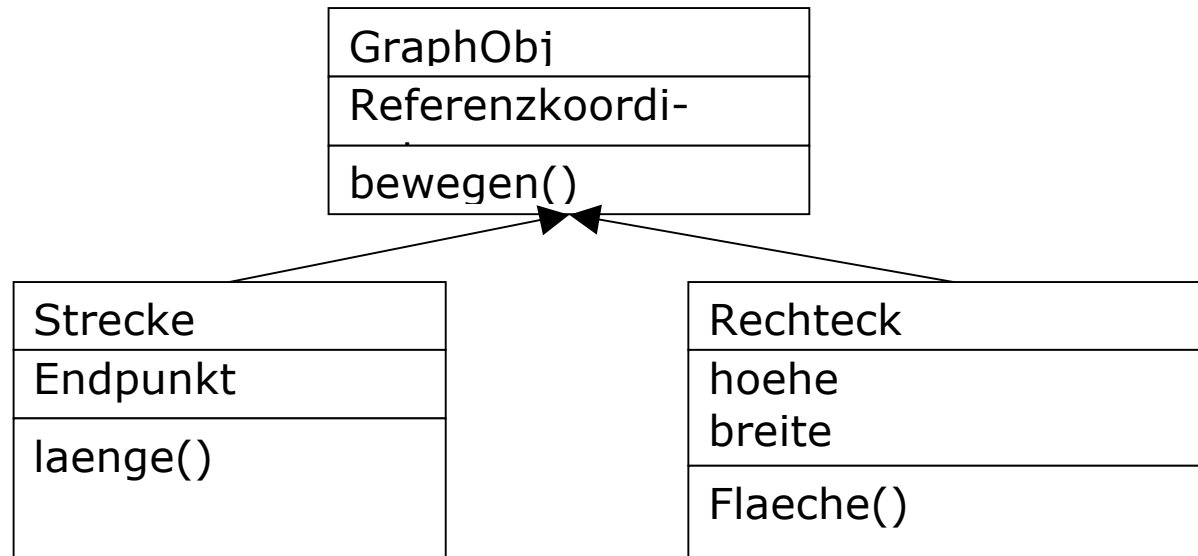
```

$ cat erben/main.cpp
#include "ort.h"
#include "graphobj.h"
int main() {
    Ort Nullpunkt; // Default-Konstruktoraufruf
    GraphObj G0(Nullpunkt);
    Ort einOrt(10, 20);
    GraphObj G1(einOrt);

    cout << "G0.X() = " << G0.X() << endl; // Ausgabe beider Bezugspunkte auf verschiedene Art
    cout << "G0.Y() = " << G0.Y() << endl;
    Ort R1 = G1.Bezugspunkt();
    cout << "R1.X() = " << R1.X() << endl;
    cout << "R1.Y() = " << R1.Y() << endl;
    cout << "Entfernung = " << Entfernung(G0, G1) << endl; // Ausgabe der Entfernung
}
$

```

Nun sollen abgeleitete Klasse erzeugt werden, die Rechtecke und Strecken ermöglichen; zunächst Rechtecke.



1.3 Die Klasse Rechteck

```
$ cat rechteck.h
#ifdef rechteck_h
#define rechteck_h rechteck_h
#include "graphobj.h"

class Rechteck : public GraphObj // von GraphObj erben
{
public:
    Rechteck(const Ort &p1, int h, int b)
        : GraphObj(p1), hoehe(h), breite(b) {} // Initialisierung des Subobjektes
                                                // und der eigenen Attribute

    double Flaeche() const
    { // int-Überlauf vermeiden
        return double(hoehe) * breite;
    }

private:
    int hoehe, breite;
};
#endif // rechteck_h
$
```

Ein Rechteck ist ein graphisches Objekt, die Klasse Rechteck erbt von GraphObj.

Rechteck mit Subobjekt GraphObj -> Tafel

1.4 Die Klasse Strecke

Auch eine Strecke ist ein graphisches Objekt.

```
$ cat strecke.h
#ifdef strecke_h
#define strecke_h strecke_h
#include "graphobj.h"

class Strecke : public GraphObj { // erben von GraphObj
public:
    // Initialisierung mit Initialisierungsliste
    Strecke(const Ort &Ort1, const Ort &Ort2)
        : GraphObj(Ort1), // Initialisierung des Subobjekts
          Endpunkt(Ort2) { // Initialisierung des Attributs
    } // leerer Code-Block

    double Laenge() const { // Methode Laenge verwendet Methode Entfernung
        return Entfernung(Bezugspunkt(), Endpunkt);
    }

private:
    Ort Endpunkt; // zusätzlich: 2. Punkt der Strecke
                 // (der erste ist GraphObj::Referenzkoordinaten)
};
#endif // strecke_h
$
```

Eine Strecke besteht aus 2 Punkten (Orten). Ein Ort ist bereits gegeben, da Strecke ein Subobjekt (GraphObj) enthält, also ist nur noch der Endpunkt zusätzlich erforderlich.

Somit können wir die Klassen in einem Testprogramm verwenden:

```
$ cat erben/main02.cpp
#include"strecke.h"
#include"rechteck.h"
using namespace std;

int main() {
    // Definition zweier graphischer Objekte
    Ort Nullpunkt;           // Default-Konstruktoraufruf
    GraphObj G0(Nullpunkt);
    Ort einOrt(10, 20);
    GraphObj G1(einOrt);
    Ort R1 = G1.Bezugspunkt();

    Ort O;
    Strecke S1(O, R1);

    cout << "Strecke von ";
    anzeigen(O);
    cout << " bis ";
    anzeigen(R1);
}
```

```

cout << "\n Fläche der Strecke S1 = "
    << S1.Flaeche()           // geerbte Methode
    << endl;

cout << "Länge der Strecke S1 = "
    << S1.Laenge()          // zusätzliche Methode
    << endl;

einOrt = Ort(20, 30);        // Neuzuweisung
Ort O2(100, 50);

Strecke S2(einOrt, O2);

cout << "= Entfernung der Bezugspunkte: "
    << Entfernung(S1.Bezugspunkt(), S2.Bezugspunkt())
    << endl;

cout << "Entfernung der Strecken S1, S2 = "
    << Entfernung(S1, S2) << endl;

Rechteck R0(Ort(0,0), 20, 50);
cout << "R0.Flaeche = "
    << R0.Flaeche() << endl;           // 1000

cout << R0.GraphObj::Flaeche(); // null!
}
$

```

```

$main02
G0.X() = 0
G0.Y() = 0
R1.X() = 10
R1.Y() = 20
Entfernung = 22.3607
neuer Bezugspunkt für G0:
G0.Bezugspunkt() = (10, 20)
Entfernung = 0
Strecke von (0, 0) bis (10, 20)
Fläche der Strecke S1 = 0
Länge der Strecke S1 = 22.3607
= Entfernung der Bezugspunkte:
36.0555
Entfernung der Strecken S1, S2 =
36.0555
R0.Flaeche = 1000
$

```

2 Vererbung und Initialisierung

Jedes Objekt einer abgeleiteten Klasse enthält ein anonymes Subobjekt der Oberklasse.

Deshalb kann der Oberklassenkonstruktor in einer Initialisierungsliste direkt aufgerufen werden:

```
class Strecke : public GraphObj { // erben von GraphObj
public:
    // Initialisierung mit Initialisierungsliste
    Strecke(const Ort &Ort1, const Ort &Ort2)
        : GraphObj(Ort1), // Initialisierung des Subobjekts
          Endpunkt(Ort2) { // Initialisierung des Attributs
...
}
```

Die Initialisierungsliste kann folgende Elemente enthalten:

- Elemente der Klasse selbst, also keine geerbten Elemente und
- Konstruktoraufrufe der Oberklassen.

3 Zugriffsschutz

Durch die Schlüsselworte `public` und `private` haben wir bis jetzt den Zugriff auf Attribute und Methoden geregelt, wobei gilt:

- Mit `public` gekennzeichnete Elemente unterliegen **keiner Restriktion**, sie sind auch von **außen** zugreifbar;
- `private` Elemente sind nur **innerhalb** der Klasse und für `friend` Klassen verfügbar.

Diese Zugriffsspezifizierer gelten auch in der Vererbungskette. Um den Zugriff auf die Vererbungskette zu beschränken, existiert ein weiteres Schlüsselwort:

- Mit `protected` definierte Elemente sind in der eigenen und in allen `public` **abgeleiteten** Klassen zugreifbar; **nicht** jedoch in anderen Klassen oder **außerhalb** der eigenen Klasse.

