Abend 6
Vererbung und Polymorphie

Ziel, Inhalt

- Wir sehen heute weitere Beispiele für Vererbung und wie sich Objekte von Klassen wie andere Objekte verhalten.

Abend 6 Vererbung und Polymorphie 1
Ziel, Inhalt 1
Vererbung und Polymorphie 2
Vererbung und statische Bindung 2
Klasse Auto 2
Klasse Cabrio 3
main 4
Übung 4
Ergänzung 5
Statische casts 5
Polymorphe Klassen und dynamische Bindung 6
Das Schlüsselwort *virtual* 6
Klasse Auto mit virtueller Methode a*usgabe* 7
Virtueller Destruktor 7
Übung „Bewegliche, farbige Objekte“ 9

Vererbung und Polymorphie

Vererbung und statische Bindung

Bis jetzt kennen wir Vererbung mit statischer Bindung. Das heisst wenn ein Objekt einer Klasse verwendet wird, bestimmen wir zur Kompilierzeit, welche Methode aufgerufen wird. Hier ein Beispiel um diesen Sachverhalt zu verdeutlichen.

Klasse Auto

Auto.h

```
#ifndef AUTO_H
#define AUTO_H

class Auto
{
public:
    // Konstruktor
    Auto();
    //Destruktor
    ~Auto();

    // Methoden:
    void ausgabe() const;
};

#endif
```

Auto.cpp

```
#include "Auto.h"
#include <iostream>

using namespace std;

Auto::Auto()
{
    cout << "Ein Auto wurde erstellt" << endl;
}

Auto::~Auto()
{
    cout << "Ein Auto wurde zerstoert" << endl;
}

void Auto::ausgabe() const
{
    cout << "Ich bin ein Auto" << endl;
}
```

Klasse Cabrio

Die Klasse Cabrio ist von der Klasse Auto abgeleitet, das heisst ein Objekt der Klasse Cabrio ist immer auch ein Objekt der Klasse Auto. In diesem Beispiel redefinieren wir die Methode *ausgabe*.

Cabrio.h

```
#ifndef CABRIO_H
#define CABRIO_H

#include "Auto.h"

class Cabrio : public Auto
{
public:
    // Konstruktor
    Cabrio();
    //Destruktor
    ~Cabrio();

    // Methoden:
    void ausgabe() const;
};

#endif
```

Cabrio.cpp

```
#include "Cabrio.h"
#include <iostream>

using namespace std;

Cabrio::Cabrio()
{
    cout << "Ein Cabrio wurde erstellt" << endl;
}

Cabrio::~Cabrio()
{
    cout << "Ein Cabrio wurde zerstoert" << endl;
}

void Cabrio::ausgabe() const
{
    cout << "Ich bin ein Cabrio" << endl;
}
```

main

Hier ein main, das ein Array von sechs *Auto*-Zeigern erzeugt und jedem von diesen Zeigern entweder ein Auto oder ein Cabrio zuweist:

```
#include "Auto.h"
#include "Cabrio.h"

// Konstante für die Anzahl Autos
const int Anzahl = 6;

int main()
{
    // Erzeuge ein Array von sechs Autozeigern
    // und initialisiere dieses Array auf 0, das heisst
    // alle sechs Zeiger sind 0
    Auto* autoZeiger[Anzahl] = { 0 };

    for(int i = 0; i < Anzahl; ++i)
    {
        // falls i durch zwei den Rest 0 ergibt
        // also immer wenn i eine gerade Zahl
        // ist wird ein Auto Objekt erzeugt
        // sonst ein Cabrio
        if(0 == i % 2)
        {
            autoZeiger[i] = new Auto;
        }
        else
        {
            autoZeiger[i] = new Cabrio;
        }
    }

    // für jeden ausgabe aufrufen
    for(int j = 0; j < Anzahl; ++j)
    {
        autoZeiger[j]->ausgabe();
    }

    // und jeden löschen
    for(int k = 0; k < Anzahl; ++k)
    {
        delete autoZeiger[k];
    }

    return 0;
}
```

Übung

Was gibt dieses Programm aus? Mit dem Wissen aus den letzten Abenden solltest du ohne abtippen in der Lage sein, diese Übung zu lösen!

