Programmieren C++: Grafische Klassen: Vererbung: Ellipsen

Klaus Kusche

Diese Übung baut auf dem Beispiel aus der vorigen Übung auf.
Nimm die *Musterlösung der vorigen Übung*, wenn du keine eigene Ausgangsbasis hast!

Wenn du es schaffst: Schreib wieder einen separaten **.cpp**- und **.h**-File pro Klasse und achte darauf, überall die richtigen Files zu inkludieren (und nicht zu viele!).
Wenn nicht, darfst du auch das gesamte Beispiel in einem File codieren.

1. Schritt: Vererbung: Rechtecke und Ellipsen

Neben Rechtecken soll unser Programm auch Ellipsen zeichnen können.
Es gibt dazu die Funktion **sdlDrawCirc** in meinem **sdlinterf**.
Die Ellipsen sollen eine *eigene Klasse* **Circ** werden.

Da alle Member-Variablen und ein Großteil des Codes von **Circ** ident zu **Rect** sind (nur **draw** und **undraw** müssen statt **sdlDrawRect** jetzt **sdlDrawCirc** aufrufen), und da wir Rechtecke und Ellipsen in unserem Hauptprogramm gemeinsam bzw. gleich behandeln können wollen (sie haben ja beide genau dieselbe Schnittstelle, d.h. bieten dieselben Methoden), *leiten wir sowohl* **Rect** *als auch* **Circ** *von einer gemeinsamen Vaterklasse* **GraObj** *ab*.

- **GraObj** enthält alles, was alle abgeleiteten Klassen *gemeinsam* haben.
  In unserem Fall sind das *alle Member-Variablen* und *alle Methoden*
  (auch **draw** und **undraw** haben zwar verschiedene Implementierungen,
  aber in allen Klassen den gleichen Prototypen). Es ist daher am einfachsten,
  wenn du die Klasse **Rect** aus der alten Musterlösung in **GraObj** *umbenennst*.
- **Rect** und **Circ** enthalten *nur jene Dinge*, in denen sich Rechtecke und Ellipsen *unterscheiden*. Das sind
  - die *Implementierungen* von **draw** und **undraw**,
  - sowie der jeweilige *Konstruktor* und *Destruktor*.

  Details dazu in den Hinweisen!

  Beide Klassen sind bis auf den Namen und die SDL-Aufrufe ident. Am schnellsten geht es, wenn du eine davon schreibst und dann kopierst und änderst.

Hinweise:

- Auf die *Member-Variablen* in **GraObj** für Farbe, Position und Größe sollen auch die *abgeleiteten Klassen* direkt zugreifen können, *nicht* aber "fremder" Code.
  Was muss daher im **class** *vor* diesen Member-Deklarationen stehen?
- Überlege dir, wie die *Konstruktoren* von **Rect** und **Circ** aussehen müssen:
  - Alle *Membervariablen* sind schon in der Basisklasse deklariert und werden daher auch wie bisher in der *Initialisierungsliste des Konstruktors der Basisklasse* **GraObj** initialisiert (man kann in einer Initialisierungsliste nur Member der *eigenen* Klasse initialisieren, *nicht geerbte*!).

- Die *Initialisierungsliste* in den Konstruktoren der *abgeleiteten Klassen* initialisiert *keine* Member, sondern ruft *nur* die *Initialisierung der Basisklasse* mit den angegebenen Parametern auf.

  Was musst du dafür in die Initialisierungsliste schreiben?!

- Die Aufrufe von **draw** und **undraw** passieren im Code des Konstruktors und Destruktors der *abgeleiteten Klasse* (weil sie ja das **draw** und **undraw** der jeweiligen abgeleiteten Klasse aufrufen sollen, nicht das der Basisklasse), in der *Basisklasse* enthalten Konstruktor und Destruktor vorläufig *keinen Code* (auch kein **draw** und **undraw**).

- **GraObj** selbst deklariert zwar **draw** und **undraw** (denn alle von **GraObj** abgeleiteten Objekte kann man zeichnen und weglöschen), kann aber selbst keinen sinnvollen Code dafür enthalten (weil die Klasse **GraObj** ja nicht weiß, wie ein **Rect**, ein **Circ** oder zukünftige weitere Klassen gezeichnet werden).

  Eigentlich dürfte man gar keine **GraObj**-Objekte anlegen können,
  sondern *nur* Objekte davon abgeleiteter Klassen.
  Wie man das korrekt löst, lernen wir erst später im Detail.

  Vorläufig machen wir nur Folgendes: In **GraObj** zeichnen wir in **draw** und **undraw** *nur den Mittelpunkt* des Objektes (mit **sdlDrawPoint**), damit wir sehen, wenn das **draw** und **undraw** von **GraObj** versehentlich doch aufgerufen werden.