Zu der o.a. Definition gelten zusätzlich folgende Regel:

- `private` Elemente sind in einer **abgeleiteten** Klasse **nicht** zugreifbar;
- in allen anderen Fällen gilt immer das **restriktivere Zugriffsrecht**, bezogen auf die Zugriffsrechte für ein Element und die Zugriffskennung der Vererbung einer Klasse.

Beispiel:

`protected` Elemente mit `private` vererbter Klasse \Rightarrow `private` in vererbter Klasse

`protected` Elemente mit `public` vererbter Klasse \Rightarrow `protected` in vererbter Klasse

Kodebeispiel:

```
class Oberklasse {
    private:                                // Voreinstellung
        int Oberklasse_priv;
        void private_Funktion_Oberklasse();
    protected:
        int Oberklasse_prot;
    public:
        int Oberklasse_publ;
        void Funktion_Oberklasse(),
};

class abgeleiteteKlasse : public Oberklasse {
    int abgeleiteteKlasse_priv;
    public:
        int abgeleiteteKlasse_publ;
        void Funktion_abgeleiteteKlasse() {
            Oberklasse_priv = 1;           // Fehler!
            Oberklasse_prot = 2;          // ok
            Oberklasse_publ = 3;          // ok
        }
};
```

```
void main() {
    int m;
    abgeleiteteKlasse Objekt;

    m = Objekt.Oberklasse_publ;           // ok
    m = Objekt.Oberklasse_prot;           // ok
    m = Objekt.Oberklasse_priv;           // Fehler!
    m = Objekt.abgeleiteteKlasse_publ;    // ok
    m = Objekt.abgeleiteteKlasse_priv;    // Fehler!
    Objekt.Funktion_abgeleiteteKlasse();  // ok
    Objekt.Funktion_OberklasseKlasse();   // ok
    Objekt.private_Funktion_OberklasseKlasse(); // Fehler!
}
```

3.1 Read-Only Daten

Sollen Daten einer Klasse zum Lesen auch außerhalb der Klasse verfügbar gemacht werden, ohne dass die Daten aber außerhalb veränderbar sind, kann dies über den Zugriff einer Methode umgesetzt werden.

```
$ cat readonly.cpp
#include<iostream>
using namespace std;

class Zahl {
public:
    Zahl(int i)
    : privateZahl(i) {
    }
    int lesen() {
        return privateZahl;
    }

private:
    int privateZahl;
};

int main() {
    Zahl X(18);
    // X.privateZahl = 18; // Fehler! Zugriff nicht möglich!

    cout << "X.privateZahl= " << X.lesen() << endl; // erlaubter lesender Zugriff über Methodenaufruf
}
$
```

Der direkte Zugriff auf `private` Elemente kann realisiert werden durch eine `public const`-Referenz auf das `private` Element:

```
$ cat readonly02.cpp
#include<iostream>
using namespace std;
```

```

class Zahl {
public:
    Zahl()
        : readonlyZahl(privateZahl),
          privateZahl(0) {
    }
    void aendern(int wert) {
        privateZahl = wert;
    }
    const int& readonlyZahl;
private:
    int privateZahl; ←
};

int main() {
    Zahl X;
    cout << "X.readonlyZahl="
         << X.readonlyZahl << endl;

    // X.privateZahl = 18;
    // X.readonlyZahl = 18;

    X.aendern(18);

    cout << "X.readonlyZahl="
         << X.readonlyZahl << endl;
}
$

```

// Initialisierung der Referenz
// Initialisierung der privaten Daten

// public-Referenz auf Konstante, Initialisierung im Konstruktor

// erlaubter direkter lesender Zugriff:
// 0

// Fehler! Zugriff nicht möglich!
// Fehler! Änderung nicht möglich!

// erlaubte Änderung

// erlaubter direkter lesender Zugriff:
// 18

4 Typumwandlung Basisklasse – abgeleitete Klasse

Da eine abgeleitete Klasse quasi ein Subtyp ihrer Oberklasse ist, ist ein Objekt der abgeleiteten Klasse zuweisungskompatibel zu einem Objekt der Oberklasse.

Beispiel:

```
Ort o1, o2, o3;  
GraphObj g(o1);  
Strecke s(o1, o2);  
g = s;
```

Wirkung: `g.Referenzkoordinaten = s.Referenzkoordinaten`

Der Endpunkt der Strecke wird **nicht** kopiert, da er in `g` nicht als Attribut vorhanden ist!

```
s = g;
```

Nicht möglich!

```
GraphObj &rg = g;  
Strecke &rs = s;  
rg = rs;
```

Zuweisungen können auch mit Referenzen und Zeigern vollzogen werden, mit der Wirkung wie oben:

```
GraphObj *pg = &g;  
Strecke *ps = &s;  
pg = ps;
```

Wirkung: `g.Referenzkoordinaten = s.Referenzkoordinaten`

5 Überschreiben von Funktionen

Jedes `GraphObj` hat eine Fläche, die durch die Methode `Flaeche()` ermittelt wird. Der Flächeninhalt einzelner abgeleiteter Klassen, wie `Rechteck` oder `Kreis` ist aber unterschiedlich.

Daher kann jede abgeleitete Klasse ihre eigene Methode `Flaeche()` definieren und so die Methode `Flaeche()` der Oberklasse überschreiben.

```
class Rechteck : public GraphObj // von GraphObj erben
{ public:
    Rechteck(const Ort &p1, int h, int b)
      : GraphObj(p1), hoehe(h), breite(b) {}

    double Flaeche() const
    { // int-Überlauf vermeiden
      return double(hoehe) * breite;
    }
private:
    int hoehe, breite;
};
```



Überschriebene Methode

Soll eine Methode der Oberklasse verwendet werden, so ist der **Bereichsoperator** zu verwenden, z.B:

```
Rechteck r(Ort(0,0), 20, 50);
cout << r.Flaeche() << endl; // 1000
cout << r.GraphObj::Flaeche() << endl; // 0
```

Hörsaalübung

Erweitern Sie das Beispiel der graphischen Objekte um eine Klasse Kreis mit dem Attribut Radius und den Methoden Umfang und Fläche.

Zur Erinnerung:

Sei k ein Kreis mit Radius r .

$$Umfang(k) = 2r\pi$$

$$Fläche(k) = r^2\pi$$

6 Polymorphismus

Eine der wichtigsten Eigenschaften, die C++ zu einem wirklich objektorientierten System machen, ist die Fähigkeit zum Polymorphismus (Vielgestaltigkeit).

Polymorphismus heißt, dass dieselben Methode eines Objektes in verschiedener Klassen verschiedene Aktionen auslösen können.

Eine Form des Polymorphismus haben wir beim **Überladen von Funktionsnamen** und Operatoren kennen gelernt. Diese Methodik wird **frühe Bindung** (*early binding*) genannt, da schon beim Compilieren feststeht, welche der vorhandenen Implementationen von Funktionen und Operatoren benützt wird.

In C++ wird noch eine andere, wesentlich leistungsfähigere Art von Polymorphismus unterstützt. Sie wird mit "**später Bindung**" (*late binding*) bezeichnet und vom Konzept der virtuellen Funktionen getragen. Erst zur **Laufzeit** des Programms (abhängig von der Interaktion des Benutzers mit dem Programm) wird **entschieden**, welcher **Programmcode** eigentlich ausgeführt wird.

6.1 Virtuelle Funktionen

Soll erst zur Laufzeit entschieden werden, welches Objekt verwendet werden soll und demzufolge welche Methode aufgerufen werden soll, so sind virtuelle Funktionen der Basisklasse zu überladen.

Die Deklaration einer Funktion als `virtual` bewirkt, dass den Objekten die Information über den Objekttyp indirekt über einen Zeiger mitgegeben wird. Zu einem Objekt gehören zwei (verstecktes) Attribute

- `vptr`, ein Zeiger auf die Tabelle `vtbl` und
- `vtbl`, ein Array von Zeigern auf virtuelle Funktionen.

Wird eine virtuelle Funktion zur Laufzeit über einen Zeiger (oder eine Referenz) aufgerufen, wird über den Zeiger `vprt` in der Tabelle `vtbl` die auszuführende Funktion gesucht und aufgerufen.

Besitzt das Objekt keine Funktion, die zu der Signatur passt, wird eine Funktion mit passender Signatur in der Oberklasse gesucht.

