7

Vererbung

Dieses Kapitel behandelt die folgenden Themen:

- Vererbung und Objektorientierung
- Beziehung zwischen Ober- und Unterklasse
- Polymorphismus und seine Vorteile
- Mehrfachvererbung
- Typ eines Objekts zur Laufzeit eines Programms ermitteln

Der Vererbungsmechanismus zeichnet sich durch folgende Punkte aus:

- 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.

Ein Beispiel ist die Klassifizierung von Transportmitteln. Abbildung 7.1 auf der nächsten Seite zeigt sie in der UML-Notation (mehr zur UML siehe Seite 579). Die vererbende Klasse heißt *Oberklasse* oder *Basisklasse*, die erbende Klasse heißt Unterklasse oder *abgeleitete Klasse*. In der Literatur werden die Begriffe nicht einheitlich gebraucht. Im Falle von nur einer abgeleiteten Klasse ist die Oberklasse gleichzeitig die Basisklasse. Die Vererbung beschreibt eine *ist-ein*-Beziehung. Ein Fahrrad *ist ein* Landtransportmittel, ein Motorboot *ist ein* Wassertransportmittel. Die Vererbung ist eine gerichtete Beziehung, weil die Umkehrung im Allgemeinen nicht gilt: Ein Landtransportmittel ist nicht unbedingt ein Fahrrad.

Wie eine Klasse die Abstraktion von ähnlichen Eigenschaften und Verhaltensweisen ähnlicher Objekte ist, ist eine Oberklasse die *Abstraktion* oder *Generalisierung* von ähnlichen Eigenschaften und Verhaltensweisen der Unterklassen. Die Unterklasse fügt zu den allgemeinen Eigenschaften und Verhaltensweisen der Oberklasse nur die für diese Unterklasse spezifischen Dinge hinzu oder definiert das von der Oberklasse geerbte Verhalten neu. Die Unterklasse ist eine *Spezialisierung* der Oberklasse. Bei der Klassifikation von Objekten muss also nach Ähnlichkeiten und Unterschieden gefragt werden. Die Unterklasse *erbt* von der Oberklasse

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

Wenn eine Oberklasse bekannt ist, brauchen in einer zugehörigen Unterklasse nur die *Abweichungen* beschrieben zu werden. Alles andere kann *wieder verwendet* werden, weil es in der Oberklasse bereits vorliegt.

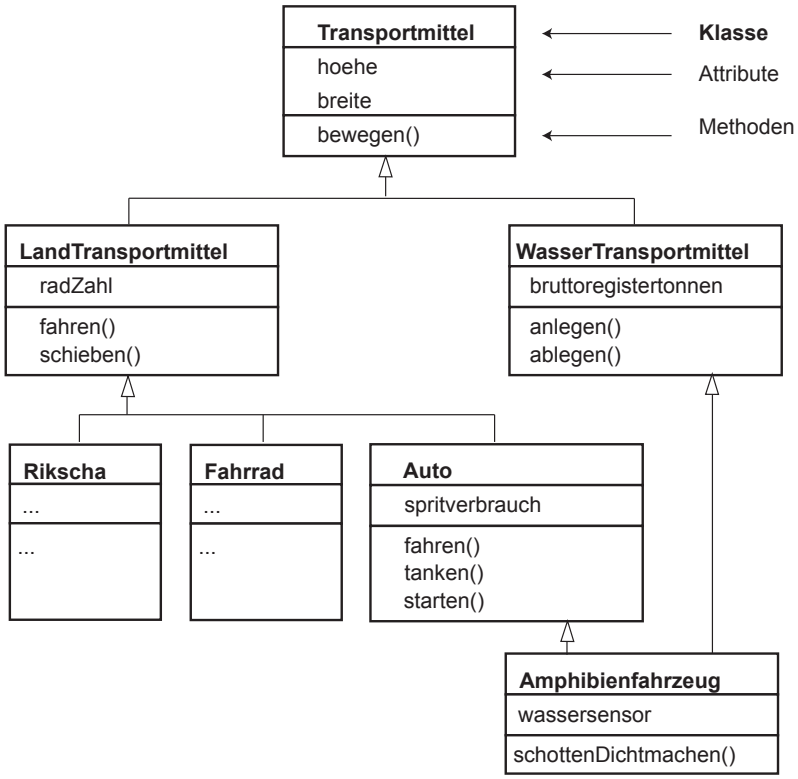

Abbildung 7.1: Vererbung von Daten und Methoden (unvollständige Klassen)

Die Menge der für ein Objekt zur Verfügung stehenden Methoden enthält die Methoden der Oberklasse(n) als Teilmenge. Umgekehrt sind alle Objekte einer Unterklasse (zum Beispiel Fahrräder) Teilmenge der möglichen Objekte einer Oberklasse (Landtransportmittel). Die Abstraktion wird durch »: `public`« ausgedrückt (siehe Syntaxdiagramm 7.2), es kann als »ist ein« oder »ist eine Art« gelesen werden.

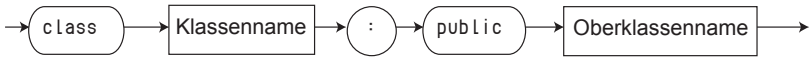

Abbildung 7.2: »`: public`« kennzeichnet Vererbung.

Ergänzend zur Abbildung 7.1 soll das Prinzip an der folgenden Formulierung in C++ gezeigt werden. Falls die Vererbung von Eigenschaften und Verhaltensweisen auf mehrere Oberklassen zurückgeführt werden kann, spricht man von Mehrfachvererbung (englisch *multiple inheritance*), hier gezeigt am Amphibienfahrzeug.

```
class Transportmittel {
   public:
    void bewegen();
   private:
    double hoehe, breite;
};

class LandTransportmittel : public Transportmittel { // erben
   public:
    void fahren();
    void schieben();
   private:
    int radZahl;
};

class WasserTransportmittel : public Transportmittel { // erben
   public:
    void anlegen();
    void ablegen();
   private:
    double bruttoregistertonnen;
};

class Auto : public LandTransportmittel {         // erben
   public:
    void fahren(); // überschreibt LandTransportmittel::fahren()!
    void tanken();
    void starten();
   private:
    double spritverbrauch;
};

class Amphibienfahrzeug:       // Mehrfachvererbung:
    public Auto, public WasserTransportmittel {
   public:
    void schottenDichtmachen();
   private:
    const char* wassersensor;
};
```

Wenn in C++ eine abgeleitete Klasse `Abgeleitet` von einer Oberklasse `Oberklasse` *erbt*, ist damit gemeint:

- Jedes Objekt `objAbgeleitet` vom Typ `Abgeleitet` enthält ein (anonymes) Objekt vom Typ `Oberklasse`, hier Subobjekt genannt, das entsprechend Speicher belegt. Dieses Subobjekt wird noch vor der Erzeugung von `objAbgeleitet` durch impliziten Aufruf des Oberklassenkonstruktors gebildet. Abbildung 7.3 zeigt, wie ein Oberklassenobjekt als Subobjekt in ein Objekt einer abgeleiteten Klasse eingebettet ist. Durch diesen Mechanismus wird erreicht, dass zu einem `Auto`-Objekt nicht nur der `spritverbrauch`, sondern auch `radZahl`, `hoehe` und `breite` als Attribute gehören.

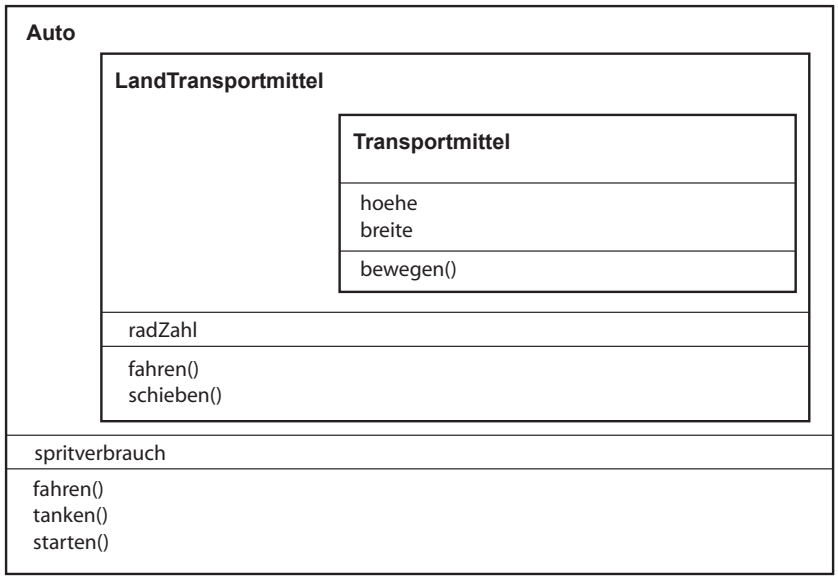

Abbildung 7.3: Einschluss von Subobjekten

- Jede Elementfunktion von `Oberklasse` kann auf ein Objekt des Typs `Abgeleitet` angewendet werden, sofern die Elementfunktion öffentlich zugänglich (`public`) ist. Die Funktion `bewegen()` ist benutzbar für ein Objekt `LandTransportmittel` ebenso wie für ein `Auto`, obwohl sie nicht speziell dort angegeben ist. Der Aufruf einer Operation für ein Objekt lässt nicht erkennen, ob sie der Klasse des Objekts oder einer Oberklasse zugeordnet ist.
- Die Klasse `Abgeleitet` kann Erweiterungen der Daten und zusätzliche Methoden enthalten, die keinen Bezug zur `Oberklasse` haben. Der `spritverbrauch` ist kennzeichnend für ein Auto. Es muss auch tanken, aber beides gilt nicht allgemein für ein (Land-) Transportmittel.
- Zusätzliche Methoden können in `Abgeleitet` deklariert werden, die in ihrer Signatur mit Elementfunktionen der Oberklasse übereinstimmen. Diese Methoden *überschreiben* die Elementfunktionen der Oberklasse bezüglich aller `Abgeleitet`-Objekte. In der Abbildung ist dies die Methode `fahren()`, die speziell für ein `Auto` entworfen wurde, weil die von `LandTransportmittel` geerbte Methode nicht geeignet ist.
- Eine Klasse ist ein *Datentyp* in C++. Eine abgeleitete Klasse kann als *Subtyp* der Oberklasse aufgefasst werden. Ein Objekt `objAbgeleitet` der abgeleiteten Klasse ist zuweisungskompatibel zu einem Objekt `objOberklasse` der Oberklasse. Die Zuweisung

```
objOberklasse = objAbgeleitet;
```

kopiert den Inhalt des in `objAbgeleitet` enthaltenen Subobjekts vom Typ `Oberklasse` nach `objOberklasse`. Die nur zu `objAbgeleitet` gehörenden spezifischen Daten werden nicht kopiert, weil in `objOberklasse` dafür kein Platz vorgesehen ist. Die Umkehrung `objAbgeleitet = objOberklasse` ist *nicht* möglich, weil der `Abgeleitet`-spezifische Teil undefiniert bleiben würde:

```
LandTransportmittel einLandTransportmittel;
Auto einAuto;
// ...
einLandTransportmittel = einAuto; // ok, aber Datenverlust
einAuto = einLandTransportmittel; // Fehler!
```

Auf die Subobjekte werde ich noch zurückkommen. In (älteren) Programmierhandbüchern findet man gelegentlich Beispiele, die Vererbung als *hat*-Beziehung einsetzen. Ein Kreis *hat* einen (Mittel-)Punkt, also wird von der Oberklasse Punkt geerbt. Dieses Vorgehen ist fehlerhaft, weil ein Kreis tatsächlich keine Spezialisierung eines Punktes darstellt: ein Kreis *ist kein* Punkt. Obwohl programmtechnisch möglich, sollte eine Vererbungshierarchie nicht *hat*-Beziehungen, auch Aggregation genannt, sondern Ebenen von Verallgemeinerungen darstellen. Im Folgenden wird die Basisklasse `GraphObj` (grafisches Objekt) als einfaches Beispiel verwendet. Sie wird in abgeleiteten Klassen (wie Linie, Rechteck, Dreieck) benutzt. Das Beispiel wird nach und nach entwickelt, ist also einigen Änderungen unterworfen. Alle möglichen auf dem Bildschirm sichtbaren Dinge sind grafische Objekte. Gemeinsam soll allen Objekten sein, dass jedes Objekt einen Bezugspunkt `referenzkoordinaten` in Pixelkoordinaten hat. Der Bezugspunkt soll nur über die Methode `bezugspunkt()` veränderbar sein, andererseits sollen die Koordinaten des Bezugspunktes von anderen gelesen werden können.

Klassen für grafische Objekte

Die Klasse `GraphObj` ist recht einfach. Das aggregierte Objekt `referenzkoordinaten` ist vom Typ `Ort`, der auf Seite 157 beschrieben ist. Nach `GraphObj` folgt eine Klasse `Strecke`, die von `GraphObj` erbt.

Listing 7.1: Klasse GraphObj, 1. Version

```
// cppbuch/k7/erben/graphobj.h
#ifndef GRAPHOBJ_H
#define GRAPHOBJ_H
#include"ort.h"

class GraphObj {                    // Version 1
  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& n0) {
        Ort temp = referenzkoordinaten;
        referenzkoordinaten = n0;
        return temp;
    }
    // Koordinatenabfrage
    int getX() const {
        return referenzkoordinaten.getX();
    }
    int getY() const {
        return referenzkoordinaten.getY();
    }
    // Standardimplementation:
    double flaeche() const {return 0.0;}
  private:
    Ort referenzkoordinaten;
};

// 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
```

Alle Methoden sind wegen ihrer Kürze `inline`. Innerhalb der am Ende der Header-Datei definierten globalen Funktion `Entfernung( const GraphObj &g1, const GraphObj &g2)` wird die schon vorher in *ort.h* für die Bezugspunkte des Typs `Ort` definierte gleichnamige Funktion aufgerufen. Der Compiler erkennt die richtige Funktion an Anzahl und Typ der Parameter. Eine kleine Besonderheit besteht darin, dass die Methode `bezugspunkt()` überladen ist. Wenn sie ohne Parameter aufgerufen wird, gibt sie den Bezugspunkt zurück. Wenn sie mit einem Ort als Parameter aufgerufen wird, setzt sie diesen Ort als neuen Bezugspunkt, gibt aber den vorherigen Bezugspunkt zurück. Diese Technik wird beim Setzen von Attributen häufig verwendet, weil sie einem aufrufenden Programm die Möglichkeit gibt, ein Attribut zu ändern und sich dabei den alten Wert zu merken, um ihn später wieder einzusetzen. Der Rückgabewert kann natürlich auch verworfen werden.