- Wenn du dein Programm jetzt probierst, wirst du feststellen,
  dass entweder gar nichts oder nur Punkte statt Rechtecken gezeichnet werden.
  Der Grund ist, dass alle geerbten Methoden der Basisklasse (**move**, **rotate**, …)
  nur das **draw** und **undraw** der *Basisklasse* **GraObj** aufrufen,
  auch wenn sie für ein **Rect**- oder **Circ**-Objekt aufgerufen werden.

  Um das zu beheben, kommt in die Kopfzeile von **draw** und **undraw** ganz vorne (vor den Returntyp **void**) das Wort **virtual**, und zwar in allen drei Klassen:
  Das bewirkt, dass C++ bei jedem **draw**- und **undraw**-Aufruf *zur Laufzeit nachschaut, zu welcher Klasse das betreffende Objekt wirklich gehört*, und das **draw** und **undraw** genau dieser Klasse aufruft.

- Vor den *Destruktur* (*nicht* den Konstruktor!) kommt ebenfalls ein **virtual**, damit auch der Destruktor-Code in den abgeleiteten Klassen ausgeführt wird, wenn man ein Objekt löscht, von dem man nur weiß, dass es von **GraObj** abgeleitet ist (denn der Destruktor ruft ja für jede abgeleitete Klasse ein *anderes* **undraw** auf).

Wenn du die drei Klassen richtig geschrieben hast, solltest du im **main** die Deklaration deines Rechteck-Objektes von **Rect** auf **Circ** (und zurück) ändern können,
ohne sonst irgendetwas im Code zu verändern, und es sollte dementsprechend
ein Rechteck oder eine Ellipse herumfliegen.

2. Schritt: Hauptprogramm mit mehreren Objekten, `new`

Das **`main`** aus dem vorigen Schritt zeigt die Fähigkeiten von C++ noch nicht wirklich:
Da unser herumfliegendes Objekt ausdrücklich als **`Rect`**- oder **`Circ`**-Objekt deklariert ist, kann der Compiler schon allein aus der Deklaration erkennen, was er aufrufen muss.

In der Praxis ist es aber meist so, dass man mehrere Objekte
verschiedener abgeleiteter Klassen gemeinsam verarbeiten will.

Damit man die Vererbung und den **`virtual`**-Mechanismus auch "bei der Arbeit" sieht, sollte dein Hauptprogramm ein *Array von Pointern* auf **`GraObj`**-Objekte enthalten, von denen einige auf **`Rect`**-Objekte und einige auf **`Circ`**-Objekte zeigen
(erinnere dich: Ein Pointer, der als Pointer auf Objekte der Klasse **`GraObj`** deklariert ist, darf auch auf Objekte aller davon abgeleiteten Klassen zeigen).

Fülle das Array, indem du in einer Schleife entsprechend viele Objekte einzeln mit **`new`** anlegst (gemischt **`Rect`**- als auch **`Circ`**-Objekte, z.B. gesteuert durch eine Zufallszahl).

Falls dein Programm nicht ohnehin eine Endlosschleife ist:
Gib die Objekte im Array am Ende des Programmes einzeln in einer Schleife wieder frei (*vor* dem **`sdlExit()`**, sonst gibt es einen Absturz, wenn der Destruktor **`undraw`** aufruft und die Grafik nicht mehr aktiv ist!).

Rufe dann für jedes der im Array gespeicherten Objekte in einer Schleife irgendeine grafische Methode (**`move`**, ...) auf. Im einfachsten Fall kannst du in meiner alten Musterlösung mit dem herumfliegenden Rechteck um den gesamten Inhalt der Endlosschleife außer dem **`sdlUpdate`** und dem **`sdlMilliSleep`** eine **`for`**-Schleife über alle deine Objekte machen und überall die alte Rechteck-Variable **`r`** durch das i-te Element deines Arrays ersetzen (was musst du noch an allen diesen Stellen ändern, wenn du es jetzt mit einem Pointer auf das Objekt zu tun hast statt mit dem Objekt selbst?).

Noch schöner schaut es aus, wenn du die Variablen **`dX`** und **`dY`** ebenfalls durch zwei (zufällig mit -10 bis +10 initialisierte) Arrays **`dX`** und **`dY`** mit ebensovielen Elementen wie Objekten ersetzt, damit die aktuelle Flugrichtung und -geschwindigkeit für jedes Objekt separat gespeichert und geändert werden kann.

Wenn das funktioniert, kannst du probehalber einmal das **`virtual`** bei **`draw`** und **`undraw`** weglassen. Was passiert und warum?