6.1.1 Verhalten nicht virtueller Funktionen

Betrachten wir folgendes Programm, bei dem ein Zeiger auf ein `GraphObj` verwendet wird:

```

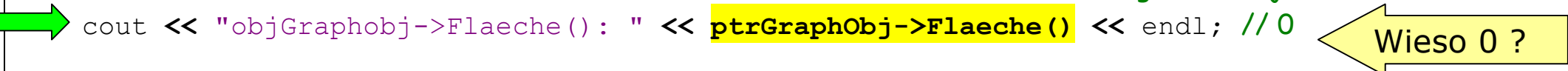
$ cat polymorphismus/main.cpp
#include"rechteck.h"
using namespace std;

int main() {
    GraphObj objGraphobj(Ort(20,20));
    Rechteck objRechteck(Ort(100,100), 20, 50);
    GraphObj *ptrGraphObj; // Zeiger auf ein GraphObj

    ptrGraphObj = &objGraphobj; // Zeiger auf objGraphobj setzen
    cout << "objGraphobj->Flaeche(): " << ptrGraphObj->Flaeche() << endl; // 0

    ptrGraphObj = &objRechteck; // Zeiger auf objRechteck setzen
    cout << "objGraphobj->Flaeche(): " << ptrGraphObj->Flaeche() << endl; // 0
    cout << "objRechteck.Flaeche(): " << objRechteck.Flaeche() << endl; //1000
}
$

```



Beim der **zweiten** Ausgabe wird 0 ausgegeben, das der Zeiger als Zeiger auf `GraphObj` definiert ist und somit die **Information**, dass er auf ein Rechteck zeigt **fehlt**. Somit wird die Methode `Flaeche` der Klasse `GraphObj` aufgerufen und nicht die der Klasse `Rechteck`. Dabei wird das anonyme Subobjekt vom Typ `GraphObj`, das bei der Erzeugung des Rechtecks angelegt wurde, angesprochen und dessen Methode `Flaeche` verwendet.

6.1.2 Verhalten virtueller Funktionen

Bei der Verwendung virtueller Funktionen wird das Laufzeitsystem dem Objekt, auf das der Zeiger zeigt, die erforderliche Typinformation mitgeben. Damit wird dann die „richtige“ Methode ausgewählt.

```
class GraphObj { // Version 1 + virtuelle Fkt
public:
...
// Standardimplementation:
double Flaeche() const {return 0.0;}
virtual double v_Flaeche() const {return 0.0;}
private:
Ort Referenzkoordinaten;
};
```

```
class Rechteck : public GraphObj // von GraphObj erben
{ public:
...
virtual double v_Flaeche() const { // int-Überlauf vermeiden
return double(hoehe) * breite;
}
private:
int hoehe, breite;
};
```

```

#include"rechteck.h"
using namespace std;

int main() {
    GraphObj objGraphobj(Ort(20,20));
    Rechteck objRechteck(Ort(100,100), 20, 50);
    GraphObj *ptrGraphObj; // Zeiger auf ein GraphObj

    ptrGraphObj = &objGraphobj; // Zeiger auf objGraphobj setzen
    cout << "objGraphobj->v_Flaeche(): " << ptrGraphObj->v_Flaeche() << endl; // 0

    ptrGraphObj = &objRechteck; // Zeiger auf objRechteck setzen
    cout << "objGraphobj->v_Flaeche(): " << ptrGraphObj->v_Flaeche() << endl; // 1000

    cout << "objRechteck.v_Flaeche(): " << objRechteck.v_Flaeche() << endl; // 1000
}

```

```

$ main02
objGraphobj->v_Flaeche(): 0
objGraphobj->v_Flaeche(): 1000
objRechteck.v_Flaeche(): 1000
$

```

Virtuelle Funktionen sind **automatisch** auch in nachfolgend abgeleiteten Klassen virtuelle. Man sollte der besseren Lesbarkeit wegen aber stets den Modifizierer `virtual` voranstellen!

Achtung:

- Alle eine virtuelle Funktion überladenen Funktionen **müssen** dieselbe Signatur besitzen. Ansonsten wird nicht erkannt, dass die Funktion in einer abgeleiteten Klasse virtuell ist und es wird der Vererbungskette nach oben folgend eine zur Signatur passende Funktion gesucht.
- Wenn Überschreiben notwendig ist, sollte die Funktion in der Basisklasse als virtuell gekennzeichnet werden, da nur dann das Verhalten eines Aufrufs unabhängig davon ist, ob es über einen Zeiger oder über einen Objektnamen erfolgt.

Welche Ausgabe erzeugt folgendes Programm (bsp01)?

<pre>class Basisklasse { public: virtual void vf1() { cout << "BK: vf1" << endl; }; virtual void vf2() { cout << "BK: vf2" << endl; }; virtual void vf3() { cout << "BK: vf3" << endl; }; void f() { cout << "BK: f" << endl; }; }; class AbgeleiteteKlasse : public Basisklasse { public: void vf1() { cout << "AK: vf1" << endl; }; void vf2(int i) { cout << "AK: vf2" << endl; }; // char void vf3() { cout << "AK: vf3" << endl; }; // Fehler: falscher rTyp void f() { cout << "AK: f" << endl; }; };</pre>	<pre>int main() { AbgeleiteteKlasse d; Basisklasse *bp = &d; bp->vf1(); bp->vf2(); d.vf2(6); bp->f(); }</pre>
--	--

Welche Ausgabe erzeugt folgendes Programm (bsp02)?

```
class Basisklasse {
public:
    void vf1() { cout << "BK: vf1" << endl; };
    void vf2() { cout << "BK: vf2" << endl; };
    void vf3() { cout << "BK: vf3" << endl; };
    void f() { cout << "BK: f" << endl; };
};

class AbgeleiteteKlasse : public Basisklasse {
public:
    void vf1() { cout << "AK: vf1" << endl; };
    void vf2(int i) { cout << "AK: vf2" << endl; };
    // char void vf3(){ cout << "AK: vf3" << endl; }; // Fehler: falscher rTyp
    void f() { cout << "AK: f" << endl; };
};

int main() {
    AbgeleiteteKlasse d;
    Basisklasse *bp = &d;
    bp->vf1();
    bp->vf2();
    d.vf2(6);
    bp->f();
}
```

Hörsaalübung

Erweitern Sie das Beispiel der graphischen Objekte mit der Klasse Kreis so dass virtuelle Funktionen zur Ermittlung der Fläche verwendet werden.

Zur Laufzeit soll ein Benutzer eingeben, ob ein Kreis oder ein Rechteck erzeugt werden soll, dann soll der Flächeninhalt ausgegeben werden indem auf das erzeugte Objekt über einen Zeiger zugegriffen wird.

6.2 Abstrakte Klassen

Bei der Realisierung großer Softwarepakete wird oft eine Menge von **allgemeinen Basisklassen** definiert, die über die **Lebensdauer** des Produktes (weitgehend) **unverändert** bleiben soll. Diese Klassen enthalten oft keine Implementierung. Sie dienen **ausschließlich** als Basis zur **Ableitung anderer Klassen**, die dann die allgemeinen Attribute und Methoden erben. Von diesen abgeleiteten Klassen können dann Objekte erzeugt werden.

Solche allgemeinen Klassen heißen „**abstrakte**“ Klassen. Die von abstrakten abgeleiteten Klassen, die selbst nicht abstrakt sind, nennt man **konkrete** Klassen.

In C++ werden abstrakte Klassen dadurch gebildet, dass sie **mindestens** eine „**rein virtuelle**“ Methode (*engl. pure virtual*) enthalten. Eine rein virtuelle Methode hat normalerweise **keinen Implementierungsteil** (kann aber einen haben).

Eine rein virtuelle Funktion wird gekennzeichnet durch Ergänzung um „=0“.

```
virtual int rein_virtuell_funktion (int) = 0;
```

Dadurch ist sicher gestellt, dass stets die zum Objekttyp passende Methode der Hierarchie aufgerufen wird.

Von einer abstrakten Klasse können **keine** Objekte instanziiert werden.

Wenn eine Klasse von einer abstrakten Klasse abgeleitet wird und in der abgeleiteten Klasse die **Implementierung fehlt**, so ist die **abgeleitete Klasse** selbst **abstrakt**.

