Notizen AIK ProgTech 3: Copy-Konstruktor und `clone`-Trick

Klaus Kusche

Der Copy-Konstruktor

- … hat den Zweck, ein **neues Objekt als Kopie eines bestehenden Objektes** zu erzeugen.
- … wird aufgerufen:
  - Bei *explizitem Konstruktor-Aufruf*
    (in einer Deklaration, in einem **`new`**, …: **`myClass newObj(origObj);`**).
  - Implizit bei einer Deklaration mit *Initialisierungs-Zuweisung*
    (z.B. **`myClass newObj = origObj;`**).
  - Implizit für jedes Argument-Objekt bei *Call by Value* und für den Returnwert einer Funktion (falls er ein Objekt ist) bei *Return by Value*.
  - *Nicht* bei normaler Zuweisung! (hier wird ein =-Operator aufgerufen!!!)
- … hat eine **Referenz** auf ein Objekt der *eigenen Klasse*
  (nämlich dem zu kopierenden Original) als *einzigen Parameter*
  (normalerweise eine **`const`**-Referenz, weil das Original ja nicht geändert wird).

  Ganz wichtig: Vergisst man die Referenz, kommt es zu einer **Endlos-Rekursion**, da bei der Übergabe eines Objektes als "by Value"-Argument (als Kopie) ja wieder der Copy-Konstruktor aufgerufen wird!
- … wird vom Compiler *automatisch erzeugt*, wenn man ihn nicht selbst definiert (d.h. es gibt *immer* einen Copy-Konstruktor[1]).

  Der automatisch erzeugte Copy-Konstruktor macht eine *exakte (bitweise) Kopie des Original-Objektes*, aber *nicht* der Daten, auf die Pointer in diesem Objekt zeigen.

  Handelt es sich um ein Objekt einer *abgeleiteten Klasse*, ruft der automatisch erzeugte Copy-Konstruktor als Erstes den *Copy-Konstruktor der Vaterklasse* auf.

  Enthält das Objekt andere Objekte als *Member*, werden diese durch Aufruf ihres Copy-Konstruktors kopiert.
- … muss selbst definiert werden, wenn
  - … die Objekte **Pointer als Member-Variablen** enthalten (im Besonderen Pointer auf dynamisch angelegte Daten), damit beim Kopieren *neue Kopien* der Daten erzeugt werden, auf die die Pointer zeigen.

    Sonst zeigen die Pointer im Original und in der Kopie auf **dieselben Daten**, und das ist meistens *ganz schlecht*:
    - *Verändert* ein Objekt die Daten, auf die die Pointer zeigen, ändern sich damit auch die Daten, die das andere Objekt "sieht".
    - Gibt ein Objekt die Daten frei (z.B. im Destruktor),

1 Außer man deklariert den Copy-Konstruktor explizit, aber gibt absichtlich *keinen Code* dafür an, wenn man jedes Kopieren und jede Call-by-Value-Übergabe eines solchen Objektes verhindern will.

zeigen die Pointer im anderen Objekt "*ins Leere*".

- … die Objekte irgendeine **eindeutige Nummer** (fortlaufende Seriennummer o.ä.) enthalten, die nicht einfach kopiert werden darf, sondern in der Kopie frisch berechnet werden muss.
- … bei jedem Anlegen der Objekte (auch beim simplen Anlegen einer Kopie) irgendein **Code** ausgeführt werden soll (Initialisierung, Statistik, …, z.B. Anzahl der Objekte mitzählen).

Achtung in abgeleiteten Klassen:

- Der **automatisch erzeugte Copy-Konstruktor** ruft wie erwartet *automatisch den* **Copy-Konstruktor** *der Vaterklasse* auf.
- Ein **selbstgeschriebener Copy-Konstruktor** ruft hingegen *nur den* **Standard-Konstruktor** *der Vaterklasse* automatisch auf, wenn man nichts Anderes programmiert hat!

  ==› Die Member der Vaterklasse sind "leer" und werden **nicht** wie erwartet automatisch kopiert!

  ==› *Man muss in der Initialisierungsliste des Copy-Konstruktors der abgeleiteten Klasse als Erstes explizit den Copy-Konstruktor der Basisklasse aufrufen* (mit dem Original-Objekt als Argument), auch dann, wenn die Vaterklasse gar keinen selbstprogrammierten, sondern nur den automatisch erzeugten Copy-Konstruktor hat!

Der `clone`*-Trick*

Oft steht man vor folgendem Problem:

> Man hat einen Pointer, der als *"Pointer auf ein Objekt der Basisklasse"* deklariert ist, aber auf Objekte der Basisklasse *oder beliebiger davon abgeleiteter Klassen* zeigt, und möchte das Objekt, auf das der Pointer zeigt, kopieren.

Der Aufruf des Copy-Konstruktors der Basisklasse reicht nicht, denn dann würde die Kopie nur ein Objekt der Basisklasse sein, und alle Member der abgeleiteten Klasse würden *fehlen*.

Man bräuche so wie bei normalen Methoden den Virtual-Mechanismus, um automatisch den Copy-Konstruktor der "richtigen" Klasse aufzurufen, aber Konstruktoren können grundsätzlich *nicht virtuell* sein.

- Man behilft sich, indem man den Copy-Konstruktor in eine *virtuelle Methode* "einpackt". Diese Methode heißt üblicherweise **`clone`** und hat den Zweck,

  einen Pointer auf eine neu erzeugte Kopie des "eigenen" Objektes zu liefern
  (d.h. auf eine Kopie des Objektes, für das **`clone`** aufgerufen wurde).

- Die Methode **`clone`** hat folgende Deklaration in der Basisklasse **`Base`**:

  ```
  virtual Base* clone() const;
  ```

- *Alle Klassen* **`Abgel`**, deren Objekte kopiert werden sollen, erhalten ihren eigenen *Copy-Konstruktor* und folgende immer gleiche Implementierung von **`clone`** *auf der Basis des eigenen Copy-Konstruktors*:

  ```
  virtual Abgel* clone() const {
    return new Abgel(*this);
  }
  ```

  In der Basisklasse ist **`clone`** entweder ebenso definiert oder rein virtuell.

- Dann funktioniert folgender Code, egal zu welcher Klasse das Objekt gehört, auf das **`origPtr`** zeigt:

  ```
  copyPtr = origPtr->clone();
  ```