Programmieren C++: Grafische Klassen, Objekte mit Pointern: Zusammengesetze grafische Objekte

Klaus Kusche

In Fortsetzung des vorigen Grafik-Beispiels (nimm die
Musterlösung vom letzten Grafik-Beispiel, wenn du keine eigene Ausgangsbasis hast):

1.) Copy Konstruktor, `clone`

An *allgemeinen Erweiterungen* in allen Klassen sind zuerst einmal Folgende notwendig:

- **`GraObj`** und alle abgeleiteten Klassen bekommen einen *Copy-Konstruktor*.

  Achtung:

  - Der Copy-Konstruktor in **`GraObj`** darf den *Z-Wert* aus der vorigen Übung nicht einfach kopieren, sondern muss ihn in der Initialisierungsliste *zuerst explizit auf 0* setzen und *dann* erst im Code des Konstruktors mit **`setZ`** auf den *Wert des Originals*: Damit erreicht man, dass das kopierte Objekt korrekt *in die Z-Liste eingefügt* wird.
  - In den abgeleiteten Klassen nicht auf den Aufruf des *Copy-Konstruktors der Basisklasse* vergessen!

    Weiters rufen die Copy-Konstruktoren *der abgeleiteten Klassen* **`draw`** auf.

- Um das Kopieren von Objekten beliebiger Klassen durchführen zu können, muss man den virtuellen **`clone`**-Mechanismus implementieren (weil das Array ja gemischt Pointer auf **`Rect`**- und **`Circ`**-Objekte sowie später zusammengesetze Objekte enthalten kann, und jedes muss auf seine Art kopiert werden).

  In **`GraObj`** ist **`clone`** pure virtual, in den abgeleiteten Klassen wie üblich mittels Copy-Konstruktor implementiert.

- Die *Zuweisung* (**`operator=`**) müsste man bei Objekten, die Pointer enthalten, auch überschreiben, aber das können wir noch nicht, also lassen wir das weg.
- Wir werden im zweiten Punkt auch die Methoden **`setPos`**, **`setSize`**, **`move`**, **`scale`** und **`rotate`** überschreiben. Mach sie daher in der Basisklasse **`virtual`** !

Ansonsten sollte die Implementierung von **`GraObj`**, **`Rect`**, **`Circ`** und **`Color`** gleich bleiben.

Mit diesen Änderungen sollte dein **`main`** ein *grafisches Objekt kopieren* können (verschiebe die Kopie dann, damit sie nicht genau über dem Original liegt!), egal, ob das Objekt jetzt ein Rechteck oder eine Ellipse ist.

2.) Zusammengesetze Objekte

Wir leiten von unserer abstrakten Klasse **GraObj** eine dritte Klasse **Comp** ("complex" oder "composite") für *zusammengesetzte Objekte* ab, d.h. für Objekte, die selbst *aus mehreren Ellipsen und Rechtecken* (und ev. wieder **Comp**-Objekten) bestehen.

Grundidee ist Folgende:

- Ein zusammengesetzes Objekt enthält einen *Pointer* auf ein *dynamisch angelegtes Array von* **GraObj**-*Pointern*.
- Die Elemente dieses Arrays zeigen auf die Subobjekte.
- Operationen auf dem zusammengesetzten Objekt werden im Normalfall an alle Subobjekte durchgereicht.

Im Detail:

- Neben dem Pointer auf das Array mit den Subobjekt-Pointern (das ist ein Pointer auf **GraObj**-Pointer!) bekommt **Comp** noch zwei Member-Variablen für die *angelegte Größe dieses Arrays* und für die *Anzahl* der derzeit darin enthaltenen Subobjekte. Alle Member sind **protected**.
- Die *Größen-Member* (Breite und Höhe ab Mittelpunkt), die **Comp** von **GraObj** erbt, lassen wir für zusammengesetzter Objekte der Einfachheit halber vorläufig unberücksichtigt (siehe Zusatzaufgabe):
  - Im Konstruktor werden sie auf 0 gesetzt.
  - **setSize** lassen wir auch vorläufig leer.
- Ebenso wird die *Farbe* zusammengesetzter Objekte auf 0/0/0 gesetzt und nicht weiter berücksichtigt.
- Der *Konstruktor* für Comp hat nur die x- und y-Koordinate für den *Mittelpunkt* des Objektes sowie die gewünschte *Größe des Subobjekte-Arrays* (maximal mögliche Anzahl von Subobjekten) als Parameter.