Beispiel (Klasse Graphobj)

Wir wollen das Beispiel so **erweitern**, dass `Graphobj` eine **abstrakte** Klasse wird, indem die Methode `Flache()` eine **rein virtuelle** Methode wird. Außerdem erweitern wir die Klasse um eine rein virtuelle Methode `zeichnen()`, die für jede abgeleitete Klasse überschrieben werden muss, da z.B. Rechtecke anders gezeichnet werden müssen als Kreise.

```
$ cat abstrakt/main.cpp
#include"strecke.h"
#include"quadrat.h" // schliesst rechteck.h ein
using namespace std;
```

```

int main() {
    // GraphObj G; // Fehler: Instanzen abstrakter Klassen gibt es nicht.

    Rechteck R(Ort(0,0), 20, 50);
    Strecke S(Ort(1,20), Ort(200,0));
    Quadrat Q(Ort(122, 99), 88);

    // Feld mit Basisklassenzeigern, initialisiert mit den Adressen der Objekte und NULL als Endekennung
    GraphObj* GraphObjZeiger[] = {&R, &S, &Q, NULL};

    // Ausgabe der Fläche aller Objekte
    int i=0;
    while (GraphObjZeiger[i])
        cout << "Fläche = " << GraphObjZeiger[i++] ->Flaeche() << endl;

    // Zeichnen aller Objekte
    i=0;
    while (GraphObjZeiger[i])
        GraphObjZeiger[i++] ->zeichnen();

    std::cout << "Auch Referenzen sind polymorph:\n";
    GraphObj &R_Ref = R, // Der statische Typ ist derselbe,
             &S_Ref = S,
             &Q_Ref = Q;
    R_Ref.zeichnen(); // der dynamische nicht,
    S_Ref.zeichnen(); // d.h. immer andere Methode
    Q_Ref.zeichnen(); // für zeichnen()
}
$

```

Durch Polymorphie wird immer die richtige Methode ausgewählt

```

$ main
Fläche = 1000
Fläche = 0
Fläche = 7744
Zeichnen: Rechteck (h x b = 20 x 50) an der
Stelle (0, 0)
Zeichnen: Strecke von (1, 20) bis (200, 0)
Zeichnen: Quadrat (Seitenlaenge = 88) an
der Stelle (122, 99)
Auch Referenzen sind polymorph:
Zeichnen: Rechteck (h x b = 20 x 50) an der
Stelle (0, 0)
Zeichnen: Strecke von (1, 20) bis (200, 0)
Zeichnen: Quadrat (Seitenlaenge = 88) an
der Stelle (122, 99)
$

```

```

$ cat abstrakt/graphobj.cpp
#ifndef graphobj_h
#define graphobj_h graphobj_h
#include "ort.h"
#include<iostream>

class GraphObj {                                // Version 2
public:
    GraphObj(const Ort &einOrt)                 // allg. Konstruktor
    : Referenzkoordinaten(einOrt) {}

    virtual ~GraphObj() {}                     // virtueller Destruktor, Erklärung später

    // Bezugspunkt ermitteln
    const Ort& Bezugspunkt() const { return Referenzkoordinaten;}

    // alten Bezugspunkt ermitteln und gleichzeitig neuen wählen
    Ort Bezugspunkt(const Ort &nO) {
        Ort temp = Referenzkoordinaten;
        Referenzkoordinaten = nO;
        return temp;
    }

    // Koordinatenabfrage
    int X() const { return Referenzkoordinaten.X(); }
    int Y() const { return Referenzkoordinaten.Y(); }
}

```

```

// rein virtuelle Methoden
virtual double Flaeche() const = 0;
virtual void zeichnen() const = 0;

private:
    Ort Referenzkoordinaten;
};

// Standardimplementierung einer rein virtuellen Methode
inline void GraphObj::zeichnen() const {
    cout << "Zeichnen: ";
}

/* Die Entfernung zwischen 2 GraphObj-Objekten ist hier als
Entfernung ihrer Bezugspunkte (überladene Funktion)
definiert.*/

inline double Entfernung(const GraphObj &g1, const GraphObj &g2) {
    return Entfernung(g1.Bezugspunkt(), g2.Bezugspunkt());
}
#endif // graphobj_h
$

```

Die Klassen Strecke und Rechteck müssen die rein virtuellen Methoden implementieren, da sie sonst selbst abstrakt wären und somit keine Instanzen erzeugbar wären.

```

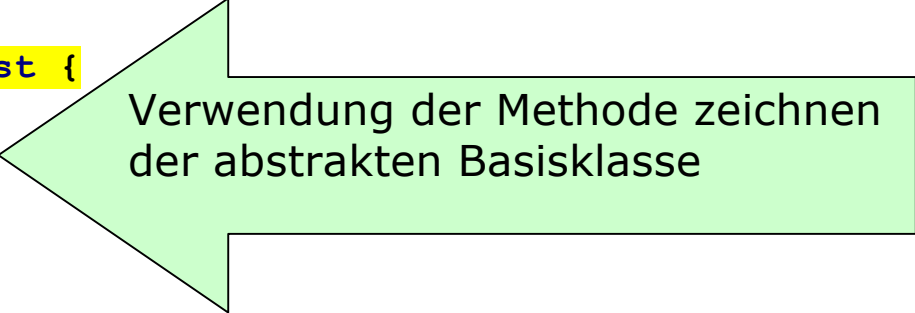
$ cat abstrakt/strecke.h
#ifndef strecke_h
#define strecke_h strecke_h
#include"graphobj.h"

class Strecke : public GraphObj    { // erben von GraphObj
public:
    // Initialisierung mit Initialisierungsliste
    Strecke(const Ort &Ort1, const Ort &Ort2)
    : GraphObj(Ort1),                // Initialisierung des Subobjekts
    Endpunkt(Ort2) {                 // Initialisierung des Attributs
    }                                 // leerer Code-Block
    double Laenge() const {
        return Entfernung(Bezugspunkt(), Endpunkt);
    }
    // Definition der virtuellen Methoden
    virtual double Flaeche() const {
        return 0.0;
    }

    virtual void zeichnen() const {
        GraphObj::zeichnen();
        cout << "Strecke von ";
        anzeigen(Bezugspunkt());
        cout << " bis ";
        anzeigen(Endpunkt);
        cout << std::endl;
    }

private:
    Ort Endpunkt;                    // zusätzlich: 2. Punkt der Strecke
    // (der erste ist GraphObj::Referenzkoordinaten)
};
#endif // strecke_h
$

```



Verwendung der Methode zeichnen
der abstrakten Basisklasse

```

$ cat abstrakt/rechteck.h
#ifndef rechteck_h
#define rechteck_h rechteck_h
#include"graphobj.h"

class Rechteck : public GraphObj // von GraphObj erben
{ public:
    Rechteck(const Ort &p1, int h, int b)
        : GraphObj(p1), hoehe(h), breite(b) {}

    // wird von Quadrat benötigt
    int Hoehe() const {return hoehe;}
    int Breite() const {return breite;}

    // Definition der rein virtuellen Methoden
    virtual double Flaeche() const {
        // int-Overflow vermeiden
        return double(hoehe) * breite;
    }

    virtual void zeichnen() const {
        GraphObj::zeichnen();
        cout << "Rechteck (h x b = "
                << hoehe << " x "
                << breite
                << ") an der Stelle ";
        anzeigen(Bezugspunkt());
        cout << endl;
    }

private:
    int hoehe, breite;
};
#endif // rechteck_h
$

```

```

$ cat abstrakt/quadrat.h
#ifndef quadrat_h
#define quadrat_h quadrat_h
#include"rechteck.h"

class Quadrat : public Rechteck    {
public:
    Quadrat(const Ort &O, int seite)
        : Rechteck(O, seite, seite) {}

    // Definition der rein virtuellen Methoden
    // Bezugspunkt(), Flaeche(), Hoehe() werden geerbt

    virtual void zeichnen() const {
        GraphObj::zeichnen();

        std::cout << "Quadrat (Seitenlaenge = " << Hoehe()
            << ") an der Stelle ";

        anzeigen(Bezugspunkt());
        std::cout << std::endl;
    }
};
#endif // quadrat.h
$

```

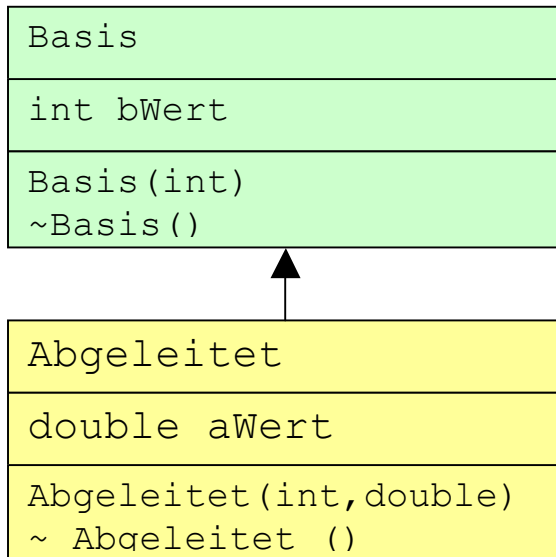
6.3 Virtuelle Destruktoren

Destruktoren sind für die Freigabe von belegtem Speicher zuständig.