Die Fläche eines allgemeinen grafischen Objekts ist eigentlich nicht 0, sondern undefiniert. Auf diese Besonderheit wird in Abschnitt 7.6.2 eingegangen. Bis dahin bietet die Funktion `flaeche()` eine Standardimplementation für abgeleitete Klassen. Damit ist klar, dass diese Funktion in einer abgeleiteten Klasse möglicherweise neu definiert werden muss, nicht in der nachfolgend besprochenen Klasse `Strecke`, wohl aber in einer Klasse `Rechteck`. Eine `Strecke` *ist ein* `GraphObj`. In der Klassendeklaration wird diese Beziehung syntaktisch durch `: public` und den Namen der Oberklasse ausgedrückt. Die Bedeutung von `public` an dieser Stelle wird im folgenden Abschnitt 7.2 erläutert.

Listing 7.2: Klasse Strecke

```
// cppbuch/k7/erben/strecke.h
#ifndef STRECKE_H
```

```
#define STRECKE_H
#include"graphobj.h"

class Strecke : public GraphObj {  // erben von GraphObj
   public:
     Strecke(const Ort& ort1, const Ort& ort2)
     : GraphObj(ort1),    // Initialisierung des Subobjekts, siehe Kap. 7.1
       endpunkt(ort2)     // Initialisierung des Attributs
     { }                  // leerer Code-Block
     double laenge() const {
        return entfernung(bezugspunkt(), endpunkt);
     }
   private:
     Ort endpunkt;            // zusätzlich: 2. Punkt der Strecke
};
#endif // STRECKE_H
```

7.1 Vererbung und Initialisierung

Auf Seite 260 wurde darauf hingewiesen, dass jedes Objekt einer abgeleiteten Klasse ein anonymes Subobjekt der Oberklasse enthält. Der Oberklassenkonstruktor sollte bei der Initialisierung eines Objekts durch eine Liste (vergleiche Seite 156) explizit aufgerufen werden.

Nur ein Endpunkt der Strecke wird als Attribut angegeben, der andere wird geerbt (Attribut `GraphObj::referenzkoordinaten`). Der Konstruktor benötigt zwei Punkte zur Konstruktion der Strecke. Weil ein Objekt der Klasse `Strecke` ein anonymes Subobjekt der Klasse `GraphObj` enthält, ist der Anfangspunkt bereits durch die Referenzkoordinaten gegeben, und es ist nur noch ein Endpunkt als Attribut notwendig. *Falls* es einen Standardkonstruktor für die Klasse `GraphObj` gäbe, bräuchte man das Subobjekt nicht zu initialisieren und könnte den Konstruktor der Klasse `Strecke` wie folgt schreiben:

```
// nur bei Standardkonstruktor GraphObj() möglich, aber nicht empfehlenswert
Strecke(const Ort& ort1, const Ort& ort2) {
     bezugspunkt(ort1);   // geerbter Code der Oberklasse
     endpunkt = ort2;
}
```

Es gibt aber keinen Standardkonstruktor `GraphObj()`; außerdem ist die Initialisierung mit einer Initialisierungsliste generell vorzuziehen, weil das Objekt in *einem* Schritt mit den richtigen Werten initialisiert wird, also

```
Strecke(const Ort& ort1, const Ort& ort2)
: GraphObj(ort1),    // Initialisierung des Subobjekts
  endpunkt(ort2) {   // Initialisierung des Attributs
}                    // leerer Code-Block
```

Die Initialisierung innerhalb des Blocks `{...}` ist aufwendiger, weil die Konstruktoren für alle Objektelemente stets *vor* Betreten des Blocks aufgerufen werden und beliebige Daten eintragen, die dann innerhalb des Blocks neu zugewiesen werden müssen. Dasselbe gilt für die Initialisierung von Subobjekten, wie hier für das in einem `Strecke`-Objekt enthaltene Subobjekt des Typs `GraphObj`. Die Initialisierungsliste darf enthalten:

- Elemente der Klasse selbst, aber keine geerbten Elemente;
- Konstruktoraufrufe der Oberklassen.

Nach dem folgenden Abschnitt über Zugriffsschutz wird das Beispiel wieder aufgegriffen.

7.2 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 `private` und `public` gelten genauso in einer Vererbungshierarchie. Um abgeleiteten Klassen gegenüber der »Öffentlichkeit« weitgehende Rechte einräumen zu können, ohne den privaten Status mancher Elemente aufzugeben, gibt es einen weiteren Zugriffsspezifizierer:

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

Vererbung von Zugriffsrechten

Gegeben sei eine Oberklasse, von der eine weitere Klasse abgeleitet wird. Für die Vererbung der Zugriffsrechte gelten folgende *Regeln*, die weiter unten anhand einiger Beispiele verdeutlicht werden:

- `private`-Elemente sind in einer abgeleiteten Klasse nicht zugreifbar.
- In allen anderen Fällen gilt das jeweils restriktivere Zugriffsrecht, bezogen auf die Zugriffsrechte für ein Element und die Zugriffskennung der Vererbung einer Klasse. Beispiel: Ein `protected`-Element einer `private`-vererbten Klasse ist `private` in der abgeleiteten Klasse. Typischerweise werden jedoch Oberklassen `public` vererbt, sodass die Zugriffsrechte von Oberklassenelementen in abgeleiteten Klassen erhalten bleiben.

Tabelle 7.1 zeigt die Vererbung von Zugriffsrechten für den häufigen Fall der `public`-Vererbung. Die `private`- und `protected`-Vererbung werden Sie in Abschnitt 7.12 kennenlernen.

Tabelle 7.1: Zugriffsrechte bei `public`-Vererbung

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

Wenn anstatt `class` das Schlüsselwort `struct` geschrieben wird, ist die Voreinstellung `public`. Im Grunde ist »`struct s {`« nur eine Abkürzung für »`class s {public:`«. Man geht Zugriffskonflikten aus dem Weg, indem man überall `class` durch `struct` ersetzt – dann aber verletzt man das Prinzip der Datenkapselung! Besser ist es, sich genau zu überlegen, auf welche Daten und Funktionen der Oberklasse eine Klasse zugreifen darf, und im Zweifelsfall restriktiver vorzugehen. Das folgende Beispiel zeigt typische Möglichkeiten, die sich aus den Regeln ergeben. Alle Programmzeilen, die einen Zugriffsfehler ergeben, sind markiert.

```
class Oberklasse {
 private:           // Voreinstellung
   int oberklassePriv;
   void privateFunktionOberklasse();
 protected:
   int oberklassenProt;
 public:
   int oberklassePubl;
   void publicFunktionOberklasse();
};

// Oberklasse wird mit der Zugriffskennung public vererbt
class AbgeleiteteKlasse : public Oberklasse {
   int abgeleiteteKlassePriv;
 public:
   int abgeleiteteKlassePubl;

   void publicFunktionAbgeleiteteKlasse() {
      oberklassePriv = 1;              // Fehler: nicht zugreifbar
      // in einer abgeleiteten Klasse zugreifbar:
      oberklassenProt = 2;
      // generell zugreifbar
      oberklassePubl = 3;
   }
};

int main() {
   int m;
   AbgeleiteteKlasse objekt;
   m = objekt.oberklassePubl;
   m = objekt.oberklassenProt;            // Fehler: nicht zugreifbar
   m = objekt.oberklassePriv;             // Fehler: nicht zugreifbar
   m = Objekt.abgeleiteteKlassePubl;
   m = Objekt.abgeleiteteKlassePriv;      // Fehler: nicht zugreifbar
   Objekt.publicFunktionAbgeleiteteKlasse(); // ok
```

```
   // Aufruf geerbter Funktionen
   Objekt.publicFunktionOberklasse();
   Objekt.privateFunktionOberklasse();   // Fehler: nicht zugreifbar
}
```

7.3 Typbeziehung zwischen Ober- und Unterklasse

Eine abgeleitete Klasse kann als *Subtyp* der Oberklasse aufgefasst werden (siehe Seite 260). Daher ist ein Objekt der abgeleiteten Klasse zuweisungskompatibel zu einem Objekt der Oberklasse:

```
GraphObj g(O1);  // O1, O2, O3 = Objekte vom Typ Ort
Strecke s(O2, O3);
g = s;
```

Eine Strecke wird einem `GraphObj` zugewiesen, explizites Typumwandeln ist zwar möglich, aber nicht notwendig. Die Wirkung ist wie

```
g.referenzkoordinaten = s.referenzkoordinaten;
```

Eine direkte Zuweisung wäre natürlich wegen `private` nicht möglich. Der Endpunkt der Strecke wird *nicht* kopiert, da er in einem `GraphObj` nicht vorhanden ist. Es gibt also einen Informationsverlust. Die umgekehrte Zuweisung ist *nicht* möglich, weil dann Informationen undefiniert blieben: Die Zuweisung eines `GraphObj`-Objekts an ein `Strecke`-Objekt würde den zweiten Endpunkt undefiniert lassen. Die Typbeziehung zwischen Basisklasse und abgeleiteter Klasse kann umgangssprachlich am Beispiel verdeutlicht werden: »Alle Tannen sind Bäume, aber das Umgekehrte (alle Bäume sind Tannen) gilt nicht.« Dabei sind die Tannen Exemplare der Unterklasse und Bäume Exemplare der Oberklasse. Für Zeiger und Referenzen gilt entsprechend:

```
GraphObj& rg = g;     GraphObj *pg;
Strecke & rs = s;     Strecke *ps = &s;
rg = rs;              pg = ps;  // erlaubte Zuweisungen
```

Zeiger und Referenzen vom Oberklassentyp (`pg`, `rg`) beziehen sich auf das im Objekt `s` enthaltene anonyme Subobjekt vom Typ `GraphObj`. Aus diesem Grund kann *ohne zusätzlichen Code* die Entfernung der Bezugspunkte zweier Strecken ohne Umweg über die Bezugspunkte berechnet werden.[1] Aufgerufen wird hier die in *graphobj.h* deklarierte globale Funktion

```
double entfernung(const GraphObj& g1, const GraphObj& g2);
```

[1] In der Parameterliste von `entfernung()` wurde der Datentyp `GraphObj` per Referenz anstatt per Wert deklariert, weil es im Allgemeinen schneller ist, und weil manche Compiler die Typumwandlung innerhalb der Parameterliste nur eingeschränkt erlauben.

Hinweis

Die Typbeziehung kann nicht auf Arrays übertragen werden! Auch wenn `GraphObj` Oberklasse von `Strecke` ist, folgt daraus nicht, dass ein `GraphObj`-Array Oberklasse eines `Strecke`-Arrays ist – ein C-Array ist gar keine Klasse. Obwohl syntaktisch korrekt (d.h. compilierbar), ist die folgende Anweisung daher falsch.

```
Oberklasse* array = new Unterklasse[4]; // sinnlos.
```

Der Compiler sieht bei `array` nur den statischen Typ. Die nicht geerbten Attribute eines Unterklassenobjekts sind nicht zugreifbar, weil nur die Oberklassenanteile abgespeichert werden (englisch *object slicing*). Die Adressen `&array[i]` ($0 \leq i < 4$) liegen nur `sizeof(Oberklasse)` Bytes auseinander, nicht `sizeof(Unterklasse)`.

7.4 Code-Wiederverwendung

In den abgeleiteten Klassen können Methoden der Oberklasse wiederverwendet werden, zum Beispiel die Methoden `bezugspunkt()` und `flaeche()`. Eine Folgerung der durch die `public`-Vererbung repräsentierten *ist-ein*-Beziehung besteht darin, dass für eine Klasse alles möglich sein soll, was für die Oberklasse möglich ist, wenn auch vielleicht mit Anpassungen. Der Code der Oberklasse wird dabei wiederverwendet. In Abschnitt 7.12 werden wir andere Fälle kennenlernen, in denen die Wiederverwendung von Programmcode sinnvoll und möglich ist, obwohl keine *ist-ein*-Beziehung zwischen Klassen besteht. Ein kleines Programm zeigt, was mit den bisherigen Deklarationen und Definitionen für die Klassen `GraphObj` und `Strecke` möglich ist.

