Kapitel 7

Vererbungslehre und Verwandtes

7.1 Syntax

Ein Großteil des Vererbens funktioniert in C++ genauso wie in Java. Es gibt `private`, `protected` und `public` Funktionen und Variablen. Allerdings ist die Syntax leicht verschieden:

```
class Kind:public Mutter {
  ⋮
}
```

Statt `extends` muss `:public` stehen, wenn man das gleiche Verhalten wie in Java haben möchte: Variablen und Funktionen, die `public` sind, sind überall verwendbar. Die abgeleitete Klasse kann Variablen und Funktionen verwenden, die `protected` sind. Außerhalb der Basisklasse sind Variablen und Funktionen, die `private` sind, nicht zu benützen.

7.2 Intermezzo: Namensräume

Namensräume haben zwar nichts mit Vererbung zu tun, aber sie kapseln ähnlich wie Klassen Namen anderer Objekte (wie Funktionen, Variablen oder Klassen). Sie entsprechen den „Packages“ aus Java und wurden in C++ eingeführt, um dabei zu helfen, wenn man verschiedene Bibliotheken in einem Projekt verwenden will. Dabei tritt es regelmäßig auf, dass ein Name wie `list`, `set`, `string`, `point` oder `vector` in mehr als einer Bibliotheken vergeben wurde. Natürlich passen dabei die definierten Objekte in den seltensten Fällen zusammen.

Ein Ausweg ist die Konvention, jedes Objekt einer Bibliothek `FOO` mit dem Namen der Bibliothek zu beginnen, d.h. man verwendet `FOOlist`, `FOOset`, `FOOpoint` bzw. `FOOvector`. Der offensichliche Nachteil ist, dass es nicht sonderlich schön aussieht und viel zu tippen ist. Namesräume entschärfen das Problem derart, dass man den Präfix nur außerhalb der Bibliothek angeben muss:

```
// Bibliothek
namespace FOO {
  void f() {
    // something useful
  }
}

// Nutzer
int main() {
  FOO::f();
}
```

Auf Elemente des Namensraumes `FOO` greift man also genauso zu, wie auf Bezeichner, die aus einer Klasse stammen, indem man `FOO::` voranstellt. Die Namen aus der Standandbibliothek, die mit jedem C++-Kompiler mitkommt, sind ebenfalls in einem eigenen Namensraum, nämlich `std`. Dies erklärt die etwas kryptischen Bezeichner `std::cout` und `std::endl`.

Wenn man ein Objekt (wie Funktion, Variable oder Klasse) sehr oft verwendet, ist jedoch auch das Voranstellen von `FOO` bei der Verwendung lästig. Daher gibt es die Möglichkeit ein für alle mal (in dieser Datei) zu sagen, dass man mit `f` eigentlich `FOO::f` meint:

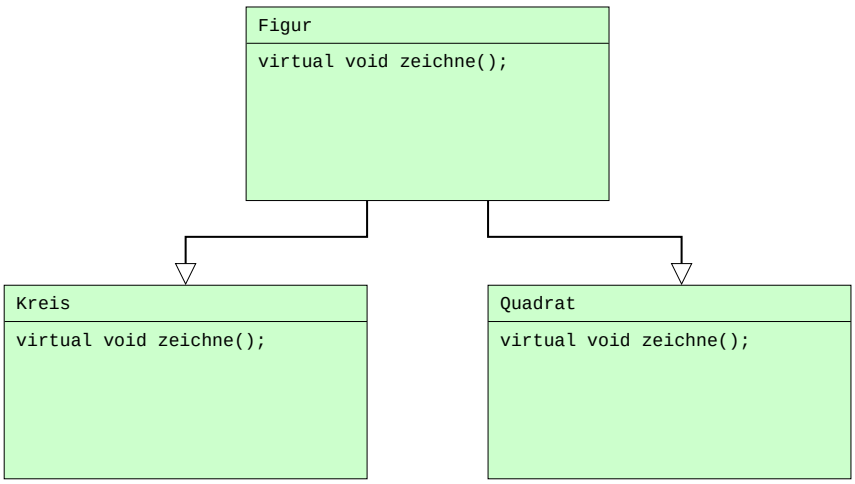
```
using FOO::f;

int main() {
  f();
}
```

Vor der Verwendung von `using` in Headerdateien muss aber gewarnt werden, da man damit allen Benutzern dieser Headerdatei seine Meinung aufzwingt, dass mit `f` in Wahrheit `FOO:f` gemeint ist.