Wird ein Objekt mit `p = new Klasse();` angelegt, so kann es mit `delete p` gelöscht werden.

Was passiert, wenn ein Zeiger vom Typ „Zeiger auf Basisklasse“, der auf ein Objekt einer abgeleiteten Klasse zeigt verwendet wird?

Beispiel:



```
$ cat virtuelleDestruktoren/dest.cpp
#include<iostream>
using namespace std;

#define PRINT(X) cout << (#X) << " = " << (X) << endl
```

```

class Basis {
    private:
        int bWert;
    public:
        Basis(int b=0)
        : bWert(b) {}

        ~Basis() {
            cout << "Objekt " << bWert << " Basis-Destruktor aufgerufen!\n";
        }
};

class Abgeleitet : public Basis {
    private:
        double aWert;
    public:
        Abgeleitet(int b = 0, double a = 0.0)
        : Basis(b), aWert(a) {}

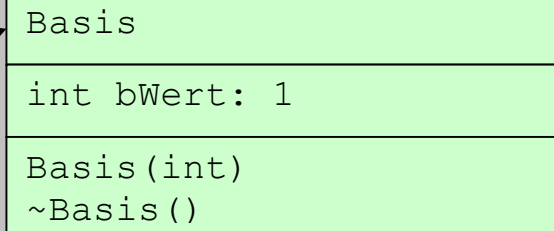
        ~Abgeleitet() {
            cout <<"Objekt " << aWert << " Abgeleitet-Destruktor aufgerufen!\n";
        }
};

```

```
int main () {  
    Basis *pb = new Basis(1);
```

```
    PRINT(sizeof(*pb));
```

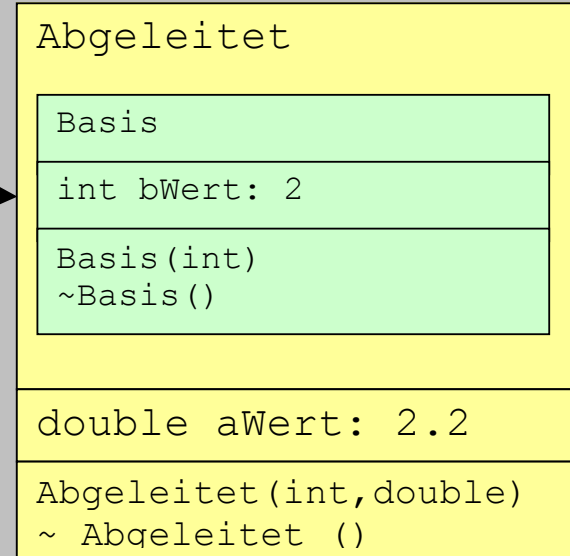
pb



sizeof liefert die statisch aus dem Typ des Zeigers ermittelbare Größe (int=4).

sizeof(*pb) = 4

```
Abgeleitet *pa = new Abgeleitet(2, 2.2);
```



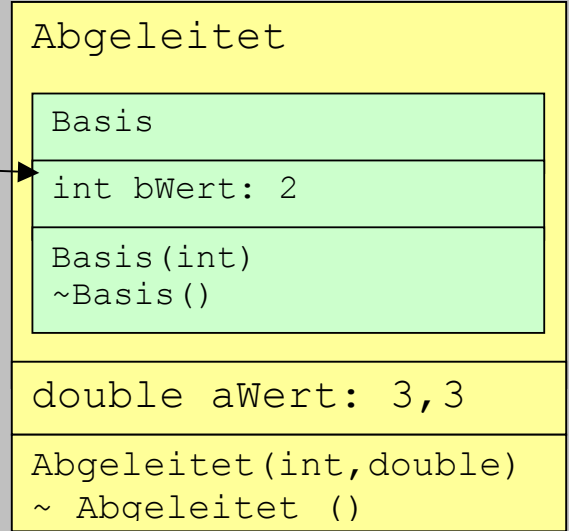
sizeof liefert die statisch aus dem Typ des Zeigers ermittelbare Größe. (8+4) aufgerundet auf 2 Worte

```
sizeof(*pa) = 16
```

```
PRINT(sizeof(*pa));
```

```
Basis *pba = new Abgeleitet(2, 3.3);
```

pba



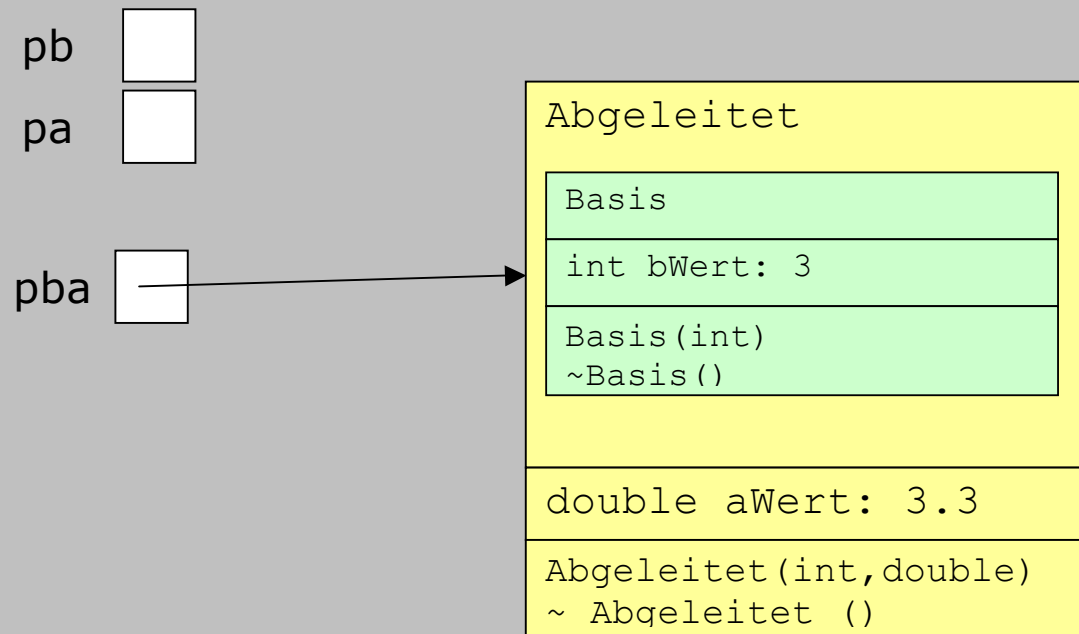
sizeof liefert die statisch aus dem Typ des Zeigers ermittelbare Größe. (4)

```
sizeof(*pba) = 4
```

```
PRINT(sizeof(*pba));
```

```
cout << "pb löschen:\n";
delete pb;

cout << "pa löschen:\n";
delete pa;
```



```
pb löschen:
Objekt 1 Basis-Destruktor aufgerufen!
pa löschen:
Objekt 2.2 Abgeleitet-Destruktor aufgerufen!
Objekt 2 Basis-Destruktor aufgerufen!
```

```
... Basis *pba ...
```

```
cout << "pba löschen:\n";  
delete pba;
```

pb

pa

pba

Da der Destruktor nicht virtual ist, bleibt der Speicherplatz nach delete stehen, da nur der Speicher gelöscht wird, der zum Typ des Zeigers passt.

Abgeleitet

double aWert: 3.3

Abgeleitet(int, double)
~ Abgeleitet ()

pba löschen:
Objekt 3 Basis-Destruktor aufgerufen!

```
}  
$
```

Ist der Destruktor **virtual**, liefert `sizeof` die **dynamisch** ermittelte Größe des Objektes. Weiterhin wird mit `delete` der **komplette** Speicher gelöscht:

```
$ cat virtuelleDestruktoren/virtdest.cpp
...
class Basis {
    private:
        int bWert;
    public:
        Basis(int b=0)
        : bWert(b) {}

        virtual ~Basis() {
            cout << "Objekt " << bWert << " Basis-Destruktor aufgerufen!\n";
        }
};
...
```

Regel:

Virtuelle Destruktoren sollten immer verwendet werden, wenn Basisklassenzeiger auf dynamisch erzeugte Objekte benutzt werden. Dies ist meistens der Fall bei virtuellen Methoden.