Listing 7.3: Beispiel mit Klasse Ort

```
// cppbuch/k7/erben/main.cpp
#include"strecke.h"
using namespace std;

int main() {
    // Definition zweier grafischer Objekte
    Ort nullpunkt;
    GraphObj g0(nullpunkt);
    Ort einOrt(10, 20);
    GraphObj g1(einOrt);
    // Ausgabe beider Bezugspunkte auf verschiedene Art
    cout << "g0.getX() = " << g0.getX() << endl;
    cout << "g0.getY() = " << g0.getY() << endl;

    Ort ort = g1.bezugspunkt();
    cout << "ort.getX() = " << ort.getX() << endl;
    cout << "ort.getY() = " << ort.getY() << endl;
    // Ausgabe der Entfernung
    cout << "Entfernung = " << entfernung(g0, g1) << endl;
```

```
    cout << "neuer Bezugspunkt für g0:" << endl;
    g0.bezugspunkt(ort);        // Rückgabewert wird hier ignoriert
    cout << "g0.bezugspunkt() = ";
    anzeigen(g0.bezugspunkt()); // ort.h, siehe Seite 157
    cout << "\n Entfernung = " << entfernung(g0, g1) << endl;
    Ort anf;
    Strecke s1(anf, ort);
    cout << "Strecke von ";
    anzeigen(anf);
    cout << " bis ";
    anzeigen(ort);
    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;
    // ...
}
```

Am Ende des Programmfragments wird zur Berechnung der Entfernung einmal

```
double entfernung(const Ort&, const Ort&);
```

aus *ort.h* aufgerufen. Anschließend wird die Entfernung noch einmal ausgegeben, wobei jetzt die Strecken direkt als Aufrufparameter dienen. Wie kann das angehen, wo doch bisher keine Funktion zur Entfernungsberechnung mit `Strecke`-Objekten in der Parameterliste beschrieben wurde? Der Grund liegt in der Typbeziehung zwischen Basisklasse und abgeleiteter Klasse.

7.5 Überschreiben von Funktionen in abgeleiteten Klassen

In diesem Abschnitt geht es um das Überschreiben von Funktionen innerhalb einer Vererbungshierarchie. Einen ähnlichen Mechanismus hatten wir bereits in Abschnitt 3.2.5 auf Seite 114 kennengelernt, der das Überladen von gleichnamigen Funktionen *mit unterschiedlicher Schnittstelle* behandelte. Dieser Abschnitt zeigt die Wirkungsweise für *Elementfunktionen mit derselben Schnittstelle* in abgeleiteten Klassen, die *Überschreiben* genannt wird. Die Methoden `bezugspunkt()` und `flaeche()` der Oberklasse `GraphObj`

können auch für die abgeleitete Klasse `Strecke` verwendet werden. Falls die gleiche Bedeutung gemeint ist, aber ein anderer Mechanismus zugrunde liegt, können Funktionen überschrieben werden. Um das zu zeigen, führen wir eine Klasse `Rechteck` ein:

Listing 7.4: Klasse Rechteck

```
// cppbuch/k7/erben/rechteck.h
#ifndef RECHTECK_H
#define RECHTECK_H
#include"graphobj.h"

class Rechteck : public GraphObj { // von GraphObj erben
   public:
     Rechteck(const Ort& ort, int h, int b)
     : GraphObj(ort), dieHoehe(h), dieBreite(b) {
     }

     double flaeche() const {
         // int-Überlauf vermeiden
         return static_cast<double>(dieHoehe) * dieBreite;
     }
   private:
     int dieHoehe, dieBreite;
};
#endif // RECHTECK_H
```

Am Beispiel der Flächenberechnung mit der Funktion `flaeche()` sehen wir das Prinzip des Überschreibens:

```
   Rechteck rechteck(Ort(0,0), 20, 50);
   cout << "rechteck.flaeche = "
        << rechteck.flaeche() << endl;   // 1000
```

Dieses Mal wird nicht wie bei der Klasse `Strecke` als Ergebnis 0 ausgegeben, sondern der Zahlenwert 1000. Die Funktion überschreibt jetzt `GraphObj::flaeche()`. Wenn aus irgendwelchen Gründen (in diesem Beispiel nicht sinnvoll) dennoch die Oberklassenfunktion aufgerufen werden soll, müssen der Klassenname und der Bereichsoperator `::` angegeben werden:

```
cout << rechteck.GraphObj::flaeche(); // null!
```

Im Gegensatz zu den überladenen Funktionen von Abschnitt 3.2.5 (Seite 114) können überschreibende Funktionen in abgeleiteten Klassen die gleiche Signatur haben, weil der Compiler sich die Klasse zusätzlich zur Signatur merkt. Das normale Überladen ist weiterhin möglich. Es gibt einen weiteren Unterschied: Das Überladen von Nicht-Elementfunktionen funktioniert nur innerhalb desselben Gültigkeitsbereichs (siehe Seite 115), während die überschreibenden Elementfunktionen in verschiedenen Klassen und damit unterschiedlichen Gültigkeitsbereichen sind.

Hinweis

Überschriebene Funktionen sollen grundsätzlich virtuell sein. Was das bedeutet und warum es so sein soll, wird im nächsten Abschnitt erläutert.

7.6 Polymorphismus

Polymorphismus heißt auf Deutsch Vielgestaltigkeit. Damit ist in der objektorientierten Programmierung die Fähigkeit einer Variable gemeint, zur *Laufzeit* eines Programms auf verschiedene Objekte zu verweisen. Anders formuliert: Erst zur Laufzeit eines Programms wird die zu dem jeweiligen Objekt passende Realisierung einer Operation ermittelt. In C++ wird die Einschränkung getroffen, dass die Objekte abgeleiteten Klassen zugeordnet sind. Ein Funktionsaufruf muss irgendwann an eine Folge von auszuführenden Anweisungen gebunden werden. Wenn es erst während der Ausführung des Programms geschieht, wird der Vorgang *dynamisches* oder spätes *Binden* genannt, andernfalls statisches oder frühes Binden. Eine zur Laufzeit ausgewählte Methode heißt *virtuelle Funktion*. Trotz der äußerlichen Ähnlichkeit und der ähnlichen Absicht dahinter sind Überladen und Polymorphismus verschiedene Konzepte. Virtuelle Funktionen haben *dieselbe* Schnittstelle in allen abgeleiteten Klassen, andernfalls würde man sie nicht brauchen.

7.6.1 Virtuelle Funktionen

Möglicherweise tritt der Fall ein, dass erst *zur Laufzeit* entschieden werden soll, welches Objekt angesprochen wird. Damit wird auch erst zur Laufzeit bestimmt, welche (Element-) Funktion verwendet werden soll, wie wir schon in Abschnitt 5.9 gesehen haben. In abgeleiteten Klassen können für solche Fälle *virtuelle* Funktionen der Basisklassen überladen werden.

Die überladenen Funktionen *müssen* in diesem Fall die *gleiche Signatur* haben, also den gleichen Namen und eine übereinstimmende Parameterliste, ansonsten werden sie wie normale überladene Funktionen aufgefasst. Der *Rückgabetyp* einer virtuellen Funktion in einer abgeleiteten Klasse muss mit dem Rückgabetyp in der Basisklasse *übereinstimmen* (Spezialisierungen sind dabei möglich, siehe unten).

Die Deklaration einer Funktion als `virtual` bewirkt, dass Objekten indirekt die Information über den Objekttyp mitgegeben wird. Dies wird ohne Zutun des Programmierers realisiert, indem im Speicherbereich eines Objekts zusätzlich zu den Objektattributen ein Zeiger *vptr* auf eine besondere Tabelle *vtbl* (virtual table = Tabelle von Zeigern auf virtuelle Funktionen) eingebaut wird. Die Tabelle gehört zu der Klasse des Objekts und enthält ihrerseits Zeiger auf die virtuellen Funktionen dieser Klasse.

Wenn nun eine virtuelle Funktion über einen Zeiger oder eine Referenz auf dieses Objekt angesprochen wird, weiß das Laufzeitsystem, dass die Funktion über den Zeiger *vptr* in der Tabelle gesucht und angesprochen werden muss. Es wird damit die *zu diesem Objekt* gehörende Funktion aufgerufen. Wenn die Klasse dieses Objekts aber *keine* Funktion mit gleicher Signatur hat, wird die entsprechende Funktion der *Oberklasse* gesucht und aufgerufen.

Um den internen Mechanismus muss man sich nicht kümmern. Es genügt zu wissen, dass Objekte durch den versteckten Zeiger *vptr* etwas größer werden und dass der Zugriff auf virtuelle Funktionen durch den Umweg über die Zeiger geringfügig länger dauert. Oft kann der Zugriff auf eine virtuelle Funktion bereits statisch aufgelöst werden, sodass der Compiler in der Lage ist, den Zugriff zu optimieren. Um den Unterschied zwischen virtu-

ellen und nicht-virtuellen Funktionen herauszuarbeiten, vergleiche ich beide Varianten anhand des bekannten Beispiels.

Verhalten einer nicht-virtuellen Funktion

Rufen wir uns die überschriebenen Funktionen `flaeche()` des obigen Beispiels in Erinnerung:

```
class GraphObj {
      // ....
      double flaeche() const { return 0.0;} // nicht virtuell
};
```

Die Fläche eines allgemeinen grafischen Objekts ist eigentlich nicht 0, sondern undefiniert. In Abschnitt 7.6.2 wird darauf eingegangen.

```
class Rechteck : public GraphObj {
     // ....
     double flaeche() const {           // nicht virtuell
        return static_cast<double>(dieHoehe) * dieBreite;
     }
};
```

Wir definieren ein grafisches Objekt `graphObj`, ein Rechteck `R` und einen Zeiger `graphObjPtr`, den wir auf `graphObj` zeigen lassen:

```
GraphObj graphObj(Ort(20, 20));
Rechteck rechteck(Ort(100, 100), 20, 50); // (x, y), Höhe, Breite
GraphObj *graphObjPtr;                    // Zeiger auf graphObj
```

Nun wird die Fläche beider Objekte ausgegeben. Dazu wird der Zeiger zuerst auf das grafische Objekt `graphObj` gerichtet. Über `graphObjPtr` wird die Funktion `flaeche()` aufgerufen. Dann (zur Programmlaufzeit!) wird `graphObjPtr` auf das Rechteck `rechteck` gerichtet und der Aufruf wiederholt. Zum Vergleich wird `rechteck.flaeche()` angezeigt:

```
graphObjPtr = &graphObj;    // Zeiger auf graphObj richten
cout << "graphObjPtr->flaeche() =" << graphObjPtr->flaeche() << endl;
graphObjPtr = &rechteck;    // Zeiger auf Rechteck richten
cout << "graphObjPtr->flaeche() =" << graphObjPtr->flaeche() << endl;
cout << "rechteck.flaeche()     =" << rechteck.flaeche()  << endl;
```

Was geschieht? Zweimal gibt es den Wert 0 und nur im dritten Aufruf den korrekten Wert 1000, obwohl `graphObjPtr` auf das Rechteck zeigt. Weil der Zeiger `graphObjPtr` vom Typ »Zeiger auf `GraphObj`« ist und keine Information über das Objekt hat, auf das er verweist, wird im ersten *und im zweiten* Fall `GraphObj::flaeche()` aufgerufen. Im zweiten Fall wird das anonyme Subobjekt vom Typ `GraphObj` angesprochen, das innerhalb des `rechteck`-Objekts liegt.

Verhalten einer virtuellen Funktion

Der Einsatz virtueller Funktionen bewirkt, dass Objekten die Typinformation über sich mitgegeben wird. Um das zu zeigen, erweitern wir das Beispiel um eine *virtuelle* Funktion `v_flaeche()`:

```
class GraphObj {
     // ....
     virtual double v_flaeche() const { return 0.0;}
};
```

```
class Rechteck : public GraphObj {
     // ....
     virtual double v_flaeche() const {
         return double(hoehe) * breite;
     }
};
```

Virtuelle Funktionen sind auch in allen nachfolgend abgeleiteten Klassen virtuell. Das Schlüsselwort `virtual` muss nur in der Basisklasse angegeben werden. Zu Dokumentationszwecken sollte es aber besser jeweils hingeschrieben werden. Das obige Beispiel wird jetzt mit der Funktion `v_flaeche()` in genau der gleichen Art und Weise wiederholt:

```
graphObjPtr = &graphObj;            // Zeiger auf graphObj richten
cout << "graphObjPtr->v_flaeche() =" << graphObjPtr->v_flaeche() << endl;
graphObjPtr = &rechteck;            // Zeiger auf Rechteck richten
cout << "graphObjPtr->v_flaeche() =" << graphObjPtr->v_flaeche() << endl;
```

Jetzt erhalten wir als Ergebnis 0 im ersten Fall (wie vorher), aber 1000 *im zweiten Fall*. Zur Laufzeit des Programms wird der Zeiger auf verschiedene Objekte gerichtet, und es wird die zum jeweiligen Objekt passende Funktion aufgerufen, nämlich im zweiten Fall `Rechteck::v_flaeche()`.

Ein Unterschied im Verhalten eines Objekts durch Aufruf einer virtuellen Funktion im Vergleich zu nichtvirtuellen Funktionen zeigt sich nur, wenn der Aufruf durch *Oberklassenzeiger oder -referenzen* geschieht statt über den Objektnamen. Im letzteren Fall gibt es ja ohnehin keine Zweifel über den Typ.

Welche Folgen hätte es, wenn die Forderung nach der gleichen Signatur nicht eingehalten wird? Dazu nehmen wir an, dass die Deklaration in der Klasse `Rechteck` einen Parameter `int` enthält (die Definition entsprechend; die Bedeutung des Parameters ist beliebig und spielt hier keine Rolle):

```
double v_flaeche(int z);  // Fehler
```

- Es gibt nun keine Funktion `Rechteck::v_flaeche()` mehr mit einer passenden Signatur, sodass der Aufruf `graphObjPtr->v_flaeche()` im Gegensatz zum obigen Beispiel als `graphObjPtr->GraphObj::v_flaeche()` interpretiert wird und daher 0 ergibt – zu wenig für ein 20*50 Rechteck! Weil für das Objekt keine passende Funktion vorhanden ist, wird zur Oberklasse »durchgegriffen«. Im Falle mehrerer Vererbungsebenen wird die Hierarchie in Richtung der Basisklasse so lange durchsucht, bis eine passende Funktion gefunden wird. Dieser Vorgang ist durch den Einsatz interner Tabellen sehr schnell.
- `rechteck.v_flaeche()` (ohne `int`-Parameter) wäre nicht mehr möglich, weil die Basisklassenfunktion `GraphObj::v_flaeche()` von `rechteck` nicht mehr zugreifbar ist. `rechteck.v_flaeche(99)` wäre zulässig.