Ergänzung

Was im Beispiel mit dem *Auto* und dem *Cabrio* nur unschön aussieht, kann Konsequenzen haben, die unlösbar sein könnten. Ergänze die Klasse Cabrio wie folgt:

```
#ifndef CABRIO_H
#define CABRIO_H

#include "Auto.h"

class Cabrio : public Auto
{
public:
    // Konstruktor
    Cabrio();
    //Destruktor
    ~Cabrio();

    // Methoden:
    void ausgabe() const;

private:
    // Zeiger auf dynamisch allozierten
    // Speicher
    int* data;
};

#endif
```

Und erweitere den Konstruktor und den Destruktor:

```
Cabrio::Cabrio()
{
    // viel Speicher reservieren
    data = new int[100000];
    cout << "Ein Cabrio wurde erstellt" << endl;
}

Cabrio::~Cabrio()
{
    // Speicher freigeben
    delete [] data;
    cout << "Ein Cabrio wurde zerstoert" << endl;
}
```

Ein Programm mit dieser Cabrio Klasse könnte man nicht lange laufen lassen, denn es erzeugt gehörige Datenlecks.

Statische casts

Wir könnten unser main folgendermassen abändern um hier immerhin eine Lösung anzubieten:

```
    // und jeden löschen
```

```
    for(int k = 0; k < Anzahl; ++k)
    {
        if(0 == k % 2)
        {
            Auto* zuLoeschen = autoZeiger[k];
            delete zuLoeschen;
        }
        else
        {
            // wir weisen den Compiler mit dem statischen
            // cast an, den Auto* als Cabrio* zu interpretieren
            Cabrio* zuLoeschen =
static_cast<Cabrio*>(autoZeiger[k]);
            delete zuLoeschen;
        }
    }

    return 0;
}
```

Jetzt werden alle Objekte korrekt aufgeräumt. Der Methodenaufruf von ausgabe könnte man ebenfalls auf diese Art lösen. Es gibt jedoch eine viel bessere Art dieses Problem anzugehen.

Polymorphe Klassen und dynamische Bindung

Wir können dafür sorgen, dass nicht zur Kompilierzeit festgelegt wird, welche Methode eines Objektes einer vererbten Klasse aufgerufen wird, sondern immer die Methode der Klasse, von der das Objekt auch erzeugt wurde. Das bedeutet im Beispiel oben, dass obwohl wir einen Zeiger auf ein *Auto*-Objekt haben, die Methode der Klasse Cabrio aufgerufen wird.
Das Wort polymorph bedeutet vielgestaltig, im Sinne von „ich rufe die Methode *display* für ein *Auto*-Objekt auf, aber das Objekt hat die Gestalt oder das Verhalten eines Cabrios“.

Das Schlüsselwort *virtual*

Mit diesem Schlüsselwort teilen wir dem Compiler mit, dass er bei einer nicht direkt den Methodenaufruf erzeugt, sondern er erzeugt einen Tabelle mit dem Methodenaufrufen, die zur Laufzeit, also dynamisch mit den richtigen Methodenadressen gefüllt wird. Man spricht von der virtuellen Methodentabelle (VMT), oder auf englisch VTBL, sprich „witäibl“. Lies das Im Buch nach S. 575.
Das Schlüsselwort wird in der Basisklasse verwendet um anzuzeigen das eine Methode virtuell ist.

Klasse Auto mit virtueller Methode a*usgabe*

```
#ifndef AUTO_H
#define AUTO_H

class Auto
{
public:
    // Konstruktor
    Auto();
    //Destruktor
    ~Auto();

    // Methoden:
    virtual void ausgabe() const;
};

#endif
```

Das genügt bereits, so dass das Beispiel bei der Methode *ausgabe* immer die korrekte Methode aufruft. Bei einem Objekt der Klasse *Auto* wird also Auto::ausgabe() aufgerufen, bei einem Objekt der Klasse *Cabrio* Cabrio::ausgabe(), unabhängig davon, ob wir einen Zeiger vom Datentyp *Cabrio* haben oder nicht.
Es genügt das Schlüsselwort *virtual* in der Basisklasse anzuwenden, für die redefinierten Methoden in der abgeleiteten Klasse gilt das automatisch.

Virtueller Destruktor

Wir hatten das Problem mit dem Destruktor, der nicht aufgerufen wurde. Durch einen statischen cast, erzeugten wir aber aus einem *Auto** einen *Cabrio**, was aber in den meisten Fällen nicht möglich ist, denn man weiss im Allgemeinen nicht, ob das Objekt wirklich ein Objekt der Klasse *Cabrio* oder der Klasse *Auto* ist.