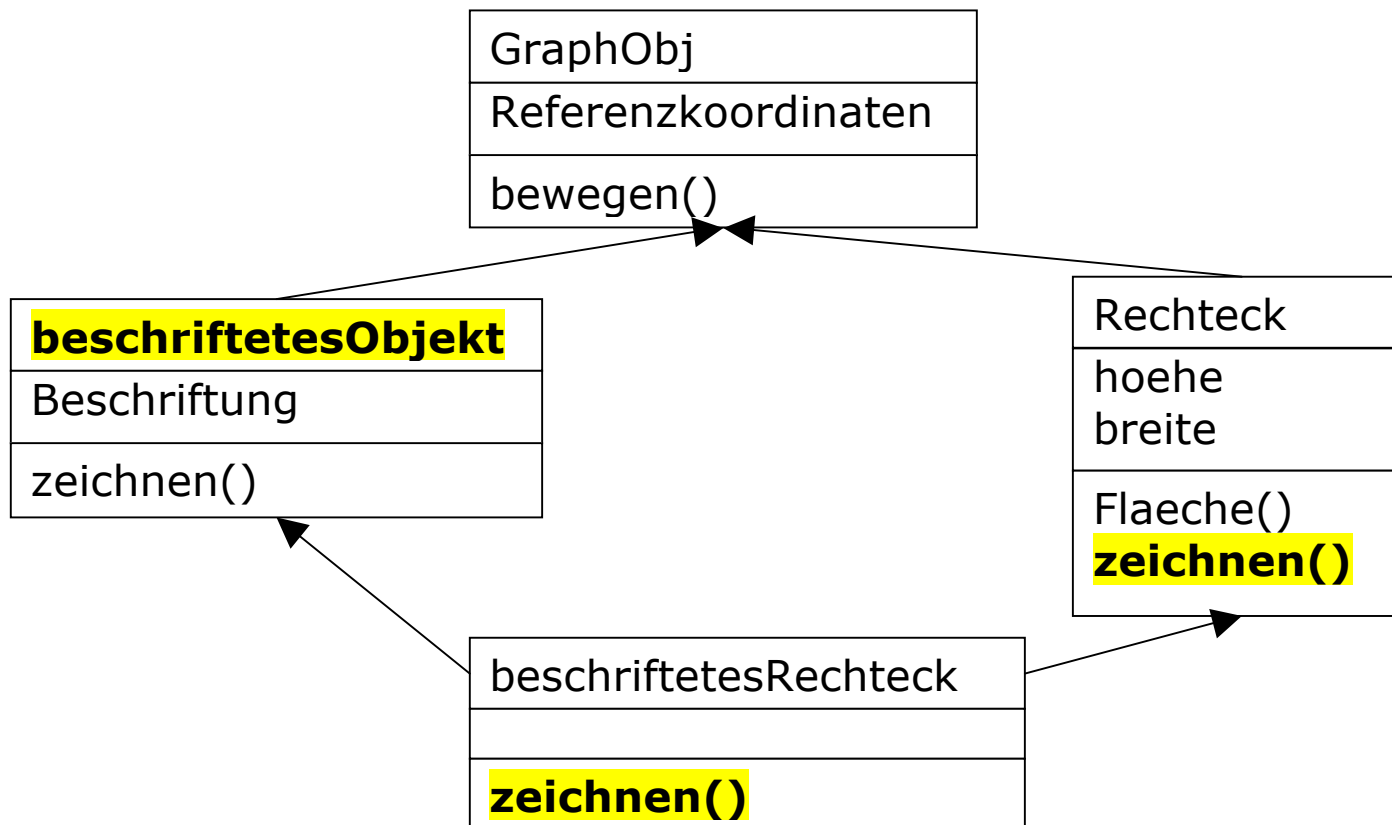
```
$ virtdest
sizeof(*pb) = 8
sizeof(*pa) = 16
sizeof(*pba) = 8
pb löschen:
Objekt 1 Basis-Destruktor aufgerufen!
pa löschen:
Objekt 2.2 Abgeleitet-Destruktor aufgerufen!
Objekt 2 Basis-Destruktor aufgerufen!
pba löschen:
Objekt 3.3 Abgeleitet-Destruktor aufgerufen!
Objekt 3 Basis-Destruktor aufgerufen!
$
```

7 Mehrfachvererbung

Große Softwaresysteme bilden komplexe Sachverhalte der realen Welt ab. Dabei sind häufig Beziehungen zu modellieren, die nicht streng hierarchisch, also baumartig sind. Vielmehr ist es erforderlich, dass eine Klasse von mehreren Basisklassen abgeleitet wird.

Beispiel:

Graphische Objekte können unbeschriftet oder beschriftet sein.



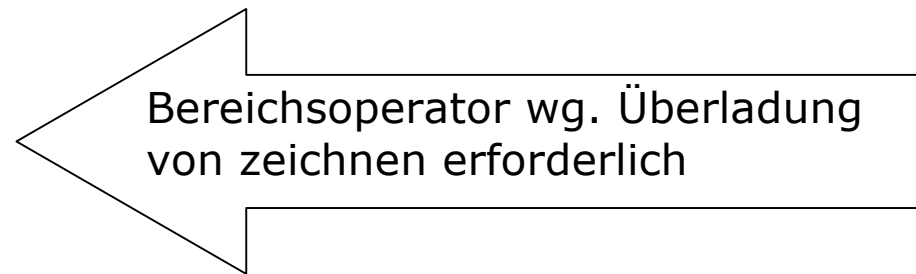
```

$ cat Version1/beschrif.h
#ifdef beschrif_h
#define beschrif_h beschrif_h
#include"graphobj.h"
#include<string>

class beschriftetesObjekt : public GraphObj { // einfache Vererbung
public:
    beschriftetesObjekt(const Ort &O, const string &m)
    : GraphObj(O), Beschriftung(m)
    {}

    virtual void zeichnen() const {
        GraphObj::zeichnen();
        cout << "Beschriftung bei ";
        anzeigen(Bezugspunkt());
        cout << Beschriftung << endl;
    }
private:
    string Beschriftung;
};
#endif // beschrif_h
$

```



```

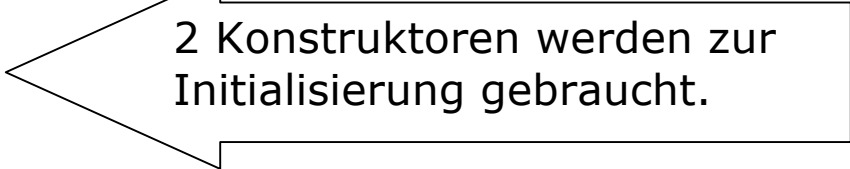
$ cat Version1/bes_r.h
#ifdef bes_r_h
#define bes_r_h bes_r_h
#include"beschrif.h"
#include"rechteck.h"

// Mehrfachvererbung
class beschriftetesRechteck
: public beschriftetesObjekt, public Rechteck { // Mehrfachvererbung
public:
    beschriftetesRechteck(const Ort &O, int h, int b, const string &m)
    : beschriftetesObjekt(O, m),
      Rechteck(O, h, b)
    {}

// Definition der rein virtuellen Methoden
virtual double Flaeche() const {
    // Definition ist notwendig, damit die Klasse nicht abstrakt ist
    // (durch Vererbung über beschriftetesObjekt und GraphObj)
    return Rechteck::Flaeche();
}

virtual void zeichnen() const {
    Rechteck::zeichnen();
    beschriftetesObjekt::zeichnen();
}
};
#endif // bes_r_h
$

```



2 Konstruktoren werden zur Initialisierung gebraucht.

Die anderen Klassen sind unverändert. Damit ist dann eine Verwendung wie folgt möglich:

```
$ cat Version1/main.cpp
#include"bes_r.h"
int main() {
    Rechteck R(Ort(0,0), 20, 50);

    beschriftetesRechteck BR(Ort(1,20), 60, 60, string("Mehrfachvererbung"));
    R.zeichnen();
    BR.zeichnen();

    beschriftetesRechteck *zBR =
        new beschriftetesRechteck(Ort(100,0), 20, 80, string("dynamisches Rechteck"));
    zBR->zeichnen();

    Rechteck R1(Ort(0,0), 20, 50);
    Rechteck R2(Ort(0,100), 10, 40);
    beschriftetesRechteck RB(Ort(1,20), 60, 60, string("Mehrfachvererbung"));

    // Feld mit Basisklassenzeigern, initialisiert mit den Adressen der Objekte, 0 als Endekennung
    GraphObj* GraphObjZeiger[] = {&R1, &R2, 0};

    // Zeichnen aller Objekte
    int i = 0;
    while (GraphObjZeiger[i])
        GraphObjZeiger[i++] ->zeichnen();
}
$
```

```
$ main
Zeichnen: Rechteck (h x b = 20 x 50) an der Stelle (0, 0)
Zeichnen: Rechteck (h x b = 60 x 60) an der Stelle (1, 20)
Zeichnen: Beschriftung bei (1, 20)Mehrfachvererbung
Zeichnen: Rechteck (h x b = 20 x 80) an der Stelle (100, 0)
Zeichnen: Beschriftung bei (100, 0)dynamisches Rechteck
Zeichnen: Rechteck (h x b = 20 x 50) an der Stelle (0, 0)
Zeichnen: Rechteck (h x b = 10 x 40) an der Stelle (0, 100)
$
```

