8.3 Vererbung/Überschreiben von Methoden

8.3.1 Problem

Der Umgang mit Vererbung kann kompliziert werden: Meistens haben die abgeleiteten Klassen irgendwelche Besonderheiten in ihren Attributen. Man braucht dann auch extra Methoden, um sie zu bearbeiten.

So muss `punkt p` mit `p.display()` ausgegeben werden, `farbpunkt f` dagegen mit `f.farbe_display()`. Wenn man aus Versehen `f.display()` aufruft, erscheint der farbige Punkt in schwarz-weiß. Ein anderes Beispiel sieht man hier:

dackel0.cpp

```
#include <iostream>
using namespace std;
//==================
class hund
{
public:
    void printit(void)
    {
        cout << "Dies ist ein Hund." << endl;
    }
};
//==================
class dackel: public hund
{
public:
    void dprintit(void)
    {
        cout << "Dies ist ein Dackel." << endl;
    }
};
//==================
int main(void)
{
    hund   lassie1;
    dackel lassie2;

    cout << "lassie1 (Hund): " <<endl;
    lassie1.printit();
    cout << "lassie2 (Dackel): " << endl;
    lassie2.dprintit();
    return 0;
}
```

Kann man das in C++ nicht einfacher haben?

8.3.2 Statische Polymorphie (=Überschreiben von Methoden)

C++ bietet dafür bei Vererbung *statische Polymorphie* an. Das bedeutet: Miteinander verwandte Klassen dürfen Methoden haben, die gleich heißen *und auch noch die gleichen Parameter haben*! Der Compiler verwendet dann die Methode, die zur aktuellen Klasse (in diesem Fall: der Unterklasse) passt. Hier das Beispiel:

dackel1.cpp

```
#include <iostream>
using namespace std;
//====================
class hund
{
public:
   void printit(void)
   {
      cout << "Dies ist ein Hund." << endl;
   }
};
//====================
class dackel: public hund
{
public:
   void printit(void)
   {
      cout << "Dies ist ein Dackel." << endl;
   }
};
//====================
int main(void)
{
   hund   lassie1;
   dackel lassie2;

   cout << "lassie1 (Hund): " <<endl;
   lassie1.printit();
   cout << "lassie2 (Dackel): " << endl;
   lassie2.printit();
   return 0;
}
```

Je nachdem, ob ein `hund`- oder ein `dackel`-Objekt die Methode `printit()` aufruft, setzt der Compiler Methode ein, die zur Klasse passt. Beim `dackel`-Objekt die der Unterklasse, beim `hund`-Objekt die der Oberklasse.

Der Anwender der beiden Klassen braucht sich nicht zwei Namen zu merken. Er kann nicht mehr aus Versehen die falsche Methode aufrufen. Man sagt, die Methode `printit()` der Unterklasse **überschreibt** die gleichnamige Methode der Oberklasse.

Es ist offiziell kein Overloading, denn beim Overloading konnte der Compiler schon anhand der unterschiedlichen Parameterliste unterscheiden, welche Funktion oder welche Methode eingesetzt wurde[1].

8.3.3 Zugriff auf Methoden der Oberklasse

Was muss man nun tun, wenn man für ein Objekt der Unterklasse die Methode der Oberklasse aufrufen möchte? Die beiden Methoden haben ja denselben Namen!

Die Lösung liegt in der Benutzung des Namensraum-Operators:

```
   lassie2.hund::printit(); // ruft Methode der Oberklasse auf
   lassie2.printit();       // ruft Methode der Unterklasse auf
```

[1] In Wirklichkeit ist es doch dem Overloading sehr ähnlich, denn der unsichtbar mitgeführte `this`-Zeiger hat je nach Art des Objekts einen unterschiedlichen Typ, und anhand dessen weiß der Compiler dann doch, welche Methode er auszuwählen hat.