Genau wie bei der Methode a*usgabe* können wir den Destruktor virtuell machen:

```
#ifndef AUTO_H
#define AUTO_H

class Auto
{
public:
    // Konstruktor
    Auto();
    //Destruktor
    virtual ~Auto();

    // Methoden:
    virtual void ausgabe() const;
};

#endif
```

Dadurch können wir den Code im main wieder vereinfachen:

```
#include "Auto.h"
#include "Cabrio.h"

const int Anzahl = 6;

int main()
{
    Auto* autoZeiger[Anzahl] = { 0 };

    for(int i = 0; i < Anzahl; ++i)
    {
        if(0 == i % 2)
        {
            autoZeiger[i] = new Auto;
        }
        else
        {
            autoZeiger[i] = new Cabrio;
        }
    }

    for(int j = 0; j < Anzahl; ++j)
    {
        autoZeiger[j]->ausgabe();
    }

    for(int k = 0; k < Anzahl; ++k)
    {
        delete autoZeiger[k];
    }

    return 0;
}
```

Übung „Bewegliche, farbige Objekte“

Bei dieser Übung geht es darum Objekte, die irgendwie etwas gemeinsam haben, in einer Basisklasse zusammenzufassen. Nehmen wir als Beispiel ein Zeichenprogramm, das viele verschiedene geometrische Objekte verwaltet wie Punkte, Linien, Dreiecke und so weiter. Wenn das Programm diese Objekte ausgeben muss, ruft es für alle eine Methode zur Darstellung auf. Das könnte zum Beispiel eine Methode *paint* oder *draw* sein. Im Normallfall haben alle diese Klasse eine gemeinsame Basisklasse, zum Beispiel *GeoObject* und diese Basisklasse hat eine virtuelle Methode *draw*. Das Programm muss diese Objekte nur noch in einem Array von *GeoObject** verwalten und kann mit einer einfachen Schleife alle Objekte ausgeben. Wir machen so etwas ähnliches mit unseren Zeichnungsobjekten aus dem TsuZeichnen.
Erzeuge ein neues Projekt und füge die Klassen für die grafische Ausgabe (TsuZeichnen, etc.) hinzu. Die verschiedenen Klassen für die Objekte, die wir erzeugen können, haben alle gemeinsam, dass man ihre Farbe ändern kann, oder ihre Koordinaten ändern kann. Definiere also eine neue Basisklasse *TsuZeichnenObjekt*, die zwei virtuelle Methoden hat, nämlich *bewege* und *setzeFarbe*. Definiere auch einen virtuellen Destruktor.
Nun leitest du für die verschiedenen Klassen, wie *Text, Kreis usw.* jeweils eine Klasse von *TsuZeichnenObjekt* ab. Zum Beispiel eine Klasse *TsuZeichnenText*. In dieser Klasse redefinierst du die beiden virtuellen Methoden und fügst als privates Datenelement ein Objekt der Klasse *Text* hinzu. Schreibe auch eine Methode *getText*. Hier ein Beispiel:

```
class TsuZeichnenText : public TsuZeichnenObjekt
{
    public:
        TsuZeichnenText(long x, long y);

        Text& getText()
        {
            return m_text;
        }

        // virtuelle Funktion der Basisklasse
        // TsuZeichnenObjekt redefinieren
        void setzeFarbe(COLORREF farbe);

    private:
        Text m_text;
};
```

Beginne zuerst nur mit der Methode *setzeFarbe*. Die *bewege* Methode braucht mehr Aufwand und kann auf das nächste mal in einem zweiten Schritt programmiert werden.

Schreibe danach ein *main*, das etwa so aussieht:

```
#include <windows.h> // für Sleep
#include „TsuZeichnenText.h“

// weitere includes

int main()
{
    ComSystem dasComSystem;

    // Hier zuerst ZeichenFläche erstellen
    ....

    // zehn Zeiger auf TsuZeichnenObjekte
    TsuZeichnenObjekt* objekte[10] = { 0 };

    // Schleife, bei der der Benutzer wählen kann
    // was er für Objekte erstellen will
    ....

    // Hier kommt eine Schleife,
    // die alle Sekunden die Farben der
    // Objekte ändert
    for(int i = 0; i < 20; ++i)
    {
        Sleep(1000); // eine Sekunde schlafen
        COLORREF farbe = RGB(10*i, 10*i, 10*i);
        for(int index = 0; index < 10; ++index)
        {
            objekte[index]->setzeFarbe(farbe);
        }
    }

    // Schleife um alle Objekte zu löschen
    ....

    return 0;
}
```

Löse diese Aufgabe schrittweise. Beginne mit der Basisklasse *TsuZeichnenObjekt*. Leite dann als erstes die Klasse *TsuZeichnenText* davon ab. Schreibe dann das main, so dass der Benutzer zehn Objekte definieren kann (in einem ersten Schritt kann der Benutzer nur Textobjekte erzeugen), die auf dem Bildschirm angezeigt werden. Danach sollten diese Objekte auf der Zeichenfläche erscheinen und mit dem Beispielcode oben sollten sie ihre Farben im Sekundentakt ändern. Schreibe nun auch die Klassen für die Linien und die Kreise.
Die *bewege*-Methode definieren wir das nächste mal.