7.1 Namenskonflikte

Durch Mehrfachvererbung können Konflikte entstehen, da es Mehrdeutigkeiten gibt:

```
$ cat Namenskonflikt01/main.cpp
#include"bes_r.h"
int main() {
    Rechteck R(Ort(0,0), 20, 50);

    beschriftetesRechteck BR(Ort(1,20), 60, 60, string("Mehrfachvererbung"));
    R.zeichnen();
    BR.zeichnen();

    beschriftetesRechteck *zBR =
        new beschriftetesRechteck(Ort(100,0), 20, 80, string("dynamisches Rechteck"));

    cout << "Rechteck-Position: ";
    anzeigen(R.Bezugspunkt());

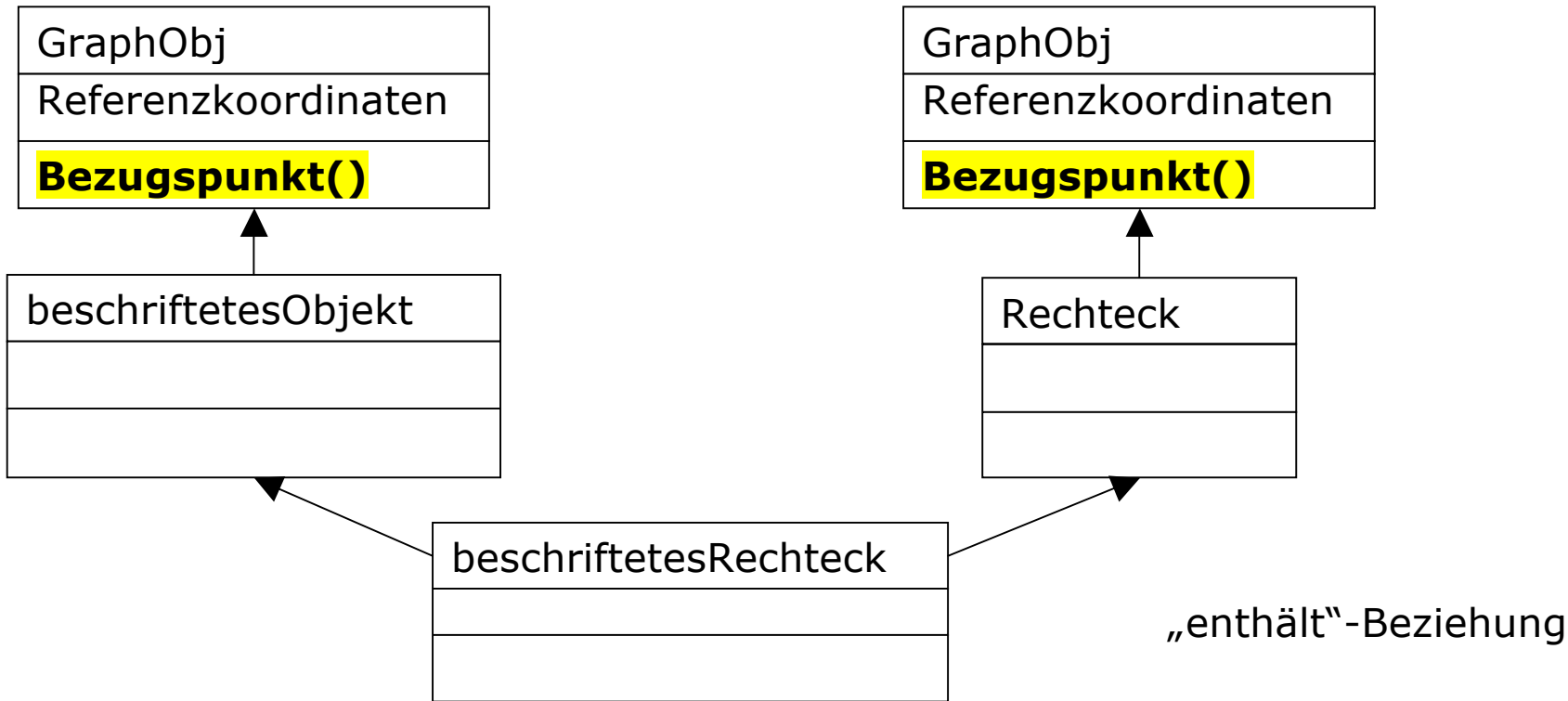
    cout << "\n beschriftetes-Rechteck-Position: ";
    // anzeigen(BR.Bezugspunkt());

    // eindeutig:
    anzeigen(BR.Rechteck::Bezugspunkt());
}
$
```

ok, da nur ein Bezugspunkt existiert, da R nur einfach erbt.

Compilerfehler: es ist nicht eindeutig, welcher Bezugspunkt gemeint ist, das Mehrfachvererbung, vgl. Abbildung

So wird es eindeutig!



Da zwei Subobjekte vom Typ `GraphObj` vorliegen kann wegen der Mehrdeutigkeit keine Zuweisung über einen Zeiger erfolgen, der Compiler meldet einen Fehler:

```

$ cat Namenskonflikt02/main.cpp
#include"bes_r.h"
int main() {
    Rechteck R1(Ort(0,0), 20, 50);
    Rechteck R2(Ort(0,100), 10, 40);
    beschriftetesRechteck BR(Ort(1,20), 60, 60, string("Mehrfachvererbung"));

    // Feld mit Basisklassenzeigern, initialisiert mit den Adressen der Objekte, 0 als Endekennung
    GraphObj* GraphObjZeiger[] = {&R1, &R2, 0}; // ok

    // Fehler: GraphObj* GraphObjZeiger[] = {R1, &R2, &BR, 0};

    // Zeichnen aller Objekte
    int i = 0;
    while (GraphObjZeiger[i])
        GraphObjZeiger[i++] -> zeichnen();
}
$

```

main.cpp:10:
cannot convert `Rechteck' to `GraphObj*'
in initialization

Diese Mehrdeutigkeiten können aufgelöst werden, wenn man virtuelle Basisklassen verwendet.

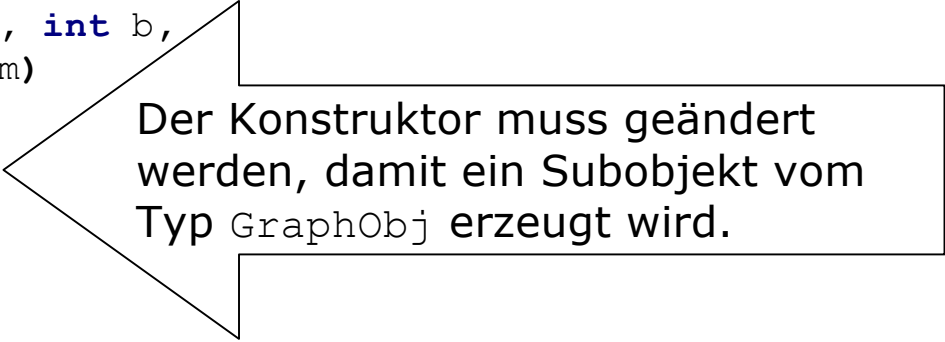
7.2 Virtuelle Basisklassen

Soll bei Mehrfachvererbung nur ein Basisklassenobjekt erzeugt werden, sind virtuelle Basisklassen zu verwenden. Von einer **virtuellen Basisklasse** wird **nur ein Subobjekt** erzeugt, auf das über verschiedene Vererbungspfade zugegriffen werden kann.

```
$ cat virtuelleBasisklasse/rechteck.h
class Rechteck : virtual public GraphObj // von GraphObj erben
{ public:
    ...
};
$
```

```
$ cat virtuelleBasisklasse/beschrift.h
class beschriftetesObjekt : virtual public GraphObj { // erben
    ...
};
$
```

```
$ cat virtuelleBasisklasse/bes_r.h
class beschriftetesRechteck
  : public beschriftetesObjekt, public Rechteck {
public:
  beschriftetesRechteck(const Ort &O, int h, int b,
                       const std::string &m)
  : GraphObj(O),
    beschriftetesObjekt(O, m),
    Rechteck(O, h, b)
  {}
...
};
$
```



Der Konstruktor muss geändert werden, damit ein Subobjekt vom Typ GraphObj erzeugt wird.