7.3 Dynamisches und statisches Binden

Zurück zum Vererben. Betrachten wir folgende Situation:



Die Klassen `Kreis` und `Quadrat` sind beide von `Figur` abgeleitet und implementieren die Funktion `zeichne()`. Wenn man nun eine Instanz z.B. von `Kreis` hat, dann ist dies insbesondere eine Figur. Man kann daher eine Referenz von Figur oder einen Zeiger auf Figur damit initialisieren.

```
Kreis MeinKreis;
Figur *Zeiger;
Zeiger = &MeinKreis;
(*Zeiger).zeichne();
```

In der letzten Zeile wird dann die Funktion `zeichne` von `Kreis` aufgerufen. In Java ist dies das Standardverhalten bei Vererbung. In C++ muss man jedoch vor die Funktion in der Basisklasse `virtual` schreiben, damit dieser Mechanismus funktioniert. Damit werden alle Funktionen mit denselben Namen und Parametern in abgeleiteten Klassen automatisch virtuell. Im vorliegenden Beispiel also `zeichne()` in `Kreis` und `Quadrat`. Es schadet aber auch nicht dort ebenfalls `virtual` zu verwenden, sondern erinnert noch einmal daran.

Falls man kein `virtual` verwendet, wird nicht überprüft, von welchem Typ die Klasse in Wirklichkeit ist. Im obigen Beispiel würde daher die Funktion `zeichne` der Basisklasse aufgerufen werden. Man kann (und muss) also zwischen virtuellen und nicht virtuellen Funktionen unterscheiden. Der Vorteil von nicht virtuellen ist derjenige, dass zur Laufzeit nicht überprüft werden muss, welche Klasse vorliegt – Kreis oder Figur. Das spart Rechenzeit. Außerdem muss keine Tabelle mit den Funktionen vorgehalten werden. Das spart Speicherplatz. Wenn man also obigen Mechanismus nicht benötigt, dann ist es von Vorteil, wenn man diese Maschinerie auch nicht anschmeißt.

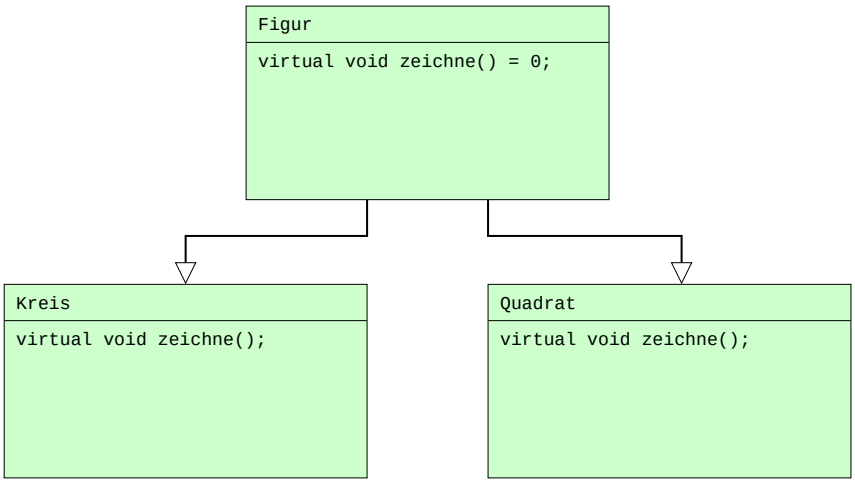
Ein Fall, in dem man ihn quasi immer braucht, ist der Destruktor. Falls der Destruktor nicht virtuell ist, und er für eine abgeleitete Klasse aufgerufen wird, die sich aber hinter einer Variable vom Typ einer Basisklasse verbirgt, dann wird der falsche Destruktor aufgerufen. Merke daher

Destruktoren müssen immer virtuell sein.

Eine Ausnahme von der Regel ist natürlich, wenn von einer Klasse nicht abgeleitet werden soll.

7.4 Abstrakte Klassen

Bei virtuellen Funktionen gibt es noch eine Besonderheit für den Fall, dass man sie eigentlich nicht definieren möchte (wie im obigen Fall `zeichne` für `Figur`). Man kann sie gleich 0 setzen, was gleichbedeutend damit ist, dass sie nicht existieren:



Klassen mit solchen Funktionen nennt man *abstrakte Klassen*. Sie können als Ersatz für Interfaces angesehen werden, die es in C++ nicht gibt.

- Eine Klasse ist automatische eine abstrakte Klasse, sobald es eine virtuelle Funktion gibt, die 0 ist.
- Abstrakte Klassen können nicht instanziiert werden.
- Im Unterschied zu Interfaces können aber Methoden definiert werden. Man kann also Teile der Klasse bereits vorfertigen und andere frei lassen.
- Man kann von einer abstrakten Klasse wieder Klassen ableiten, die ebenfalls abstrakt sind.
- Eine abgeleitete Klasse ist dann nicht mehr abstrakt, wenn alle 0-Funktionen definiert sind.

7.5 Aufgaben

1. Schreiben Sie eine Klasse `Person`, die einen Namen vom Typ `std::string` hat. (Dazu muss man die Header-Datei `string` einbinden.) Leiten Sie davon eine Klasse `Angestellter` ab, der auch noch ein Gehalt vom Type `unsigned int` hat.
2. Schreiben Sie eine virtuelle Funktion, die den Namen bzw. den Namen und das Gehalt ausgibt.
3. Verwenden Sie bei der Deklaration der Klasse `std::string` und bei der Definition mittels `using` nur `string`.