- `graphObjPtr->v_flaeche(100)` wäre nicht möglich, weil `v_flaeche(int)` *keine* virtuelle Funktion ist und damit ausschließlich zur Klasse Rechteck gehört und nicht zu einem `graphObjPtr` als »Zeiger auf `GraphObj`« passt.

Eigenschaften virtueller Funktionen

Als wesentliche Merkmale virtueller Funktionen lassen sich zusammenfassen:

- Virtuelle Funktionen dienen zum Überladen bei gleicher Signatur und bei gleichem Rückgabetyp. Erlaubte Erweiterung: Wenn der Rückgabetyp einer virtuellen Funktion eine Referenz auf eine Klasse ist, dann darf der Rückgabetyp der entsprechenden Funktion in der abgeleiteten Klasse eine Referenz auf die abgeleitete Klasse sein. Das Gleiche gilt für Zeiger anstelle von Referenzen.
- Der Aufruf einer nicht-virtuellen Elementfunktion 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. Der Aufruf von virtuellen Funktionen über Basisklassenzeiger oder -referenzen, die auf ein Objekt einer abgeleiteten Klasse zeigen, bezieht sich auf die genau *zu diesem Objekt* passende Funktion.
- 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. Ein Programm, das Zeiger oder Referenzen auf die Basisklasse benutzt, kann damit sehr leicht um abgeleitete Klassen erweitert werden, weil der Aufruf einer virtuellen Funktion über Zeiger oder Referenzen sicherstellt, dass die zum referenzierten Objekt gehörende Realisierung der Funktion aufgerufen wird (siehe weiteres Beispiel in Abschnitt 7.6.2).
- Der vorstehende Punkt gilt auch für Destruktoren. Wenn es überhaupt virtuelle Funktionen in einer Klasse gibt, sollte der Destruktor als `virtual` deklariert werden. Die Definition der Klasse `GraphObj` muss um die Zeile

  ```
  virtual ~GraphObj() {}
  ```

  erweitert werden. Einzelheiten folgen ab Seite 280.

Aus diesen Punkten lässt sich eine wichtige Regel ableiten: Nicht-virtuelle Funktionen einer Basisklasse sollen *nicht* in abgeleiteten Klassen überschrieben werden! Oder anders ausgedrückt: Wenn ein Überschreiben notwendig erscheint, sollte die Funktion in der Basisklasse als `virtual` deklariert werden. Der Grund liegt darin, dass die Bedeutung (= das Verhalten) eines Programms sich nicht ändern sollte, wenn auf eine Methode über den Objektnamen oder über Basisklassenzeiger bzw. -referenzen zugegriffen wird.

Das folgende Programm ist ein erweitertes und erläutertes Beispiel aus [ES]. Es zeigt in konzentrierter Form die Eigenschaften virtueller Funktionen. Zum besseren Verständnis sollten wir uns daran erinnern, dass ein Name (englisch *identifier*), der in einem Gültigkeitsbereich (*scope*) definiert wird, alle außerhalb dieses Bereichs getroffenen Definitionen desselben Namens überdeckt. Diese Regel gilt unabhängig von der `virtual`-Eigenschaft von Funktionen. Namen in einer Basisklasse sind in einem äußeren Gültigkeitsbereich relativ zu Namen in einer abgeleiteten Klasse.

```
class Basisklasse {
  public:
    virtual void vf1();
    virtual void vf2();
    virtual void vf3();
    virtual Basisklasse* vf4();
    virtual Basisklasse& vf5();
    void f();
};

class AbgeleiteteKlasse : public Basisklasse {
  public:
    virtual void vf1();
    virtual void vf2(int);
    virtual char vf3();          // Fehler! falscher Rückgabetyp
    virtual AbgeleiteteKlasse* vf4(); // geänderter Rückgabetyp
    virtual AbgeleiteteKlasse& vf5(); // geänderter Rückgabetyp
    void f();
};

int main () {
    AbgeleiteteKlasse d;
    Basisklasse *bp = &d;
    // In der folgenden Anweisung wird richtig AbgeleiteteKlasse::vf1()
    // aufgerufen, weil vf1() virtuell ist.
    bp->vf1();           // AbgeleiteteKlasse::vf1()
    // Eine Funktion AbgeleiteteKlasse::vf2(), das heißt ohne Parameter, gibt
    // es nicht. Deshalb wird durch bp->vf2(); die Funktion Basisklasse::vf2()
    // aufgerufen.
    bp->vf2();           // Basisklasse::vf2()

    // Die Funktion Basisklasse::vf2() ist von d aus nicht mehr zugreifbar. Mit
    // dem int-Parameter gibt es kein Problem, weil vf2(int) in der abgeleiteten
    // Klasse deklariert ist.
    d.vf2();             // Fehler!
    d.vf2(7);            // ok

    // Obwohl bp auf ein Objekt der abgeleiteten Klasse zeigt, ist bp->vf2(7) nicht
    // möglich, weil eine Funktion mit int-Parameter in der Basisklasse nicht existiert.
    bp->vf2(7);          // Fehler!

    // bp->f() ruft Basisklasse::f() für das in d enthaltene Subobjekt auf, weil
    // f() nicht virtuell ist.
    bp->f();

    AbgeleiteteKlasse* dp;
    dp = d.vf4();        // AbgeleiteteKlasse::vf4()
    d.vf5();             // AbgeleiteteKlasse::vf5()
    d.vf5().vf1();
    // Eine Referenz kann als Alias-Name für ein Objekt aufgefasst werden.
    // Weil d.vf5() eine Referenz auf AbgeleiteteKlasse
    // zurückgibt, wird der Aufruf der Funktion d.vf5().vf1()
    // interpretiert als (d.vf5()).vf1(). Typischerweise wird das
```

```
    // (möglicherweise veränderte) Objekt selbst als Referenz zurückgegeben.
    // Die Zeile kann dann in zwei Teile zerlegt werden:
    // d.vf5();
    // d.vf1();
}   // Ende von main
```

7.6.2 Abstrakte Klassen

In vielen Fällen sollte die Basisklasse einer Hierarchie sehr allgemein sein und Code enthalten, der aller Voraussicht nach nicht geändert werden muss. Es ist dann oft nicht notwendig oder gewünscht, dass Objekte dieser Klassen angelegt werden. Diese *abstrakten Klassen* dienen ausschließlich als *Ober- oder Basisklassen*. Objekte werden nur von den abgeleiteten Klassen erzeugt, die dann jeweils ein Subobjekt vom Typ der abstrakten Basisklasse enthalten. Das syntaktische Mittel, um eine Klasse abstrakt zu machen, sind *rein virtuelle* Funktionen (englisch *pure virtual*). Abstrakte Klassen haben mindestens eine rein virtuelle Funktion, die typischerweise *keinen* Definitionsteil hat, aber einen haben kann. Durch die rein virtuelle Funktion wird gewährleistet, dass stets die zum Objekttyp passende Methode aufgerufen wird. Definieren einer abstrakten Klasse heißt also nichts anderes, als ein gemeinsames Protokoll für alle abgeleiteten Klassen zu definieren. Eine rein virtuelle Funktion wird durch Ergänzung von »= 0« deklariert:

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

Unser Beispiel mit den grafischen Objekten ist wie geschaffen zur Anwendung abstrakter Klassen, denn ein grafisches Objekt ist entweder ein Rechteck, ein Polygon, ein Kreis oder was man sich sonst noch ausdenken kann, aber niemals ein grafisches Objekt »an sich«. Ein *allgemeines* grafisches Objekt kann *nicht* gezeichnet werden und hat keine definierte Fläche. Also benötigen wir in einem Programm *keine* Objekte der Klasse `GraphObj`, außer natürlich als (versteckte) Subobjekte von Rechtecken, Kreisen und so weiter. Wir können die Klasse `GraphObj` daher als abstrakte Klasse formulieren, indem wir `flaeche()` in eine rein virtuelle Funktion umwandeln:

```
virtual double flaeche() const = 0;
```

Klassen, von denen Objekte erzeugt werden können, nennt man konkrete Klassen, wenn der Unterschied zu abstrakten Klassen betont werden soll. Wenn eine konkrete Klasse von einer abstrakten Klasse erbt, muss sie zu den rein virtuellen vorgegebenen Funktionsprototypen konkrete Implementierungen bereitstellen, zum Beispiel um die Fläche als Produkt von Höhe mal Breite zu berechnen.

Wenn in einer vermeintlich konkreten Klasse eine Implementierung fehlt, zum Beispiel, weil sie vergessen wurde, ist sie tatsächlich nicht konkret, sondern selbst abstrakt. Die Eigenschaft »abstrakt« wird auf Klassen ohne oder mit unvollständiger Implementation vererbt.Falls versucht wird, von einer Klasse dieser Art ein Objekt zu erzeugen, gibt es eine Fehlermeldung des Compilers.

Das unten stehende Beispiel zeigt eine typische Art, abstrakte Klassen und virtuelle Funktionen einzusetzen. Wir erweitern dazu die Klasse `GraphObj` um eine Funktion `zeichnen()`, die das Objekt auf dem Bildschirm darstellen soll. Die Funktion sieht natürlich für Kreise und Rechtecke unterschiedlich aus, der Aufruf jedoch beziehungsweise die Schnittstelle

ist stets die gleiche. Um das Beispiel nicht mit graphikspezifischen Details zu überfrachten, besteht die einzige Aufgabe der Funktion `zeichnen()` darin, eine Meldung auf dem Bildschirm auszugeben.

Weitere Besonderheiten des Beispiels sind wie folgt:

- Die Methode `flaeche()` ist in der Klasse `GraphObj` als rein virtuelle Funktion ohne Definition deklariert.
- Im Unterschied dazu stellt die ebenfalls rein virtuelle Methode `zeichnen()` eine Standarddefinition bereit, die von den abgeleiteten Klassen benutzt wird.
- Die Klasse `Quadrat`[2] braucht die Funktion `flaeche()` nicht neu zu implementieren, weil die Implementierung von der Klasse `Rechteck` geerbt wird. Dies gilt auch für `zeichnen()`, wenn auf eine Unterscheidung bei der Ausgabe verzichtet werden soll.
- Die `while`-Schleife im Main-Programm zeigt die Stärke des Polymorphismus. Ohne dass man sich um den Typ der einzelnen Objekte kümmern muss, wird stets die richtige Funktion aufgerufen.

Der Übersichtlichkeit halber und weil später Bezug darauf genommen wird, sind die Dateien mit den Änderungen vollständig wiedergegeben.

Listing 7.5: Klasse GraphObj, 2. Version

```
// cppbuch/k7/abstrakt/graphobj.h
#ifndef GRAPHOBJ_H
#define GRAPHOBJ_H
#include"ort.h"   // enthält #include<iostream>

class GraphObj {                    // Version 2
  public:
    GraphObj(const Ort& einOrt)     // allg. Konstruktor
    : referenzkoordinaten(einOrt) {}
    virtual ~GraphObj() {}          // virtueller Destruktor

    const Ort& bezugspunkt() const { // Bezugspunkt ermitteln
       return referenzkoordinaten;
    }
    // alten Bezugspunkt ermitteln und gleichzeitig neuen wählen
    Ort bezugspunkt(const Ort& nO) {
        Ort temp = referenzkoordinaten;
        referenzkoordinaten = nO;
        return temp;
    }
    // Koordinatenabfrage
    int getX() const { return referenzkoordinaten.getX(); }
    int getY() const { return referenzkoordinaten.getY(); }
    // rein virtuelle Methoden
    virtual double flaeche() const = 0;
    virtual void zeichnen() const = 0;
  private:
    Ort referenzkoordinaten;
  };
```

[2] Zur Diskussion, ob ein Quadrat ein Rechteck im Sinn der objektorientierten Programmierung ist, siehe unten Seite 282.

```
// Die Standardimplementierung einer rein virtuellen Methode
// muss außerhalb der Klassendefinition stehen:
inline void GraphObj::zeichnen() const {
    std::cout << "Zeichnen: ";
}
// Die Entfernung zwischen zwei 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. Andernfalls wären die Klassen ebenfalls abstrakt, und es könnte keine Instanzen von ihnen geben. Ein Endpunkt der Strecke wird von `GraphObj` geerbt, der andere ist Attribut der Klasse.

Listing 7.6: Klasse Strecke

```
// cppbuch/k7/abstrakt/strecke.h
#ifndef STRECKE_H
#define STRECKE_H
#include"graphobj.h"

class Strecke : public GraphObj { // erben von GraphObj
  public:
    // Initialisierung von Subobjekt und Attribut mit Initialisierungsliste
    Strecke(const Ort& ort1, const Ort& ort2)
    : GraphObj(ort1), endpunkt(ort2) {
    }

    double laenge() const {
        return entfernung(bezugspunkt(), endpunkt);
    }

   // Definition der rein virtuellen Methoden
   virtual double flaeche() const {
      return 0.0;
   }

   virtual void zeichnen() const {
       GraphObj::zeichnen();
       std::cout << "Strecke von ";
       anzeigen(bezugspunkt());
       std::cout << " bis ";
       anzeigen(endpunkt);
       std::cout << std::endl;
   }
  private:
    Ort endpunkt;           // zusätzlich: 2. Punkt der Strecke
};
#endif // STRECKE_H
```

Listing 7.7: Klasse Rechteck

```
// cppbuch/k7/abstrakt/rechteck.h
#ifndef RECHTECK_H
#define RECHTECK_H
#include"graphobj.h"

class Rechteck : public GraphObj { // von GraphObj erben
  public:
    Rechteck(const Ort& ort, int h, int b)
    : GraphObj(ort), dieHoehe(h), dieBreite(b) {}

    // wird von Quadrat benötigt
    int hoehe() const {
        return dieHoehe;
    }
    int breite() const {
        return dieBreite;
    }

    // Definition der rein virtuellen Methoden
    virtual double flaeche() const {
       return static_cast<double>(dieHoehe) * dieBreite;
    }

    virtual void zeichnen() const {
       GraphObj::zeichnen();
       std::cout << "Rechteck (h x b = " << dieHoehe << " x "
                 << dieBreite << ") an der Stelle ";
       anzeigen(bezugspunkt());
       std::cout << std::endl;
    }
  private:
    int dieHoehe, dieBreite;
};
#endif
```

Listing 7.8: Klasse Quadrat

```
// cppbuch/k7/abstrakt/quadrat.h
#ifndef QUADRAT_H
#define QUADRAT_H
#include"rechteck.h"

class Quadrat : public Rechteck { // siehe Text
 public:
   Quadrat(const Ort& ort, int seite)
   : Rechteck(ort, seite, seite) {
   }

   // Definition der rein virtuellen Methoden
   virtual void zeichnen() const {
       GraphObj::zeichnen();
       std::cout << "Quadrat (Seitenlaenge = " << hoehe()
```

```
                << ") an der Stelle ";
        anzeigen(bezugspunkt());
        std::cout << std::endl;
    }
    // Die Methoden bezugspunkt(), flaeche(), hoehe(), breite() werden geerbt.
};
#endif // QUADRAT_H
```

Das folgende Beispielprogramm ruft die Methoden der grafischen Objekte polymorph auf. Entscheidend ist nicht der (statische) Typ des Zeigers, den der Compiler sieht, sondern der polymorphe oder dynamische Typ, das heißt, der Typ des Objektes, auf das der Zeiger zur Laufzeit verweist. Die Elemente des Feldes `GraphObjZeiger` sind alle vom statischen Typ `GraphObj*`, sie verweisen aber zur Laufzeit auf Objekte von Klassen, die von `GraphObj` abgeleitet wurden.

Dasselbe gilt für Referenzen. So sind die Referenzen `R_Ref`, `S_Ref` und `Q_Ref` im Programm alle vom Typ der Basisklasse `GraphObj`. Die Referenzen verweisen aber zur Laufzeit auf Objekte verschiedener Typen, nämlich der Klassen `Rechteck`, `Strecke` und `Quadrat`.

Das Beispiel ist sehr leicht um beliebige grafische Klassen erweiterbar (zum Beispiel Kreis, Ellipse, Polygon ...), ohne dass die Anweisung »Zeichnen aller Objekte« überhaupt geändert werden muss.

Listing 7.9: Anwendung von Polymorphismus

```
// cppbuch/k7/abstrakt/main.cpp
#include"strecke.h"
#include"quadrat.h" // schließt rechteck.h ein

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);
    // C-Array mit Basisklassenzeigern, initialisiert mit
    // den Adressen der Objekte und 0 als Endekennung
    GraphObj* graphObjZeiger[] = {&r, &s, &q, 0};

    // Ausgabe der Fläche aller Objekte
    int i = 0;
    while(graphObjZeiger[i]) {
       std::cout << "Fläche = " << graphObjZeiger[i++]->flaeche() << std::endl;
    }
    // Zeichnen aller Objekte
    i = 0;
    while(graphObjZeiger[i]) {
         graphObjZeiger[i++]->zeichnen();
    }
    // Referenzen statt Zeiger
    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();
    q_ref.zeichnen();
}
```

7.6.3 Virtueller Destruktor

Ein virtueller Destruktor sorgt ähnlich wie virtuelle Funktionen dafür, dass Zeigern die Typinformation über ein Objekt zur Verfügung steht und deshalb die Speicherfreigabe exakt erfolgt. Über einen Zeiger `px` vom Typ »Zeiger auf Basisklasse«, der auf ein Objekt `X` einer abgeleiteten Klasse zeigt, kann zur Compilierzeit, also statisch, nur die Größe des Subobjekts (vom Typ Basisklasse) von `X` ermittelt werden. Die Operation `delete` auf `px` angewendet, gäbe ohne virtuellen Destruktor Platz entsprechend `sizeof(*px)` frei, also zu wenig, sodass langlaufende Programme Speicherprobleme bekommen können. Interessant ist hier aber der *dynamische* Typ, also der Typ des Objekts `X`, denn für diesen Typ muss der Speicherplatz freigegeben werden. Das Beispielprogramm demonstriert die Notwendigkeit für virtuelle Destruktoren.

Listing 7.10: Beispielprogramm mit virtuellem Destruktor

```
// cppbuch/k7/virtdest.cpp
#include<iostream>
using namespace std;
#define PRINT(X) cout << (#X) << " = " << (X) << endl

class Basis {
    int bWert;
  public:
    Basis(int b = 0)
    : bWert(b) {}
    virtual ~Basis() {                          // virtueller Destruktor!
        cout << "Objekt " << bWert << " Basis-Destruktor aufgerufen!\n";
    }
};
class Abgeleitet : public Basis {
    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));
    Abgeleitet *pa = new Abgeleitet(2, 2.2);
    PRINT(sizeof(*pa));
    Basis *pba = new Abgeleitet(3, 3.3);
    PRINT(sizeof(*pba));
    cout << "pb löschen:\n";
    delete pb;                          // ok
```

```
    cout << "pa löschen:\n";
    delete pa;                          // ok
    cout << "pba löschen:\n";
    delete pba;          // ok nur mit virtuellem Destruktor!
}
```

Das Makro PRINT ist auf Seite 132 erklärt. Die Basisklassenobjekte werden durch eine ganze Zahl, die Objekte der abgeleiteten Klasse durch eine Zahl des Typs `double` identifiziert. Es werden ein Basisklassenobjekt und zwei Objekte der abgeleiteten Klasse erzeugt. Im Beispielprogramm werden 4 Bytes für `int`, 8 Bytes für `double` und 4 Bytes für den versteckten Zeiger *vptr* (Seite 270) benötigt. Es liefert die Ausgabe (die Zahlen können auf Ihrem System andere sein):

*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!

`sizeof` gibt die statisch aus dem Typ des Zeigers ermittelbare Objektgröße an. `delete` ruft den korrekten Destruktor auch im letzten Fall auf. *Ohne* das Schlüsselwort `virtual` würde nur jeweils der Destruktor aufgerufen, der zum Typ des Zeigers passt. Ausgabe bei *Fehlen* des Schlüsselworts `virtual`:

*sizeof(*pb) = 4* veränderte Werte!
*sizeof(*pa) = 12*
*sizeof(*pba) = 4*

. . . und so weiter wie oben, aber es *fehlt* die Ausgabe

Objekt 3.3 Abgeleitet-Destruktor aufgerufen!

Man sieht daran, dass nur der Basisklassenanteil des Objektes `*pba` freigegeben wurde, entsprechend dem statischen Datentyp von `pba`. Der Rest bleibt im Speicher hängen.

Merke:
Virtuelle Destruktoren sollten immer dann verwendet werden, wenn von der betreffenden Klasse abgeleitet wird oder nicht auszuschließen ist, dass von ihr zukünftig durch Ableitung neue Klassen gebildet werden.

An den nun ausgegebenen, veränderten `sizeof`-Werten ist ferner zu erkennen, dass die Objekte nunmehr *keine* besondere Typinformation enthalten, das heißt in diesem Fall, dass die Tabelle der Zeiger auf virtuelle Funktionen (siehe Seite 270) nicht existiert. Der Effekt ist hier mit `sizeof` natürlich nur deshalb erkennbar, weil es keine weitere virtuelle Funktion gibt (die die Objektgröße verändern würde).

Immer wenn Basisklassenzeiger oder -referenzen auf dynamisch erzeugte Objekte benutzt werden, was normalerweise im Zusammenhang mit der Benutzung virtueller Methoden steht, sollte ein virtueller Destruktor eingesetzt werden. Wenn eine Klasse von anderen per Vererbung genutzt werden kann, kann die Art der zukünftigen Benutzung nicht bekannt sein. Also: Destruktoren immer virtuell machen, falls vererbt werden könnte!

Übung

7.1 Auf Seite 268 wurde die Funktion `flaeche()` für ein Objekt der Klasse `Strecke` aufgerufen. Ist der Aufruf auch möglich, wenn `GraphObj` als *abstrakte* Klasse definiert ist?

7.7 Probleme der Modellierung mit Vererbung

Dass Vererbung die geeignete programmiertechnische Umsetzung einer *ist-ein-* oder *ist-eine-Art*-Beziehung zwischen Objekten ist, kann durchaus fraglich sein. Das Für und Wider wird hier anhand einiger Grenzfälle diskutiert.

Eine abgeleitete Klasse kann als Subtyp der Oberklasse aufgefasst werden. *Ein Objekt einer abgeleiteten Klasse kann damit stets an die Stelle eines Objekts der Oberklasse treten* – es sind ja alle Methoden der Oberklasse vorhanden, wenn auch möglicherweise überschrieben (Liskovsches Substitutionsprinzip, siehe [Lis]). Dies erscheint auf den ersten Blick einleuchtend. Dennoch gibt es Fälle, in denen dieser Satz der Konvention oder der menschlichen Erfahrung widerspricht. Ein einfaches Beispiel soll dies erläutern.

Seit Euklid, also seit mehr als 2000 Jahren, ist bekannt, dass ein Quadrat ein Rechteck und ein Kreis eine Ellipse ist. Genauer formuliert, ist ein Quadrat ein Rechteck mit gleichen Seitenlängen, also ein Spezialfall eines Rechtecks. Die Spezialisierung wird in C++ durch `public`-Vererbung ausgedrückt:

```
class Quadrat : public Rechteck { ...};
```

Nun kann man sich aber eine Klasse `Rechteck` vorstellen, die es erlaubt, die Seiten ungleichmäßig zu ändern; denken wir nur an einen grafischen Editor, mit dem ein Rechteck in verschiedene Richtungen auseinandergezogen werden kann:

```
class Rechteck {
   public:
      virtual void hoeheAendern(int neu) { hoehe = neu;}
      virtual void breiteAendern(int neu) { breite = neu;}
      // ...
    private:
      int hoehe;
      int breite;
};
```

Vordergründig ist klar, dass diese Methoden in einer Klasse `Quadrat` nichts zu suchen haben, wenn die Forderung aufrechterhalten bleiben soll, dass ein `Quadrat`-Objekt stets an die Stelle eines `Rechteck`-Objekts treten kann. Die manchmal empfohlene »Lösung«, dass zwischen `Quadrat` und `Rechteck` gar keine Vererbungsbeziehung besteht und beide Klassen von einer abstrakten Klasse `Viereck` erben sollten, ist nicht sinnvoll, weil das Problem nur auf eine andere Ebene verschoben wird: Ein allgemeines Viereck kann man diagonal zu einer Raute verformen, ein Rechteck nicht, wenn es eines bleiben soll. Um solche Fälle vernünftig darstellen zu können, wird Vererbung gelegentlich benutzt, um *Einschränkungen* (englisch *constraints*) einer Oberklasse zu formulieren (*inheritance for restriction*, siehe nachfolgendes Beispiel). Dennoch sollte man sorgfältig überlegen, ob es nicht andere Wege gibt.

Ein großer Vorteil der Objektorientierung besteht darin, dass die Begriffe der Anwendung weit mehr als in nicht-objektorientierten Programmiersprachen durchgängig von der Analyse zum Code benutzbar sind. Davon sollte man nicht ohne schwerwiegenden Grund abweichen – den es durchaus geben kann. Die Beziehung »ein Quadrat ist ein Rechteck« ist die natürliche Beziehung in einer mathematisch-geometrischen Anwendung, die beibehalten werden sollte. Nur vom Standpunkt der Implementierung her sollte man sich überlegen, ob der zusätzliche Aufwand in Kauf genommen werden soll, Platz für zwei Seitenlängen zu spendieren, obwohl nur eine nötig ist.

In der objektorientierten Programmierung geht es unter anderem um einen *Vertrag* mit dem Benutzer einer Klasse. Der Benutzer muss sich darauf verlassen können, dass die Klasse den Vertrag einhält, das heißt, dass die Seitenlängen im Quadrat untereinander stets gleich bleiben.

Wenn die Klasse `Quadrat` von der Klasse `Rechteck` erben soll, lässt sich das Einhalten der Bedingung gleicher Seitenlängen leicht bewerkstelligen:

```
class Quadrat : public Rechteck { // empfehlenswert?
   public:       // ... (Konstruktor usw. weggelassen)
      virtual void hoeheAendern(int neu) {
         Rechteck::hoeheAendern(neu);
         Rechteck::breiteAendern(neu);
      }
      virtual void breiteAendern(int neu) {
         hoeheAendern(neu);
      }
};
```

Die vertragliche Einschränkung, dass Höhe und Breite eines Quadrats stets gleich sind, wird an alle von `Quadrat` abgeleiteten Klassen vererbt. Eine Möglichkeit, ohne Vererbung auszukommen und ohne auf die Funktionen eines Rechtecks zu verzichten, soweit sie angemessen sind, zeigt das folgende Beispiel, in dem ein Quadrat ein Rechteck *benutzt*:

```
class Quadrat {                  // empfehlenswert?
   public:
      Quadrat(const Ort& ort, int seite)
      : r(ort, seite, seite) { // privates Rechteck initialisieren
      }
      virtual void seiteAendern(int neu) {
         r.hoeheAendern(neu);
```

