

Programmieren C++: Statische Member und Methoden, STL list: Automatischer Farbkreis, Verzeichnis aller Objekte

Klaus Kusche

Dieses Beispiel ist eine Fortsetzung des Beispiels mit den geometrischen Objekten, nimm deine Lösung (oder meine Musterlösung) des Schrittes mit Vererbung und Kreisen als Ausgangsbasis (mit "normalem" Copy-Konstruktor ohne Farb- oder Größenänderung).

1. Schritt: Automatischer Farbkreis: Klassenweit Objekte mitzählen

Eine wichtige Anwendung statischer Member und Methoden sind klassenweite Zähler, z.B. um jedem neu erzeugten Objekt automatisch eine fortlaufende Nummer zu geben.

Das Hauptprogramm dieses Beispiels ruft für die Farbe neu erzeugter grafischer Objekte den Standard-Konstruktor von **Color** auf.

Color hat aber noch keinen Standard-Konstruktor, er muss erst implementiert werden: Er soll bei jedem Aufruf die nächste Farbe aus einem zyklischen Farbkreis liefern.

Will man einen Farbkreis mit 192 einzelnen Farben, so liefert die folgende Berechnung ausgehend von einer fortlaufenden ganzen Zahl x die drei Farbkomponenten der jeweils nächsten Farbe im Farbkreis:

```
int n = (x % 192) * 4 + 2;  
rot = max(0, min(-256 + n, 768 - n));  
gruen = max(0, min(n, 512 - n));  
blau = max(0, max(-512 + n, 256 - n));
```

Es sind also zwei Erweiterungen der Klasse **Color** nötig:

- Die Klasse bekommt einen Zähler als klassenweites Member (dazu musst du an zwei Stellen etwas tun!).
- Weiters bekommt sie einen Standard-Konstruktor (Implementierung bitte außerhalb des **class!**). Er erhöht zuerst den klassenweiten Zähler um eins und berechnet dann mit dem oben angegebenen Code aus diesem Zähler die Farbwerte des neu erzeugten **Color**-Objektes (für **min** und **max** brauchst du den Header **algorithm**).

2. Schritt: Methode drawAll: Liste aller Objekte

Die zweite wichtige Anwendung klassenweiter Member und Methoden sind Klassen, die intern in irgendeiner Form automatisch über alle Objekte der Klasse Buch führen.

Das Zeichnen unserer grafischen Objekte hat zwei Schwächen:

- Wenn ein Objekt weggelöscht, bewegt oder skaliert wird, hinterlässt es unschöne "Löcher" bzw. "Ränder" in darunterliegenden Objekten.
- Wir können nicht kontrollieren, in welcher Reihenfolge übereinander liegende Objekte gezeichnet werden: Das zuletzt bewegte Objekt liegt immer obenauf.

Das wollen wir lösen, und zwar in einem ersten Schritt wie folgt:

- Die Klasse verwaltet intern eine Liste aller gerade existierenden GraObj-Objekte, und zwar in der Reihenfolge ihrer Erzeugung.
- Eine neue Methode **drawALL** geht diese Liste durch und zeichnet alle darin enthaltenen Objekte in dieser Reihenfolge neu.

Hinweise:

- Die Liste soll alle grafischen Objekte (Rechtecke und Ellipsen) gemeinsam erfassen. In welcher Klasse gehört sie daher implementiert?
- Wir verwenden für die Liste das Container-Template list aus der STL (intern ist **list** als doppelt verkettete Liste implementiert). Als Template-Parameter müssen wir den Typ der Elemente angeben. Unsere Liste zeigt auf beliebige grafische Objekte (sie enthält keine Kopien der Objekte!). Welchen Element-Typ hat sie daher?
- Die Liste aller Objekte gibt es einmal pro Klasse, d.h. für alle Objekte gemeinsam. Auch die Methode **drawALL** zum neu Zeichnen aller Objekte soll für die ganze Klasse aufgerufen werden, ohne Angabe eines bestimmten Objektes (bzw. auch dann, wenn gerade gar keine Objekte existieren). Wie sind beide daher deklariert?
Was brauchst du für dieses Listen-Member im .cpp-File?
- Weiters soll es abgeleiteten Klassen nicht möglich sein, die Liste aller Objekte irgendwie zu manipulieren. Welche Sichtbarkeit sollte die Liste daher haben?
- Die Liste soll automatisch aktualisiert werden, ohne dass der “Verwender” der Klasse **GraObj** dafür seinen bestehenden Code modifizieren muss:
Wenn ein neues **GraObj** entsteht, soll es sich von selbst hinten an die Liste hängen, und wenn ein **GraObj** gelöscht wird, soll es sich von selbst aus der Liste entfernen.
Such dir die dafür nötigen **list**-Methoden aus der Online-Doku von **list** (ich empfehle <http://www.cplusplus.com/reference/list/list/>).
An welchen Stellen deines **GraObj**-Codes gehören sie eingefügt, und mit welchem Parameter?
Wenn das Programm läuft, kannst du auch einmal probierhalber “hinten anhängen” durch “vorne anhängen” ersetzen. Dann sollten die Objekte in umgekehrter Reihenfolge gezeichnet werden (jüngste zuunterst, älteste obenauf).
Achtung: Bisher reichte der implizite Copy-Konstruktor für **GraObj**. Jetzt brauchen wir einen expliziten Copy-Konstruktor, der wie bisher alle Member einfach kopiert, aber zusätzlich das neu entstandene Objekt in die Objektliste einträgt!
- **drawALL** hat weder Parameter noch einen Returnwert und ist einfach: Einmal die Liste durchwandern und für jedes Objekt **draw** aufrufen (du wirst dafür entweder eine **for**-Schleife oder eine Schleife mit einem **list-Iterator** brauchen, geht am einfachsten mit **auto**).
- Der Code von **Rect** und **Circ** sollte völlig unverändert bleiben!

