

15. Abend, Teilobjekte und statische Elemente

Ziel :

- Wir können ab heute Abend Objekte erstellen, die aus anderen Objekten bestehen und diese als Teilobjekte enthalten. Wir sind auch in der Lage Datenelemente zu einer Klasse zu erstellen, die nur einmal im Speicher sind.

Teilobjekte und deren Initialisierung

Gewisse Objekte, die wir in C++ abstrahieren wollen bestehen aus gewissen anderen und enthalten diese. Nehmen wir an wir schreiben eine Renn-Simulation in der Autos vorkommen. Nehmen wir weiter an, dass ein Programmierer sich nur um die Räder eines Autos kümmert um grösstmögliche Realität zu erreichen. Ein anderer Programmierer beschäftigt sich dann mit dem Auto als Ganzes. Ein Auto hat vier Räder, man spricht von einer Hat-Beziehung. Das könnte ungefähr so aussehen :

```
class Rad
{
    public:
        // Konstruktor
        Rad();
        // Destruktor
        ~Rad();

        // weitere Methoden
};

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

        // weitere Methoden

    private:
        // Teilobjekte
        Rad    m_linkesVorderrad;
        Rad    m_rechtesVorderrad;
        Rad    m_linkesHinterrad;
        Rad    m_rechtesHinterrad;
};
```

Ein Auto besteht also aus vier Rädern.

Angenommen die Klasse Rad hat nur einen speziellen Konstruktor, der als Parameter den Radius des Rades hat :

```
class Rad
{
    public:
        // Konstruktor mit Parametern
        Rad(long radius);

    private:
        long    m_Radius;
};
```

Ein Konstruktor mit Parametern führt dazu, dass der Compiler keinen Konstruktor ohne Parameter mehr zur Verfügung stellt (ausser er wird explizit definiert). Der Programmierer der Klasse Rad wollte aber wohl nicht, dass jemand den Defaultkonstruktor verwendet.

Da beim Erzeugen eines Objektes der Klasse Auto auch alle Teilobjekte - also auch alle Räder - mit dem default-Konstruktor erzeugt werden wollen, wird die Klasse Auto nicht mehr compilieren. Um Teilobjekte mit Parametern zu initialisieren gibt es die Möglichkeit eine Initialisierungsliste zu verwenden. Hier der Konstruktor der Klasse Auto mit Initialisierungsliste :

```
Auto::Auto()
    :m_linkesVorderrad(4),
      m_rechtesVorderrad(4),
      m_linkesHinterrad(4),
      m_rechtesHinterrad(4)
{
}
```

Vor der {}-Klammer des Konstruktors kommt ein : (Doppelpunkt) und danach die Datenelement mit dem speziellen Konstruktorparameter. Auch andere Datenelemente können mittels Initialisierungsliste erzeugt werden. Hier als Beispiel der Konstruktor der Klasse Rad :

```
Rad::Rad(long radius)
    :m_radius(radius)
{
}
```

Der Konstruktor könnte aber auch so aussehen :

```
Rad::Rad(long radius)
{
    m_radius = radius;
}
```

Der Konstruktor von Auto muss aber die Initialisierungsliste verwenden, denn die Rad hat nur einen Konstruktor mit Parametern.
Konstanten müssen auf die gleiche Art initialisiert werden !

```
class Rad
{
public:
    // Konstruktor mit Parametern
    Rad(long radius);

private:
    long m_radius;
    const double m_pi;
};

Rad::Rad(long radius)
    :m_pi(3.1415),
    m_radius(radius)
{
}
```

Statische Datenelemente

Die Eigenschaften eines jeden Objektes sind in seinen Datenelementen gespeichert. Ein Objekt unterscheidet sich von einem anderen durch seine Datenelemente. Es gibt manchmal aber auch Daten, die nicht nur zu einem Objekt gehören, sondern nur einmal für die ganze Klasse existieren müssen.