```
        r.breiteAendern(neu);
    }
    // ... viele weitere Funktionen, die Methoden der Klasse Rechteck benutzen
  private:
    Rechteck r;
};
```

Die Methoden des Rechtecks sind für Quadratbenutzer nicht mehr zugreifbar, aber die Klasse `Quadrat` macht sich die Methoden zunutze, indem es die Aufgaben an das Rechteck `r` *delegiert*. Der Nachteil dieser Lösung besteht darin, dass `Quadrat` und `Rechteck` nicht weiterhin polymorph benutzbar sind. Wenn man `Quadrat` von der Klasse `GraphObj` erben ließe, hätte man das Problem, dass der Bezugspunkt doppelt angelegt wäre: im anonymen Subobjekt und im privaten `Rechteck`-Objekt. Falls `Quadrat` nur wenige Funktionen von `Rechteck` benutzt, der Aspekt der Wiederverwendung von Code also keine große Rolle spielt, ist es besser, `Quadrat` als eigenständige Klasse zu implementieren, die von `GraphObj` erbt. Problemstellungen dieser Art kommen gelegentlich vor. Ein weiteres Beispiel: Eine sortierte Liste ist doch sicherlich auch eine Liste – oder? Bei näherer Betrachtung stellt man fest, dass die sortierte Reihenfolge zerstört werden kann. Die Operation, ein beliebiges Element am Anfang einer Liste einzufügen, darf nicht für eine sortierte Liste gelten.

Die Ursache für das Dilemma liegt im Verständnis des Begriffs Spezialisierung bzw. der *ist-ein*-Relation. Mit der `public`-Vererbung ist stets eine Spezialisierung der Schnittstellen oder eine Erweiterung gemeint, in der Mathematik oder in der Umgangssprache kann es aber auch eine *Einschränkung* oder *Verminderung* der Schnittstellen bedeuten.

Nur wenn ein Objekt einer abgeleiteten Klasse jederzeit an die Stelle eines Basisklassenobjekts treten kann, ist die `public`-Vererbung sinnvoll, und nur dann kann der Typ der abgeleiteten Klasse als Subtyp der Basisklasse aufgefasst werden. Andernfalls ist die umgangssprachlich in der Modellierung benutzte *ist-ein*-Beziehung auf andere Art darzustellen. Damit kann `Quadrat` zwar von der oben beschriebenen Klasse `Rechteck` erben. Dies würde jedoch nicht mehr gelten, wenn die Klasse `Rechteck` eine weitere Methode `seitenverhaeltnisAendern()` hätte, weil sie vom Quadrat nicht ohne Verletzung des Vertrags realisiert werden kann. Die erwähnte sortierte Liste sollte nicht `public` von einer Listenklasse erben.

Übungen

7.2 Schreiben Sie eine Klasse `Person` mit den zwei Attributen `Nachname` und `Vorname`, sowie eine Klasse `StudentIn` und eine Klasse `ProfessorIn`, die beide von `Person` erben. Die Klasse `StudentIn` soll ein Attribut »Matrikelnummer«, die Klasse `ProfessorIn` ein Attribut »Lehrgebiet« haben. Der Einfachheit halber seien alle Attribute vom Typ `string`. Fügen Sie Methoden zum Lesen der Attribute hinzu, zum Beispiel `const string& getNachname()` bei der Klasse `Person`. Es soll auch eine Methode `toString()` geben, die die vollständigen Informationen liefert und deren Schnittstelle und eine Standardimplementierung in der Klasse `Person` definiert ist. Die Standardimplementierung soll einen aus Vor- und Nachnamen zusammengesetzten String zurückliefern, die in den Unterklassen zu redefinierenden Implementierungen auch den Status (StudentIn/ProfessorIn) und die Matrikelnummer bzw. das Lehrgebiet enthält. Von der Klasse `Person` soll kein Objekt erzeugt werden können, sie sei also abstrakt. Der folgende Programmauszug zeigt die Benutzung der Klassen:

```
vector<Person*> diePersonen;
diePersonen.push_back(
    new StudentIn("Risse", "Felicitas", "635374"));
diePersonen.push_back(
    new ProfessorIn("Philippsen", "Nele", "Datenbanken"));
diePersonen.push_back(
    new StudentIn("Spillner", "Julian", "123429"));
for(size_t i = 0; i < diePersonen.size(); ++i) {
   cout << diePersonen[i]->getVorname() << endl;
}
for(size_t i = 0; i < diePersonen.size(); ++i) {
   cout << diePersonen[i]->toString() << endl;
}
```

Die Ausgabe des Programms sei z.B.:

Felicitas
Nele
Julian
Student/in Felicitas Risse, Mat.Nr.: 635374
Prof. Nele Philippsen, Lehrgebiet: Datenbanken
Student/in Julian Spillner, Mat.Nr.: 123429

7.3 Wie kann man im obigen Programmauszug auf eine Methode der Klasse `StudentIn` zugreifen, zum Beispiel auf die Methode `getMatrikelnummer()`?

7.8 Mehrfachvererbung[3]

Die Mehrfachvererbung gewährt eine große Flexibilität insbesondere bei der Systemmodellierung, wird jedoch nicht häufig benötigt – je nach Art der Problemstellung. Die Mehrfachvererbung bietet gegenüber der Einfachvererbung bessere Möglichkeiten, Objekte der realen Welt abzubilden.

Eine Klasse kann von *mehreren* Basisklassen erben, wie in Abbildung 7.1 auf Seite 258 zu sehen ist. Da hier *nur das Prinzip* der Mehrfachvererbung gezeigt werden soll, betrachten wir im Folgenden ein möglichst einfaches Beispiel, das als C++-Programm ausformuliert wird. Auf einem Graphikbildschirm sollen verschiedene Objekte dargestellt werden, hier ein Rechteck (`Rechteck`) und ein beschriftetes Rechteck (`beschriftetesRechteck`).

Ein beschriftetes Rechteck *ist ein* beschriftetes grafisches Objekt, und ein beschriftetes grafisches Objekt wiederum *ist ein* grafisches Objekt. Dieser Zusammenhang wird durch die in Abbildung 7.4 dargestellte Vererbungsstruktur gezeigt. Die Klasse `beschriftetesObjekt` ist wie `GraphObj` abstrakt, weil die in der letzteren Klasse deklarierte rein virtuelle Funktion `flaeche()` nicht in `beschriftetesObjekt` definiert ist und daher die Eigenschaft »abstrakt« geerbt wird.

[3] Dieser Abschnitt kann beim ersten Lesen übersprungen werden.

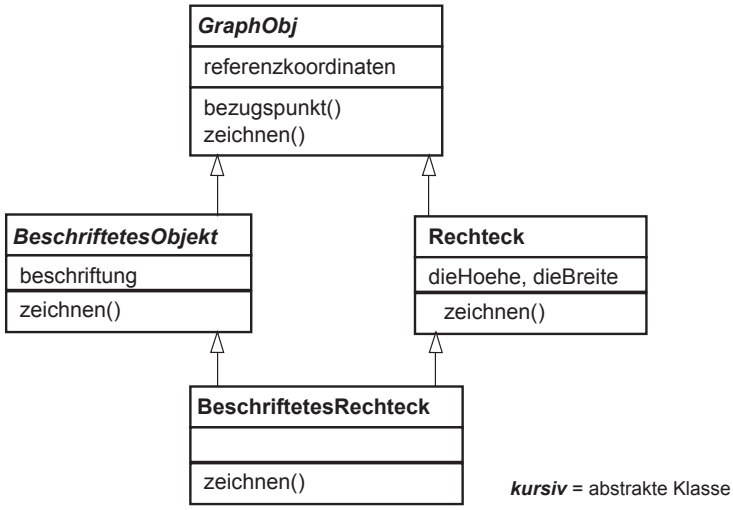

Abbildung 7.4: Vererbungsstruktur grafischer Objekte

Es ist im Allgemeinen nicht notwendig, dass von einer gemeinsamen Basisklasse geerbt wird. Hier wurde das Beispiel absichtlich so gewählt, weil mit einer gemeinsamen Basisklasse eine spezielle Problematik auftritt, die in Abschnitt 7.8.1 besprochen wird.

Alle grafischen Objekte haben bestimmte gemeinsame Eigenschaften. Zum Beispiel hat jedes Objekt einen bestimmten Ort auf dem Bildschirm, nämlich den Bezugspunkt `referenzkoordinaten`. Es folgen die Header-Dateien **.h* mit den Deklarationen für `BeschriftetesObjekt` und `BeschriftetesRechteck`. Die anderen Deklarationen sind in Abschnitt 7.6.2 ab Seite 276 zu finden.

Listing 7.11: Klasse BeschriftetesObjekt

```
// cppbuch/k7/mehrfach/konflikt/beschriftetesobjekt.h
#ifndef BESCHRIF_H
#define BESCHRIF_H
#include"graphobj.h"
#include<string>

class BeschriftetesObjekt : public GraphObj { // erben
  public:
    BeschriftetesObjekt(const Ort& ort, const std::string& b)
    : GraphObj(ort), beschriftung(b) {
    }

    virtual void zeichnen() const {
        GraphObj::zeichnen();
        std::cout << "Beschriftung bei ";
        anzeigen(bezugspunkt());
        std::cout << beschriftung << std::endl;
    }
  private:
    std::string beschriftung;
```

```
};
#endif // BESCHRIF_H
```

Die Klasse `BeschriftetesObjekt` enthält ein Objekt `beschriftung` des Typs `string`. Der Einfachheit halber sind alle Methoden `inline`. Die Klasse `BeschriftetesObjekt` benötigt keinen Destruktor, weil der systemerzeugte Destruktor die Destruktoren für alle Elemente einer Klasse aufruft, sodass zum Beispiel der Destruktor von `string` den dynamisch bereitgestellten Platz für die Beschriftung freigibt.

Die zur tatsächlichen Ausgabe auf dem Bildschirm notwendigen Graphikfunktionen sind systemspezifisch, sodass hier nur eine schlichte Textausgabe auf dem Bildschirm erscheinen soll. Der Konstruktor ruft jeweils den Basisklassenkonstruktor zur Initialisierung auf.

Auch ein `BeschriftetesRechteck` wird mit den Oberklassenkonstruktoren initialisiert, die ihrerseits den Basisklassenkonstruktor aufrufen. Die Funktion `zeichnen()` ruft die entsprechenden Methoden der Subobjekte auf.

Listing 7.12: Klasse BeschriftetesRechteck

```
// cppbuch/k7/mehrfach/konflikt/beschriftetesrechteck.h
#ifndef BES_R_H
#define BES_R_H
#include"beschriftetesobjekt.h"
#include"rechteck.h"

// Mehrfachvererbung
class BeschriftetesRechteck
   : public BeschriftetesObjekt, public Rechteck {
  public:
    BeschriftetesRechteck(const Ort& o, int h, int b,
                          const std::string& beschr)
    : BeschriftetesObjekt(o, beschr),
      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
```

In einem Hauptprogramm könnten nach diesen Definitionen Anweisungen folgender Art stehen:

```
// cppbuch/k7/mehrfach/konflikt/main.cpp
// Auszug:
   Rechteck r(Ort(0,0), 20, 50);
   BeschriftetesRechteck bR(Ort(1,20), 60, 60,
```

```
                        std::string("Mehrfachvererbung"));
  r.zeichnen();
  bR.zeichnen();
  BeschriftetesRechteck *zBR = new BeschriftetesRechteck(
                Ort(100,0), 20, 80,
                std::string("dynamisches Rechteck"));
  zBR->zeichnen();
```

Das Objekt `*zBR` muss mit `delete` gelöscht werden, weil es mit `new` erzeugt wurde. Die Anwendung von `delete` auf einen Zeiger ruft automatisch den Destruktor des referenzierten Objekts auf.

7.8.1 Namenskonflikte

Bei Mehrfachvererbung können Namenskonflikte und Mehrdeutigkeiten auftreten. Zum Beispiel könnte man versuchen, sich die Koordinaten der Objekte ausgeben zu lassen:

```
std::cout << "Rechteck-Position: ";
anzeigen(r.bezugspunkt());
std::cout << "beschriftetes-Rechteck-Position: ";
anzeigen(bR.bezugspunkt());     // Compiler-Fehlermeldung!
```

Vom Rechteck `r` würde der Bezugspunkt ausgegeben werden, die Ausgabe der Koordinaten des beschrifteten Rechtecks `bR` führt hingegen zu einer Fehlermeldung des Compilers. Warum? Der Aufruf ist zweideutig. Die Ursache liegt darin, dass `GraphObj` *zweimal* geerbt wurde. Der Compiler weiß nicht, ob er den Bezug zu `GraphObj::bezugspunkt()` über das in `BeschriftetesObjekt` oder das in `Rechteck` enthaltene Subobjekt vom Basisklassentyp `GraphObj` konstruieren soll. Durch die Angabe der Basisklasse wird die Zweideutigkeit beseitigt:

```
anzeigen(bR.Rechteck::bezugspunkt()); // eindeutig
```

Ferner wird durch verschiedene Bezugspunkte im Konstruktor nachgewiesen, dass `BeschriftetesRechteck` *zwei* `GraphObj`-Objekte besitzt:

```
    // absichtlich veränderter Konstruktor
    BeschriftetesRechteck(const Ort& ort, int h, int b,
                          const std::string& b)
    : BeschriftetesObjekt(ort, b),
      Rechteck(Ort(100, 100), h, b) { // verschiedene Koordinaten!
    }
// jetzt verschiedene Werte:
anzeigen(bR.Rechteck::bezugspunkt());
anzeigen(bR.BeschriftetesObjekt::bezugspunkt());
```

Weil zwei Subobjekte vom Typ `GraphObj` vorliegen, ist wegen der Nicht-Eindeutigkeit die Zuweisung eines Zeigers nicht möglich:

```
int main() {
    Rechteck r1(Ort(0,0), 20, 50);
    Rechteck r2(Ort(0,100), 10, 40);
    BeschriftetesRechteck bR2(Ort(1,20), 60, 60,
                std::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 im Feld
    int i = 0;
    while(graphObjZeiger[i]) {
        graphObjZeiger[i++]->zeichnen();
    }
}
```

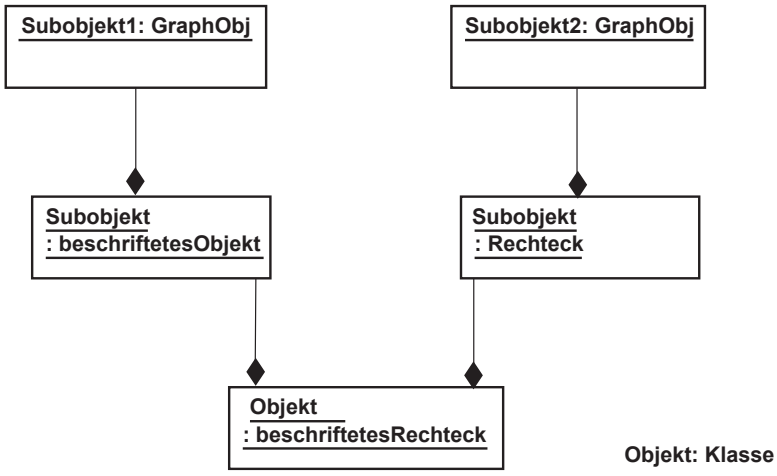

Abbildung 7.5: Zweideutig: *enthält*-Beziehungen bei nicht-virtueller Vererbung

Auf welches Subobjekt soll der Zeiger `graphObjZeiger[2]` verweisen, wenn die `//`-Markierung in der Zeile unter dem »Fehler«-Hinweis entfernt würde? Die tatsächliche Objekthierarchie für ein `BeschriftetesRechteck`-Objekt `bR` ergibt sich aus der Abbildung 7.5, wobei die Pfeile hier eine *enthält*-Beziehung symbolisieren, das heißt, das `beschriftetesRechteck bR` *enthält* ein `Rechteck`- und ein `beschriftetesObjekt`-Subobjekt, die beide *je ein* `GraphObj`-Subobjekt *enthalten*, deren Koordinaten nicht notwendigerweise gleich sein müssen. Im nächsten Abschnitt wird gezeigt, wie die Zweideutigkeiten aufgelöst werden.

7.8.2 Virtuelle Basisklassen

Wenn bei Mehrfachvererbung nicht erwünscht ist, dass mehrere Basisklassensubobjekte erzeugt werden, können *virtuelle Basisklassen* verwendet werden. Von diesen Basisklassen wird nur *ein* Subobjekt erzeugt, auf das über verschiedene Vererbungswege zugegriffen werden kann. Die Mehrdeutigkeit im obigen Beispiel wäre dadurch aufgehoben. Im Folgenden werden *nur* die Deklarationen und die Methoden aus dem vorherigen Abschnitt ganz oder teilweise aufgelistet, die notwendige Änderungen enthalten.

```
// Datei rechteck.h:
class Rechteck : virtual public GraphObj {
    // ... Rest wie vorher
};
```

```
// Datei beschriftetesobjekt.h:
class BeschriftetesObjekt : virtual public GraphObj {
   // ... Rest wie vorher
};
```

```
// Datei beschriftetesrechteck.h:
class BeschriftetesRechteck
   : public BeschriftetesObjekt, public Rechteck {
     // geänderter Konstruktor
     BeschriftetesRechteck(const Ort& ort, int h, int b,
                           const std::string& b)
     : GraphObj(ort),
       BeschriftetesObjekt(ort, b), // Diese Initialisierungen mit ort
       Rechteck(ort, h, b) { // werden ignoriert, siehe Abschnitt 7.8.2
     }
     // ... Rest wie vorher
};
```

Mit diesen Änderungen sind Aufrufe wie

```
cout << "BeschriftetesRechteck–Position: ";
anzeigen(bR.bezugspunkt());
```

möglich und unproblematisch, weil nun genau *ein* Basisklassensubobjekt für `bR` existiert. Der Konstruktor der Klasse `BeschriftetesRechteck` initialisiert jetzt das Basisklassensubobjekt; die Erklärung dafür finden Sie im folgenden Unterabschnitt »Virtuelle Basisklassen und Initialisierung«.

Weil nun genau ein Basisklassensubobjekt pro vollständigem Objekt existiert, kann ein Basisklassenzeiger auf ein Objekt der abgeleiteten Klasse gerichtet und damit der Polymorphismus ausgenutzt werden. Unter einem »vollständigen Objekt« wird ein Objekt verstanden, das nicht als Subobjekt dient, also nicht in einem anderen Objekt durch Vererbung enthalten ist. Im folgenden Beispiel sind `R1`, `R2` und `bR` vollständige Objekte, nicht aber die in ihnen enthaltenen Subobjekte.

```
int main() {                                          // geändert
    Rechteck R1(Ort(0,0), 20, 50);
    Rechteck R2(Ort(0,100), 10, 40);
    BeschriftetesRechteck bR(Ort(1,20), 60, 60,
                 std::string("virtuelle Mehrfachvererbung"));
    // Feld mit Basisklassenzeigern, initialisiert mit
    // den Adressen der Objekte, 0 als Endekennung
    GraphObj* graphObjZeiger[] = {&R1, &R2, &bR, 0}; // jetzt ok!
    // Zeichnen aller Objekte
    int i = 0;
    while(graphObjZeiger[i])
          graphObjZeiger[i++]->zeichnen();
}
```

Virtuelle Basisklassen und Initialisierung

Im Abschnitt 7.1 (Seite 263) wird die Initialisierung von Subobjekten behandelt. Dabei werden Initialisierer in einer Liste angegeben, die noch vor dem Codeblock des Konstruktors abgearbeitet wird. In einer Klassenhierarchie kann es mehrere Initialisierer für eine Basisklasse geben. Falls wir jedoch virtuelle Basisklassen haben, wird *nur ein* Subobjekt dieser Basisklasse in Objekten einer abgeleiteten Klasse angelegt. Dann darf natürlich nur *ein* Initialisierer wirksam werden, damit es keine widersprüchlichen Ergebnisse gibt, wenn einer »Links!«und der andere »Rechts!« sagt. Um dieses Problem zu lösen, wird in C++ der Basisklasseninitialisierer genommen, *der bei dem Konstruktor eines vollständigen Objekts angegeben ist*, also einem Objekt, das bei der Definition in der Vererbungshierarchie ganz unten steht und das daher nicht als Subobjekt innerhalb eines anderen Objekts dient. Die anderen Basisklasseninitialisierer werden *ignoriert*. Wenn im Konstruktor *kein* Basisklasseninitialisierer aufgeführt ist, wird der Standardkonstruktor der virtuellen Basisklasse genommen. Das Programm zeigt die Initialisierung von Subobjekten virtueller Basisklassen. Es gibt zweimal *Basis-Standardkonstruktor* aus. Der Basisklasseninitialisierer `Basis(a)` in der Klasse `Rechts` wird beim Konstruktor von `Unten` ignoriert.

Listing 7.13: Initialisierung bei virtueller Basisklasse

```
// cppbuch/k7/mehrfach/basinit.cpp
#include<iostream>
class Basis {
    public:
       Basis() { std::cout << "Basis-Standardkonstruktor\n"; }
       Basis(const char* a) { std::cout << a << std::endl; }
       virtual ~Basis(){} // virtueller Destruktor
};

class Links : virtual public Basis {
    public:
        Links(const char* a)
        // : Basis(a) // siehe Text unten
        { }
};

class Rechts : virtual public Basis {
   public:
        Rechts(const char* a) : Basis(a) {}
};

class Unten: public Links, public Rechts {
    public:
        Unten(const char* a) :
        // Basis(a), // siehe Text unten
        Links(a), Rechts(a) {}
};

int main() {
   Unten un("Unten");
   Links li("Links");
}
```

Stattdessen wird nur der beim Konstruktor von `Unten` direkt angegebene Basisklassenkonstruktor berücksichtigt. Da er hier auskommentiert ist, wird der Standardkonstruktor von Basis genommen. Wenn jedoch die Kommentarzeichen // aus den Initialisierungslisten entfernt werden, ist die Ausgabe

Unten
Links.

`Rechts::Basis(a)` wird weiterhin ignoriert. Die Regel ist: Der Konstruktor eines vollständigen Objekts ist für die Initialisierung des Basisklassensubobjekts bei virtueller Vererbung verantwortlich.

7.9 Standard-Typumwandlungsoperatoren

Meistens sind zunächst scheinbar notwendige Typumwandlungen nur ein Zeichen für schlechtes Design und sollten daher zum Nachdenken anregen. Die Typumwandlung (*cast*) im C-Stil umgeht die Typkontrolle durch den Compiler und ist deshalb gefährlich. Die syntaktische Notation nur durch Klammern kann leicht übersehen werden und ist auch mit Werkzeugen oder automatisierter Suche mit dem Editor schwierig, wenn man alle Casts verschiedener Datentypen finden will.

Andererseits sind Typumwandlungen für spezielle Zwecke notwendig, wie unter anderem in Abschnitt 7.10 gezeigt wird. Um die Nachteile der Casts im C-Stil zu umgehen, wurden neue Typumwandlungsoperatoren entworfen, die die vorherigen Casts überflüssig machen. Sie haben einige Vorteile:

- Sie sind durch ihre Namen optisch und syntaktisch leicht zu erkennen.
- Sie sind spezialisiert, sodass nur der erwünschte Effekt eintritt – also nicht mehr ein Cast für alles.

Die Syntax ist bis auf den Operatornamen für alle Typumwandlungs-Operatoren gleich:

Operatorname<*T*>(*Ausdruck*)

Das Ergebnis des Ausdrucks soll in den Typ *T* gewandelt werden.

Der static_cast-Operator

Der `static_cast`-Operator ist dazu gedacht, implizit erlaubte Standard-Typumwandlungen durchzuführen oder rückgängig zu machen (vergleiche Beispiel auf Seite 80):

```
enum Wochentag {sonntag, montag, dienstag, mittwoch,
                donnerstag, freitag, samstag
               } heute = dienstag;

int i = dienstag;       // implizite Umwandlung nach int
heute = i;              // Fehler, Datentyp inkompatibel
heute = static_cast<Wochentag>(i); // erlaubt!
```

Falls die Variable `i` einen Wert hat, der nicht einem der Werte des Aufzählungstyps entspricht, ist der Wert der Variablen `heute` undefiniert.

Die implizite Typumwandlung in einer Klassenhierarchie, die auf Seite 266 beschrieben wird, lässt sich ebenfalls invertieren, sodass zum Beispiel Wandlungen wie `Basis*` zu `Abgeleitet*` vorgenommen werden können:

```
GraphObj g(Ort(3, 17));
Strecke s(Ort(3, 17), (Ort(0, 0)); // Strecke ist von GraphObj abgeleitet
GraphObj *pg;
Strecke *ps = &s;
pg = ps;                          // bekannte implizite Konversion
ps = pg;                          // verboten!
ps = (Strecke*) pg;               // gefährlicher C-Stil!
ps = static_cast<Strecke*> (pg); // richtig (falls pg auf ein Strecke-Objekt zeigt)
```

Der `static_cast`-Operator ist nur dann geeignet, wenn zur Compilierzeit bereits feststeht, dass der Basisklassenzeiger (`pg`) auf ein Objekt einer abgeleiteten Klasse zeigt. Anstelle von Zeigern sind Referenzen möglich. Die Typumwandlung von einer Basisklasse zur abgeleiteten Klasse wird *downcast* genannt und ist nicht erlaubt, wenn die Basisklasse virtuell ist. Die `const`-Eigenschaft von Objekten kann nicht mit dem `static_cast` eliminiert werden.

Der dynamic_cast-Operator

Der Operator `dynamic_cast<T>(Ausdruck)` wirkt ähnlich wie der `static_cast`-Operator, jedoch mit folgenden Unterschieden:

- Die Typprüfung findet *zur Laufzeit* statt, falls das Ergebnis nicht schon zur Compilierzeit bestimmt werden kann. Dann verhält sich `dynamic_cast` wie ein `static_cast`. Weitere Möglichkeiten zur Typprüfung zur Laufzeit siehe Abschnitt 7.10.
- Typ *T* muss ein Zeiger oder eine Referenz auf eine Klasse sein.
- Falls das Argument *Ausdruck* ein Zeiger ist, der nicht auf ein Objekt vom Typ *T* (oder abgeleitet von *T*) zeigt, wird als Ergebnis der Typumwandlung ein Null-Zeiger auf den Ergebnistyp, d.h. `(T*)0` zurückgegeben.
- Falls das Argument *Ausdruck* eine Referenz ist, die nicht auf ein Objekt vom Typ *T* (oder abgeleitet von *T*) verweist, wird eine Ausnahme (Exception) vom Typ `bad_cast` ausgeworfen.

Die wesentlichen Varianten sind im Beispiel dargestellt:

```
class Basis {
  public:
    virtual void f() {}
};

class Abgeleitet : public Basis {
  public:
    virtual void f() {}
};

Abgeleitet* g(Basis *pB) {     // g() benutzt f()
   Abgeleitet *pA = dynamic_cast<Abgeleitet*>(pB);
```

```
    if(pA)       // NULL bei Scheitern des dynamic_cast
      pA->f();   // Abgeleitet::f()
    return static_cast<Abgeleitet*>(pB);
  }

int main() {
    Basis einB;
    Abgeleitet einA;
    Basis *pBB = & einB;
    Basis *pBA = & einA;
    Abgeleitet *pErgebnis;
    // Durch den folgenden Aufruf von g() wird Abgeleitet::f() ausgeführt,
    // weil pBA auf ein Abgeleitet-Objekt zeigt. pErgebnis zeigt auf einA:
    pErgebnis = g(pBA);

    // Abgeleitet::f() wird unten nicht ausgeführt, weil pB in g() auf ein Basis
    // -Objekt zeigt. pErgebnis ist undefiniert(!), weil der static_cast ungeeignet
    // ist: Der dynamische Typ des per Zeiger übergebenen Objekts ist nicht vom Typ
    // Abgeleitet.
    pErgebnis = g(pBB);
} // Ende von main()
```

Übung

7.4 Lösen Sie die Aufgabe 7.3 auf Seite 285 mit dem `dynamic_cast<>()`-Operator. Geben Sie die Matrikelnummern aller Personen aus, sofern diese eine haben.

Der const_cast-Operator

Dieser Operator ist der einzige, der die `const`-Eigenschaft eines Objekts beseitigen kann. Dementsprechend sollte er möglichst nicht eingesetzt werden, zumal ein verändernder Zugriff auf ein konstantes Objekt über `const_cast` zu einem unerwarteten Verhalten eines Programms führen kann.

```
const int i = 100;
const int *ip = &i;
*ip = 0;                          // geht nicht
int *iq = const_cast<int*>(&i);  // explizite Typumwandlung
*iq = 0;                          // Wert von i wird geändert!
```

Der Operator `const_cast<T>(Obj)` ändert die `const`-Eigenschaft des Objektes `Obj`. Der Datentyp von `Obj` muss `const T` (oder `T`) sein, wobei `T` auch ein Zeiger oder eine Referenz sein kann. Der `const_cast`-Operator ersetzt die früher übliche Form `(T) Obj`, die eine erzwungene Typumwandlung eines *beliebigen* Datentyps nach `T` bewirkt. Die Typumwandlung sollte nur in begründeten Ausnahmefällen vorgenommen werden; schließlich hat eine `const`-Deklaration ihren Sinn. Der Operator kann auch benutzt werden, um einen nichtkonstanten Typ als konstant erscheinen zu lassen. Beispiel für einen Typ `X`:

```
X einX;
X const& cr = const_cast<X const&>(einX);
```

Über `cr` können für das Objekt `einX` nur nicht verändernde (d.h. `const`-qualifizierte) Methoden aufgerufen werden.

Der reinterpret_cast-Operator

Dieser Operator kann die `const`-Eigenschaft eines Objekts nicht ändern, aber ansonsten ist jede Typumwandlung möglich. Die Typumwandlung findet zur Compilierzeit statt. Ein Objekt wird im Sinne des gewünschten Datentyps »re-interpretiert«. Weil ganz verschiedene Datentypen ineinander gewandelt werden können, ist das Ergebnis meistens implementationsabhängig. Dieser Operator sollte nur in den ganz seltenen Fällen benutzt werden, in denen die Anwendung der vorher beschriebenen Typumwandlungsoperatoren nicht möglich ist, zum Beispiel wenn es nur um die reinen Bits geht wie bei der binären Ein-/Ausgabe in Abschnitt 5.8.

7.10 Typinformationen zur Laufzeit

Der oben beschriebene `dynamic_cast`-Operator wandelt den Typ eines Objekts zur Laufzeit und führt dabei gleichzeitig eine Prüfung durch. In den meisten Fällen ist dies ausreichend, manchmal möchte man aber mehr wissen. Die Laufzeit-Typinformation kann für alle Methoden benutzt werden, die als Argument den Klassentyp selbst (d.h. auch Zeiger und Referenzen auf die Basisklasse) haben und polymorph benutzt werden sollen.

Typidentifizierung mit typeid()

Das Ergebnis eines `typeid()`-Ausdrucks ist vom vordefinierten Typ `type_info&`. Wenn das Argument von `typeid()` ein polymorpher Typ ist, bezieht sich das Ergebnis von `typeid()` auf das zugehörige vollständige Objekt. Mit »polymorpher Typ« ist gemeint, dass das Argument eine Referenz vom Basisklassentyp ist, die auf ein Objekt einer abgeleiteten Klasse verweist. Die Dereferenzierung eines Zeigers durch ein vorangestelltes '*' liefert ebenfalls eine Referenz:

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

class Basis { ... };
class Abgeleitet: public Basis { ... };

int main() {
  Basis einBasisObjekt;
  Abgeleitet Objekt1, Objekt2;
  Basis *p = &Objekt1;
  Basis *pNull = 0;
  if(typeid(Objekt2) == typeid(*p)) { // *p ist polymorph
     cout << "true";
  }
```

```
  else {
     cout << "false";
  }
  if(typeid(Objekt1) == typeid(einBasisObjekt)) {
    cout << "true";
  }
  else {
    cout << "false";
  }
  if(typeid(Objekt1) == typeid(*pNull))
  // ...
```

Im Programm wird nacheinander *true* und *false* ausgegeben, bevor in der letzten `if`-Anweisung eine `bad_typeid`-Ausnahme ausgeworfen wird, weil `pNull` ein Null-Zeiger ist. Der Vergleichsoperator vergleicht die von `typeid()` zurückgegebenen `type_info`-Objekte. Anstelle eines Objekts kann der Klassenname verwendet werden, die beiden Abfragen

```
if(typeid(Objekt2) == typeid(*p)) { ... }
if(typeid(Abgeleitet) == typeid(*p)) { ... }
```

haben dieselbe Wirkung. Der Typ eines Objekts (Klassenname) kann als compilerabhängiger Wert vom Typ `const char*` erhalten werden:

```
cout << typeid(Objekt1).name(); // Ausgabe: Abgeleitet
```

Auf Seite 371 finden Sie ein Beispiel für die Anwendung von `typeid`.

Übung

7.5 Lösen Sie die Aufgabe 7.3 auf Seite 285 mit dem `typeid()`-Operator. Geben Sie die Matrikelnummern aller Personen aus, sofern diese eine haben.

7.11 Using-Deklaration für Klassen

Ein Namespace ist ein Sichtbarkeitsbereich (englisch *scope*) ähnlich wie der einer Klasse, der ebenfalls einen Namen hat. Die für Namespaces verwendete Using-Deklaration wird in ähnlicher Form für Klassenmethoden verwendet, um in einer abgeleiteten Klasse den gezielten Zugriff auf eine Basisklassenmethode zu ermöglichen:

```
class Basis {
   protected:
     void f(int);
};

class Abgeleitet : public Basis {
   public:
     using Basis::f;
     // ...
};
```

Mit dieser Using-Deklaration ist `Abgeleitet::f()` ein *öffentliches* Synonym für `Basis::f()`:

```
Basis einBasisObjekt;
einBasisObjekt.f(0); // Fehler! f() ist nicht public.

Abgeleitet einAbgeleitetObjekt;
einAbgeleitetObjekt.f(0); // ok
```

Private Methoden der Klasse `Basis` können auf diese Art nicht öffentlich gemacht werden.

7.12 Private- und Protected-Vererbung[4]

Delegation ist *eine* Möglichkeit zur Wiederverwendung von Code, private Vererbung, auch Implementationsvererbung genannt, ist eine andere. Die Delegation ist vorzuziehen, um mit der Vererbung ausschließlich eine *ist-ein*-Beziehung zwischen Klassen abzubilden (Vererbung der *Schnittstellen*). Aber Sie sollen wenigstens wissen, was private Vererbung bedeutet, wenn auf der nächsten Party die Rede davon ist, um elegant zu einem für eine Party interessanteren Thema wechseln zu können. Die private Vererbung wird hier am Beispiel einer Warteschlange oder Queue gezeigt (ansonsten sollten Sie die Klasse `std::queue` der Standardbibliothek benutzen). Dabei machen wir uns die Eigenschaften der Klasse `std::list`, einer doppelt-verketteten Liste (siehe auch Seite 772), zunutze. Eine einfache Anwendung könnte wie folgt aussehen:

Listing 7.14: Art der Vererbung ist nicht sichtbar

```
// cppbuch/k7/privat/main.cpp
#include<string>
#include<iostream>
#include"warteschlange.t"
using namespace std;

int main() {
  Warteschlange<string> fifo;
  fifo.push( string("eins"));
  fifo.push( string("zwei"));
  fifo.push( string("drei"));

  while(!fifo.empty()) {
    cout << fifo.size() << " Element(e) vorhanden!\n";
    string buf = fifo.front(); // lesen
    fifo.pop();                // löschen
    cout << "Element " << buf << " entnommen\n";
  }
  cout << "Liste ist leer!" << endl;
}
```

[4] Dieser Abschnitt kann beim ersten Lesen übersprungen werden.

Bei privater Vererbung dürfen öffentliche Methoden der Oberklasse zwar innerhalb der Unterklasse benutzt werden, nicht aber von Objekten der Unterklasse (vgl. Tabelle 7.1 auf Seite 265). Es wird nicht mehr die Schnittstelle geerbt, sondern die Implementierung. Sollen einzelne Methoden für Objekte abgeleiteter Klassen nutzbar sein, also für Objekte der Klasse `Warteschlange`, sind sie durch eine Benutzungsdeklaration (englisch *using declaration*) zu kennzeichnen, wie in Abschnitt 7.11 beschrieben. Die Benutzungsdeklaration besteht nur aus dem Schlüsselwort `using` und dem Namen der Funktion einschließlich der Klassenbezeichnung, aber ohne Parameterliste und Rückgabetyp. Nun ist erreicht, dass wirklich nur die gewünschten Methoden aufgerufen werden können. Falls von `Warteschlange` selbst eine weitere Klasse `public` abgeleitet würde, könnte sie nur die in der öffentlichen Schnittstelle von `Warteschlange` deklarierten Methoden benutzen. Auf diese Art wird der von der `std::list`-Klasse vererbte Methodenumfang der Oberklasse ausgewählt.

Listing 7.15: Klasse mit privater Vererbung

```
// cppbuch/k7/privat/warteschlange.t Warteschlangen-Template
#ifndef WARTESCHLANGE_T
#define WARTESCHLANGE_T
#include<list>

template<typename T>
class Warteschlange
  : private std::list<T> { // mit privater Vererbung (Implementationsvererbung)
    public:
      using std::list<T>::empty;
      using std::list<T>::size;
      // am Ende einfügen:
      void push(const T& x) {
         std::list<T>::push_back(x);
      }
      // am Anfang entnehmen:
      void pop() {
         std::list<T>::pop_front();
      }
      // am Anfang bzw. Ende lesen
      using std::list<T>::front;
      using std::list<T>::back;
};
#endif
```

Ein privater Teil ist überflüssig, das verborgene Basisklassensubobjekt vom Typ `std::list` erledigt alles. Die Methoden `push()` und `pop()` existieren nicht unter diesem Namen in der Oberklasse und können deshalb nicht per `using`-Deklaration öffentlich gemacht werden. Konstruktor, Destruktor und Zuweisungsoperator sind nicht notwendig. Zum Vergleich sei hier ein Template gezeigt, in dem die Delegation anstelle der privaten Vererbung tritt:

```
template<typename T>
class Warteschlange { // mit Delegation an ein list-Objekt (Attribut liste)
 public:
   bool empty( ) {
      return liste.empty();
```

```
  }
  size_t size() {
    return liste.size();
  }
  // am Ende einfügen:
  void push(const T& x) {
    liste.push_back(x);
  }
  // am Anfang entnehmen:
  void pop() {
    liste.pop_front();
  }
  // am Anfang bzw. Ende lesen
  const T& front() {
    return liste.front();
  }
  const T& back() {
    return liste.back();
  }
 private:
  std::list<T> liste;
};
```

Das Prinzip ist einfach: Ein Listenobjekt `liste` wird privat angelegt, und die Elementfunktionen der Klasse `Warteschlange` rufen die öffentlichen Elementfunktionen des Objekts `liste` auf. Die Klasse `Warteschlange` delegiert damit Aufgaben an die Klasse `std::list`, weswegen das Prinzip *Delegation* genannt wird. Bisher sind wir davon ausgegangen, dass man für dynamische Datenstrukturen einen besonderen Kopierkonstruktor benötigt, wie am Beispiel der »flachen« und »tiefen« Kopie auf Seite 236 gezeigt. Wenn ein besonderer Konstruktor notwendig ist, gilt dies meistens auch für einen Destruktor und einen Zuweisungsoperator. Das alles können wir hier vergessen! Durch die Delegation enthält jedes `Warteschlange`-Objekt ein Objekt vom Typ `std::list`, und nur dieses enthält eine dynamische Struktur. Weil bei der Kopie oder Zuweisung ein Objekt *elementweise* kopiert wird, wird also das einzige Element der Klasse `Warteschlange` kopiert (das private Objekt `liste`), indem der Kopierkonstruktor bzw. Zuweisungsoperator für dieses Objekt aufgerufen wird. Die Klasse `std::list` stellt alle Dienstleistungen bereit, sodass sie nicht besonders programmiert werden müssen, und die Klasse `Warteschlange` wird dadurch zu einem »Datentyp erster Klasse«, der genauso einfach wie die Grunddatentypen zu handhaben ist. Der Zuweisungsoperator jedes Elements (hier nur `liste`) wird bei der Zuweisung eines `Warteschlange`-Objekts aufgerufen.

protected-Vererbung

Die `protected`-Vererbung spielt nur selten eine Rolle. Ein bruchstückhaftes Beispiel zeigt die Syntax:

```
class Basis {
  public:
    void f();
  // ...
};
```

```
class Abgeleitet : protected Basis {
  public:
   void g();
  // ....
};
```

Die Wirkung ist, dass alle `public`-Elemente der Klasse `Basis` nunmehr `protected`-Elemente der Klasse `Abgeleitet` werden. Damit sind sie innerhalb der Klasse `Abgeleitet` und aller von ihr abgeleiteten Klassen benutzbar, aber nicht von außerhalb. Beispiel:

```
void Abgeleitet::g() { // Implementierung
  f();                 // ok, f() ist zugreifbar
}
```

```
// Benutzung im main-Programm:
Basis einBasisObjekt;
einBasisObjekt.f();     // ok, public
Abgeleitet einAbgeleitetObjekt;
einAbgeleitetObjekt.g(); // ok, public
einAbgeleitetObjekt.f(); // Fehler, nicht zugreifbar
```