Zum Hauptprogramm:

Das Hauptprogramm im Angabe-Code ist noch das aus dem letzten Vererbungs-Beispiel (Schlangen). Damit sieht man die Effekte zwar ein bisschen, aber noch nicht wirklich gut. Ich empfehle, das Hauptprogramm ungefähr wie folgt umzubauen, um den Effekt deutlich hervorzuheben:

- Das Hauptprogramm enthält wieder ein Array von Objekten. Dieses Array wird zuerst einmal mit einer zufälligen Mischung von Rechtecken und Ellipsen befüllt: Für alle Elemente wird mittels einer Hilfsfunktion ein neues Objekt erzeugt.
- Diese Hilfsfunktion, die ein neues, zufällig erzeugtes, dynamisch angelegtes, grafisches Objekt liefert, ist im Vergleich zum vorigen Beispiel etwas modifiziert:
 - Standard-Konstruktor für die Farbe, damit die Objekte automatisch aufeinanderfolgende Farben des Farbkreises bekommen.
 - Bildmitte, langsamere Fluggeschwindigkeit (**-3 ... +3**).
 - Deutlich größere, zufällig festgelegte Breite und Höhe (**1 ... 150**), damit sich die Objekte stark überlappen.
- Die Hauptschleife lässt wieder alle Objekte fliegen. Diesmal lasse ich die Objekte allerdings nicht vom Rand abprallen (d.h. **fly** wird ohne Parameter oder mit **false** aufgerufen): Sobald ein Objekt am Rand ansteht, wird das Objekt vernichtet und das Array-Element mittels Hilfsfunktion mit einem neuen Objekt befüllt.

Damit sollte man die Probleme unseres **draw** deutlich sehen:

- Die Objekte hinterlassen beim Verschieben einen schmalen schwarzen Rand auf dem darunterliegenden Objekt.
- Die Reihenfolge, in der die Objekte übereinanderliegen, sieht zufällig aus.

Wenn man jetzt unmittelbar vor dem **sdUpdate**-Aufruf einen **drawAll**-Aufruf einfügt, der alle Objekte in Erzeugungs-Reihenfolge neu zeichnet, sollten beide Probleme gelöst sein.

Zusatzaufgabe:

Unsere Verwendung der Liste hat noch ein Performance-Problem: Das Löschen eines Elementes aus einer Liste ist sehr schnell, wenn man bereits einen Pointer auf das zu löschende Element hat, aber (vor allem bei langer Liste) sehr langsam, wenn man die Liste erst nach dem zu löschenden Element durchsuchen muss, weil man nur dessen Wert, aber nicht dessen Position in der Liste kennt.

Genau das passiert hier aber im Destruktor.

Wir wollen uns also in jedem GraObj das dazugehörige Listen-Element merken:

- **GraObj** bekommt ein zusätzliches, privates Member vom Typ "Iterator auf unsere Liste".
- In den beiden Konstruktoren wird das **push_back** bzw. **push_front** durch ein **insert** ersetzt: **insert** liefert einen Iterator auf das neu eingefügte Element, und diesen Iterator speichern wir in unserem Member.
- Im Destruktor wird der "Löschen nach Wert"-Aufruf durch einen "Löschen per Iterator"-Aufruf (siehe Online-Doku) ersetzt.

3. Schritt: Beliebige Tiefen-Ordnung der Objekte: **set** nach Z-Wert sortiert

Mit der Liste aller Objekte ist nur ein Zeichnen in Erzeugungs-Reihenfolge oder in umgekehrter Erzeugungs-Reihenfolge effizient möglich.