Ein Beispiel, das man häufig antrifft, ist wenn man die Anzahl der Objekte einer bestimmten Klasse zählen will. Man verwendet hierfür statische Variablen, die mit dem Schlüsselwort `static` definiert werden.

```
class Rad
{
    public:
        // Konstruktor mit Parametern
        Rad(long radius);

    private:
        long          m_radius;
        const double m_pi;
        // diese Zahl gibt es
        // nur einmal, unabhängig
        // der Anzahl Räder die
        // erzeugt werden
        static long  s_anzahlRaeder;
};

// Initialisierung der statischen Variablen
long Rad::s_anzahlRaeder = 0;

Rad::Rad(long radius)
    :m_pi(3.1415),
    m_radius(radius)
{
    // Gesamtanzahl der Räder
    // erhöhen
    s_anzahlRaeder++;
}
```

Erzeugen wir ein Objekt der Klasse Rad, werden die Datenelemente die zu diesem Objekt gehören automatisch im Speicher erzeugt, das heisst wenn wir ein Rad erzeugen wird im Speicher auch gleich ein long für m_radius und ein double für m_pi erzeugt.

Da s_anzahlRaeder aber nicht direkt zu einem einzelnen Objekt gehört, muss diese statische Variable speziell initialisiert werden (siehe oben).

Übrigens wäre es für m_pi auch nicht nötig, dass jedes Objekt ein eigenes Datenelement m_pi besitzt. Man könnte also pi auch als statische Variable von Rad definieren !

Statische Elementfunktionen

Da ein statisches Datenelement existieren kann ohne, dass es ein Objekt gibt kann man Elementfunktionen (Methoden) definieren, die den Zugriff auf solche Datenelemente ermöglichen. Die Methoden werden auch statisch definiert und können so unabhängig von einem Objekt aufgerufen werden:

```
class Rad
{
    public:
        // Konstruktor mit Parametern
        Rad(long radius);

        static long holeAnzahlRaeder();

    private:
        long          m_radius;
        const double m_pi;
        // diese Zahl gibt es
        // nur einmal, unabhängig
        // der Anzahl Räder die
        // erzeugt werden
        static long  s_anzahlRaeder;
};

long Rad::s_anzahlRaeder = 0;

long Rad::holeAnzahlRaeder()
{
    return s_anzahlRaeder;
}

int main()
{
    long radius = 6;

    Rad einRad(radius);
    Rad einZweitesRad(radius);

    // statische Methode aufrufen
    long anzahl = Rad::holeAnzahlRaeder();
    // oder wenn es ein Objekt gibt
    anzahl = einRad.holeAnzahlRaeder();

    return 0;
}
```

Klassenspezifische Konstanten, enum

Mit enum kann ganz einfach eine Aufzählung definiert werden.

```
enum AmpelFarbe
{
    rot,
    gelb,
    gruen
};

int main()
{
    AmpelFarbe farbe;
    farbe = gelb;
    farbe = rot;
    farbe = gruen;

    int test = -1;

    // solche enums können einer
    // Ganzzahl zugewiesen werden
    test = rot;
    test = gelb;
    test = gruen;

    return 0;
}
```

Der erste Wert in einer Aufzählung bekommt den numerischen Wert 0, die folgenden immer den um eins erhöhten des Vorgängers, hier also gelb = 1 und gruen = 2. Diese Werte können aber auch beeinflusst werden :

```
enum AmpelFarbe
{
    rot = 10,
    gelb,
    gruen = 15
};
```

"rot" hat jetzt den Wert 10, gelb den Wert 11 und gruen den Wert 15. Ein solcher enum kann auch in einer Klassen definiert werden und gehört dadurch zur Klasse. Hier ein komplettes Beispiel dazu :

```
class Ampel
{
    public:
        enum AmpelFarbe
        {
            rot,
            gelb,
            gruen
        };

        AmpelFarbe getFarbe() const
        {
            return m_farbe;
        }

    private:
        AmpelFarbe m_farbe;
};

int main()
{
    Ampel eineAmpel;

    Ampel::AmpelFarbe farbe = eineAmpel.getFarbe();

    switch(farbe)
    {
        case Ampel::rot:
            // nicht fahren
            break;
        case Ampel::gelb:
            // na ja
            break;
        case Ampel::gruen:
            // fahren
            break;
        default:
            // gibts doch gar nicht
            break;
    }

    return 0;
}
```