Notizen AIK ProgTech 3: `virtual`

Klaus Kusche, 2011/2012

Wo & Wie?

- Im **`class`** *vor den Prototyp* der betreffenden Methode **`virtual`** schreiben.
- Schon in der *Basisklasse*, denn in der abgeleiteten Klasse allein nutzt es nichts!

Funktionsweise

In C++ kann ein Pointer **`p`** (oder eine Referenz), der als "Pointer auf Objekte der Klasse **`A`**" deklariert ist (**`A *p;`**), *auch auf Objekte davon abgeleiteter Klassen* (**`B`**, **`C`** usw.) *zeigen*.

`virtual` beeinflusst, *welche Methode* für Objekte, auf die **`p`** zeigt, *aufgerufen wird* (bei **`p->myMeth(...)`**):

- **`myMeth`** ist **ohne** **`virtual`** deklariert:

  Der Compiler legt die Methode *schon zur Compilezeit* fest (*"statische Bindung"*):

  Er nimmt das **`myMeth`** aus derjenigen Klasse,
  mit der **`p`** *deklariert* wurde
  (bei uns: *immer* das **`mymeth`** von **`A`**, auch für **`B`**- und **`C`**-Objekte!).

- **`myMeth`** ist **mit** **`virtual`** deklariert:

  Die Methode wird *zur Laufzeit* gesucht (*"dynamische Bindung"*):

  Es wird im Objekt, auf das **`p`** gerade zeigt,
  nachgeschaut, zu welcher Klasse es wirklich gehört,
  und das **`myMeth`** aus der *tatsächlichen Klasse des Objektes* genommen
  (das kann das **`myMeth`** aus **`A`**, **`B`** oder **`C`** sein).

Warum?

- **`virtual`** ist das Verhalten, das man normalerweise erwartet und das *in den meisten Fällen sinnvoll* ist. Man wird also vor Methoden, die man in abgeleiteten Klassen überschreibt, fast immer **`virtual`** davorschreiben.

  Auch Java und fast alle anderen Programmiersprachen arbeiten immer so, als ob **`virtual`** dastünde, wenn sie Methoden aufrufen.

- **`virtual`** ist allerdings *langsamer*, weil es die richtige Methode erst zur Laufzeit suchen muss. Es braucht auch *mehr Speicher*, weil bei jedem Objekt seine Klasse und irgendwo die Liste aller Methoden einer Klasse gespeichert werden muss (bei Klassen ohne **`virtual`** fällt das weg!).

  Da bei C++ höchste Effizienz das Ziel ist, wurde die schnelle (aber meist "falsche") Methoden-Auswahl zum Normalfall gemacht, und die langsamere (aber meist "richtige") muss man sich mit **`virtual`** extra wünschen.

Hinweise:

- Konstruktoren können *nicht* virtuell sein!
- Destruktoren können virtuell sein.

  Eine (zwar technisch falsche, aber in der Praxis in 95 % aller Fälle zutreffende) Faustregel schreibt vor, den *Destruktor* auch *virtuell* zu machen, sobald die Klasse *irgendeine virtuelle Methode* enthält.
- **Achtung:** Wenn man im Code des Konstruktors oder des Destruktors virtuelle Methoden aufruft, wird immer die Methode der *eigenen* Klasse aufgerufen, nicht die aus der abgeleiteten Klasse: Der **`virtual`**-Mechanismus ist im Code von Konstruktor und Destruktor *abgeschaltet*!

"Rein virtuell"

Häufiger Fall:

- Die Basisklasse enthält den Prototyp (die Deklaration) einer *gemeinsamen Methode*, die in allen abgeleiteten Klassen vorhanden sein muss, aber *in jeder abgeleiteten Klasse anders implementiert* wird.
- Es gibt *keine sinnvolle Implementierung* dieser Methode für die *Basisklasse*, d.h. man kann für diese Methode in der Basisklasse *keinen Code* angeben.

==› Wenn man den Code in der Basisklasse *einfach weglässt*, kommt ein *Compiler-Fehler*!

==› Daher: Die Methode in der *Basisklasse* ***"rein virtuell"*** deklarieren,
d.h. "**`... = 0;`**" an den Prototyp anhängen.
Das heißt: "Für diese Methode gibt es in dieser Klasse *absichtlich keine Implementierung*"

Effekte:

- Die *Basisklasse* wird durch eine rein virtuelle Methode zur ***"abstrakten Klasse"***, d.h. man kann *keine Objekte* dieser Klasse mehr *anlegen* (weil die Objekte der Basisklasse die rein virtuelle Methode gar nicht "können", obwohl sie auch für diese Objekte im Prinzip aufrufbar wäre).

  Eine abstrakte Klasse ist nur eine *gedankliche Zusammenfassung* der *gemeinsamen Eigenschaften* aller abgeleiteten Klassen, aber keine Klasse, die "reale" Objekte darstellt:
  Jedes reale Objekt muss zu irgendeiner der abgeleiteten Klassen gehören.
- Jede *abgeleitete Klasse* ***muss*** alle rein virtuellen Methoden *überschreiben*, damit man Objekte der abgeleiteten Klasse anlegen kann (wenn nicht, ist auch die abgeleitete Klasse abstrakt).