Wir wollen das Programm so erweitern, dass man die Tiefe jedes Objektes (d.h. vor oder hinter welchen Objekten es gezeichnet wird) frei festlegen kann. Der Wert, der die Tiefe festlegt, heißt üblicherweise Z-Wert (dritte Dimension). Ohne Angabe einer Tiefe soll die Reihenfolge wie bisher bleiben.

Dafür bekommen unsere Objekte zwei neue int-Member:

- Eines für den frei wählbaren Z-Wert.
- Eines für eine automatisch vergebene, fortlaufende Nummer (für eine eindeutige Reihung bei gleichem Z-Wert und für das Default-Verhalten).

Unsere Objekte sollen nach diesen beiden Members wie folgt sortiert werden:

- Nach aufsteigendem Z-Wert.
- Bei gleichem Z-Wert nach aufsteigender fortlaufender Nummer.

Zur Speicherung aller Objekte verwenden wir den vordefinierten Container-Typ **set**. Ein **set** ist intern ein balancierter Binärbaum, d.h. es erlaubt ein effizientes Durchlaufen aller Elemente in aufsteigender Reihenfolge und ein effizientes Einfügen und Löschen an beliebiger Stelle unter Beibehaltung der Sortierung.

Normalerweise sortiert **set** nach dem normalen <-Vergleich. Das hilft uns aber nichts: Für Pointer ist bereits ein < vordefiniert, das nach aufsteigendem Adresswert sortiert (also für uns nicht brauchbar), und dieses < lässt sich auch nicht einfach überschreiben.

Es gibt viele verschiedene Wege, bei einem vordefinierten Container eine eigene Sortierung anzugeben. Hier ist eine Hilfs-Klasse mit Funktions-Operator am einfachsten:

- Deklariere eine Hilfs-Klasse, die nur einen öffentlichen ()-Operator enthält. Er braucht zwei Parameter vom Typ **const GraObj *** und returniert einen **bool**.
- Damit du diese Klasse vor der **GraObj**-Klasse schreiben kannst und dabei **GraObj** akzeptiert wird, brauchst Du eine Vorab-Deklaration von **GraObj** als **class**.
- Mach diese Klasse in **GraObj** zum Freund, damit Du beim Vergleichen direkt auf die **GraObj**-Member zugreifen kannst.
- Implementiere den ()-Operator wie ein < : Er soll genau dann true liefern, wenn das linke Objekt den kleineren Z-Wert hat, oder wenn beide denselben Z-Wert haben und das linke die kleinere Nummer. Du solltest den Code dafür außerhalb des **class** schreiben.
- Unser **set**-Typ bekommt dann zwei Template-Parameter: **GraObj *** als Typ der Elemente im **set** und die Hilfs-Klasse als Sortier-Typ (ich habe mir für diesen **set**-Typ ein **typedef** gemacht, um Tipparbeit zu sparen).

Weitere Änderungen:

- Das bestehende klassenweite list-Member wird durch ein **set**-Member ersetzt. Dazu kommen die oben genannten 2 Member für Z-Wert und fortlaufende Nummer, sowie ein klassenweiter Zähler zur Vergabe der fortlaufenden Nummern. Alle diese Member sollen nur in **GraObj** sichtbar sein.

- In **GraObj**, **Rect** und **Circ** bekommt der Konstruktor ganz hinten einen zusätzlichen Parameter für den Z-Wert (mit Default-Wert 0, wenn man ihn nicht angibt). **Rect** und **Circ** reichen ihn an **GraObj** durch, **GraObj** speichert ihn im Member.
- Der **GraObj-Copy-Konstruktor** muss auch den Z-Wert kopieren.
- Der **GraObj-Konstruktor** und der **GraObj-Copy-Konstruktor** müssen dem Objekt die nächste fortlaufende Nummer geben.
- Die Einfüge- und Rauslösch-Methoden heißen bei **set** anders als bei **list**.
- Für den Z-Wert soll es eine Get-Methode und eine Set-Methode geben. Get funktioniert wie üblich, Set ist etwas komplizierter: Da man den Sortierschlüssel-Wert eines Baum-Elementes nicht einfach ändern darf, muss man das eigene Objekt aus dem Baum entfernen, den Z-Wert ändern, und das Objekt wieder in den Baum einfügen.

Zum Hauptprogramm:

- Wenn man das Hauptprogramm aus Schritt 2 unverändert übernimmt, sollte das Programm wie bisher arbeiten: Da alle Z-Werte **0** sind, erfolgt die Sortierung nach aufsteigender fortlaufender Objektnummer.
- Wenn man dem Konstruktor als Z-Wert eine absteigende Zahl übergibt (verwende dafür eine statische Zählvariable in der Objektanlege-Hilfsfunktion), dann ergibt sich die umgekehrte Reihenfolge.
- Wenn man dem Konstruktor als Z-Wert eine Zufallszahl übergibt, wird das neue Objekt zufällig zwischen den bestehenden Objekten eingefügt.