  Das Pointer-Array wird *dynamisch* angelegt, aber bleibt uninitialisiert: Die Subobjekte werden erst nachträglich dazugefügt.
- Daneben gibt es einen *Copy-Konstruktor* und **clone**:

  Der *Copy-Konstruktor für zusammengesetze Objekte* muss nach dem Aufruf des Vater-Copy-Konstruktors ein *neues Pointer-Array* anlegen und die *Subobjekte* aus dem Array des Originals *kopieren* (mit **clone**), nicht nur die Pointer auf die Subobjekte (d.h. das kopierte zusammengesetze Objekt soll *eigene Subobjekte* haben, nicht Pointer auf die des Originals!).
- Der *Destruktor* muss *zuerst die Subobjekte alle freigeben* und *dann das Array* selbst.
- In **Comp** kommt eine *neue Member-Funktion* **add** dazu, die mit einem **GraObj**-Objekt-Pointer aufgerufen wird und eine Kopie (**clone** !) des angegebenen Objektes zum Array der Subobjekte dazufügt (prüfen, ob noch Platz ist, hinten anhängen, Belegt-Zähler erhöhen).

  Der Returnwert soll **bool** sein (**true**: erfolgreich, **false**: Array voll).

- **`draw`** und **`undraw`** rufen einfach in einer Schleife
  für alle Subobjekte **`draw`** und **`undraw`** auf.
- **`setPos`** berechnet den Unterschied zwischen der aktuellen eigenen Position
  und der Ziel-Position und ruft dann das eigene **`move`** auf.
- **`move`** verschiebt nicht nur den eigenen Mittelpunkt,
  sondern ruft auch für alle Subobjekte **`move`** auf.
- **`setSize`** macht wie oben angegeben vorläufig nichts.
- **`scale`** ist kompliziert, es macht bis zur Zusatzaufgabe auch einfach nichts.
- Bei **`rotate`** passieren für jedes Subobjekt zwei Dinge:
  - Alle Subobjekte müssen mit ihrem **`rotate`** rotiert werden.
  - Die *Position jedes Subobjektes* in Relation zum eigenen Mittelpunkt
    muss um 90 Grad gedreht und neu gesetzt werden:
    Der neue x-Abstand ist der alte y-Abstand mit umgekehrtem Vorzeichen,
    der neue y-Abstand ist der alte x-Abstand.
    Dann wird der *neue Mittelpunkt* des Subobjektes ausgehend vom eigenen
    Mittelpunkt berechnet und das Subobjekt mit **`setPos`** dorthin *verschoben*.

Schreib dazu wieder ein kleines *Hauptprogramm* zum Testen.
Es sollte u.a. mit mehreren **`add`** ein zusammengesetzes Objekt basteln
und das Kopieren zusammengesetzter Objekte testen.

Zusatzaufgabe:

Kannst du auch die *Größe* (geerbte Member für Breite und Höhe) richtig behandeln?

Ich empfehle folgende Vorgangsweise:

- **`scale`** ist kompliziert, denn es muss *für jedes Subobjekt zwei Dinge* machen:
  - **`scale`** aufrufen.
  - Die *x- und y-Abstände* zwischen dem eigenen Mittelpunkt
    und dem Mittelpunkt des Subobjektes berechnen,
    diese Abstände *skalieren* und zum eigenen Mittelpunkt dazurechnen,
    und das Subobjekt mit **`setPos`** auf die resultierende *neue Position* setzen.
- Mach in **`Comp`** eine *private Methode* ohne Parameter und Returnwert,
  die die beiden Member *neu berechnet*, indem sie alle Subobjekte durchgeht
  und das Maximum von deren Breite bzw. Höhe zuzüglich
  des horizontalen bzw. vertikalen Abstandes (Absolutwert nehmen!)
  von deren Mittelpunkt zum eigenen Mittelpunkt berechnet.
- Rufe diese Methode am Ende von **`add`**, **`scale`** und **`rotate`** auf.
- Wenn deine **`Comp`**-Objekte eine korrekte Größe enthalten,
  dann lässt sich auch **`setSize`** ganz einfach implementieren:
  - Berechne, wieviel Prozent der gespeicherten bisherigen Breite und Höhe
    die neue Breite und Höhe sind.
  - Rufe das eigene **`scale`** mit diesem Prozentsatz
    von alter und neuer Breite bzw. Höhe auf.