Damit kann nun über einen Zeiger zugegriffen werden:

```

$ cat main.cpp
#include"bes_r.h"
int main() {
    Rechteck R1(Ort(0,0), 20, 50);
    Rechteck R2(Ort(0,100), 10, 40);
    beschriftetesRechteck BR(Ort(1,20), 60, 60, string("Mehrfachvererbung"));

    // Feld mit Basisklassenzeigern, initialisiert mit den Adressen der Objekte, 0 als Endekennung
    GraphObj* GraphObjZeiger[] = {&R1, &R2, &BR, 0}; // mit virtueller Basisklasse ok

    // Zeichnen aller Objekte
    int i = 0;
    while (GraphObjZeiger[i])
        GraphObjZeiger[i++] -> zeichnen();
}
$

```

7.3 Virtuelle Basisklassen und Initialisierung

Die Initialisierungsliste bei Konstruktoren wird **vor** dem eigentlichen Codeblock ausgeführt. Gibt es (wg. Vererbung) mehrere Initialisierungslisten für eine Basisklasse, entstehen Probleme, wenn eine virtuelle Basisklasse verwendet wird und unterschiedliche Initialisierungen vorgenommen werden, da ja nur ein Subobjekt existiert.

In C++ wird das Problem gelöst durch die Regel:

1. Ist im Konstruktor kein Basisklasseninitialisierer angegeben, so wird der **Standardkonstruktor** der **virtuellen** Basisklasse verwendet;
2. ansonsten wird der Basisklasseninitialisierer verwendet, der im Konstruktor eines **vollständigen** Objektes angegeben ist.

Beispiel:

```

$ cat basinit.cpp
class Basis
{
public:
    Basis() { cout << "Basis-Standardkonstruktor\n"; }
    Basis(char* a) { cout << a << endl; }
};
class Links : virtual public Basis
{
public:
    Links(char* a) { // kein Basisklasseninitialisierer
    }
};
class Rechts : virtual public Basis
{
public:
    Rechts(char* a)
    : Basis(a) {
};
class Unten: public Links, public Rechts
{
public:
    Unten(char* a)
    : Links(a), Rechts(a) {
};

int main() {
    Links li("Links");
    Unten x("Unten"); // Regel 1
}
$

```

Basisklasseninitialisierer wird ignoriert (Regel 1)

```

$ basinit
Basis-Standardkonstruktor
Basis-Standardkonstruktor
$

```

```

$ cat basinit02.cpp
class Basis
{
public:
    Basis() { cout << "Basis-Standardkonstruktor\n"; }
    Basis(char* a) { cout << a << endl; }
};
class Links : virtual public Basis
{
public:
    Links(char* a)
        : Basis(a) { // Basisklasseninitialisierer vorhanden!
    }
};
class Rechts : virtual public Basis
{
public:
    Rechts(char* a)
        : Basis(a) {
    }
};
class Unten: public Links, public Rechts
{
public:
    Unten(char* a)
        : Basis(a), Links(a), Rechts(a) {
    }
};

int main()
{
    Links li("Links");
    Unten x("Unten");    // Regel 2
}
$

```

```

$ basinit02
Links
Unten
$

```

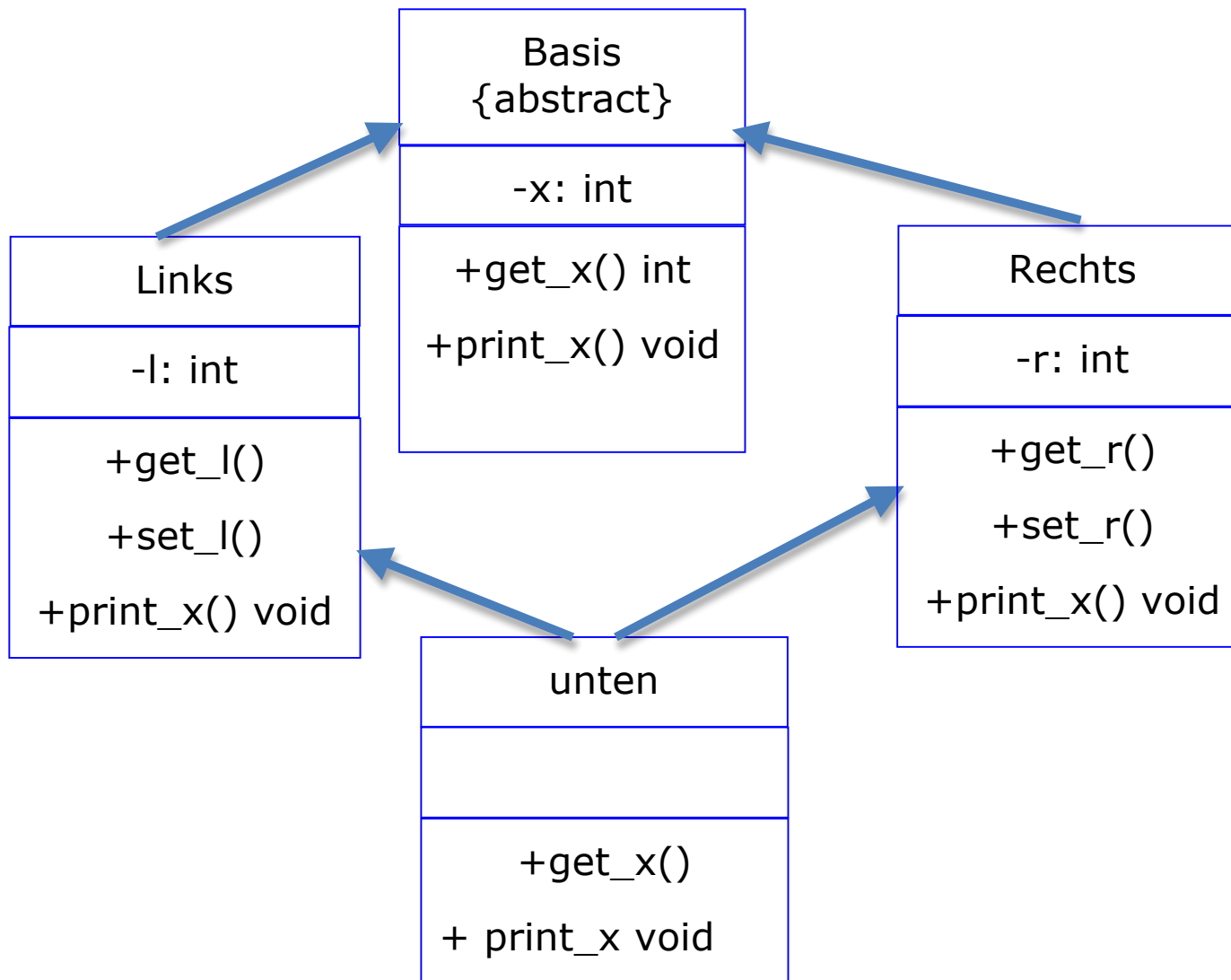
Hörsaalübung:

Was druckt das folgende Programm?

```
#include<iostream>    // basinit03.cpp
using namespace std;
class Basis {
    public:
        Basis() { cout << "Basis-Standardkonstruktor\n"; }
        Basis(char* a) { cout << a << endl; }
};
class Links : public Basis {
    public:
        Links(char* a) : Basis(a) { }
};
class Rechts : public Basis {
    public:
        Rechts(char* a) : Basis(a) { }
};
class Unten: public Links, public Rechts {
    public:
        Unten(char* a)
            : Links(a), Rechts(a) { }
};
int main() {
    Links li("Links");
    Unten x("Unten");
}
```

Hörsaalübung:

Realisieren Sie die C++ Klassen für folgendes UML-Schema:



Die Klasse Basis soll eine virtuelle Basisklasse sein.

Verwenden Sie folgendes Testprogramm:

```
int main() {
    Links li(1);
    Rechts re(2);
    Unten un(3);
    li.print_x();
    re.print_x();
    un.print_x();
    Basis p=new Unten(4,5);
    un.print_x();
}
```

Das Programm soll dann folgende Ausgabe erzeugen:

```
Links: 1
Rechts: 2
Unten: 3
Unten: 4
```

Hörsaalübung:

Realisieren Sie die C++ Klassen für folgendes UML-Schema inklusive eines Testprogramms:

