

11. Abend, Klassen

Ziel :

- Mit C++ ist es möglich Problemstellungen zu abstrahieren und vereinfachte Modelle der Wirklichkeit abzubilden. Wir lernen mit Hilfe von Klassen (class) Dinge zu beschreiben und in C++ damit zu arbeiten.

Datenabstraktion

Dinge des alltäglichen Lebens sind, wenn man sie genau betrachtet nicht ganz einfach in ihrer Gesamtheit zu beschreiben. Eine CD zum Beispiel ist oberflächlich betrachtet eine silbige Scheibe - mit Musik darauf. Genau gesehen ist es aber eine mehrschichtige Scheibe mit reflektierenden und nicht reflektierenden Schichten, mit Pits, die in die eine Schicht eingelassen sind. Diese Pits beschreiben ein Bitmuster, das einem bestimmten Protokoll gehorcht und so weiter. Es gibt vermutlich Menschen, die Bücher darüber schreiben könnten.

Solche komplexen Dinge vereinfachen wir Menschen im allgemeinen ganz automatisch und reduzieren sie auf das für uns wesentliche. Abstraktion bedeutet genau das : Das für eine bestimmte Aufgabe Wesentliche herauszufiltern und zu vereinfachen. Dabei ist die Abstraktion eines bestimmten Artikels je nach Problemstellung verschieden. Für den Hersteller einer CD ist möglicherweise wichtig weiviele Tracks darauf sind, wie dicht die Pits sind und was die reflektierende Folie für eine Farbe haben soll. Für einen Musikliebhaber ist hingegen wichtig von wem die Musik auf der CD ist, was die Titel für einen Namen tragen und wie lang die CD ist.

Bei der Analyse eines Problems, das in C++ gelöst werden muss versuchen wir eben diese Abstraktion passend durchzuführen. Dies führt zu einer "Beschreibung", eine Klasse. Eine Klasse ist eine Beschreibung, ein Muster nach dem zur Laufzeit eines Programms Objekte erzeugt werden können. Eine C++ Klasse ist ein selbstdefinierter Datentyp (im Gegensatz zu den eingebauten Datentypen wie **long**, **double**, etc.). Eine Klasse besteht aus Datenelementen, die Eigenschaften eines Objektes beschreiben und Elementfunktionen (=Methoden), die die Fähigkeiten eines Objektes zeigen.

Datenkapselung

Gewisse Elemente einer Klasse sind "privat", sie sind von Benutzern der Klasse nicht direkt zugänglich. Anwendungen von Klassen arbeiten mit der "öffentlichen" Schnittstelle einer Klasse. Um Eigenschaften von Objekten einer Klasse abzufragen werden im allgemeinen Methoden aus der öffentlichen Schnittstelle der Klasse verwendet. Dem Benutzer bleibt dadurch der interne Aufbau des Objektes verborgen. Wird dieser interne Aufbau eines Objektes später verändert, das heisst die Datenelemente verändert, stört das den Anwender der Klasse nicht, solange der öffentliche Teil einer Klasse nicht verändert wird.

Beispiele

Genug der leeren Worte, hier einige Beispiele, die diese Prinzipien ein wenig beleuchten sollen. Ein Musikfan will ein Programm schreiben mit dem er seine CD's verwalten kann. Er überlegt, was für ihn wichtig ist an einer CD und kommt zum folgenden Schluss : Zu einer CD gehört ein Interpret, ein Titel und die Länge der CD. Das führt zu folgender Klassendefinition :

```
#ifndef CD_H // mehrfaches includieren
#define CD_H // verhindern

#include <string>

// wir verwenden in dieser
// Datei nur die string-Klasse
using std::string;

// Schlüsselwort class gefolgt
// vom Klassennamen
class CD
{
    // öffentliche Schnittstelle der Klasse
    // CD
    public:
        // mit folgenden Funktionen
        // können wir die Datenelemente
        // jeweils setzen
        void setInterpret(const string& interpret);
        void setTitle(const string& title);
        void setLength(long length);

        // diese Funktion gibt die
        // Daten der CD aus
        void display();

    // Datenelemente im private-Teil verstecken
    private:
        string m_interpret;
        string m_title;
        long m_length;
}; // wichtig ! Semikolon nicht vergessen

#endif
```

Diese Klassendefinition wird im allgemeinen in einer Header-Datei gespeichert (hier CD.h). Die Headerdatei beginnt mit einem sogenannten "Include-Blocker", der am Beginn der Datei steht, wobei das **#endif** am Ende auch dazugehört. Auf diese Art wird verhindert, dass die Datei mehrfach mit **#include** eingebunden werden kann, was zu Compiler-Fehlern führen kann !

Wie ihr seht hat der Entwickler beschlossen, dass die einzelnen Datenelemente mittels dreier Funktionen jeweils gesetzt werden können. Zusätzlich hat er eine Funktion "display" vorgesehen, mit der die CD sich auf dem Bildschirm ausgeben kann.

Beachte übrigens, dass die **string**'s jeweil als konstante Referenz übergeben werden. Seit der letzten Lektion weisst Du auch was das bedeutet, sonst liest Du am besten nach !

Die Funktionen, müssen jetzt noch definiert das heisst ausprogrammiert werden. Dies machen wir in diesem Fall in der Datei CD.cpp :

```
#include "CD.h"

#include <iostream>
using namespace std;

/////////////////////////////////////////////////////////////////

void CD::setInterpret(const string& interpret)
{
    m_interpret = interpret;
}

/////////////////////////////////////////////////////////////////

void CD::setLength(long length)
{
    m_length = length;
}

/////////////////////////////////////////////////////////////////

void CD::setTitle(const string& title)
{
    m_title = title;
}

/////////////////////////////////////////////////////////////////
```

```
void CD::display()
{
    cout << "-----" << endl;
    cout << "Title      : " << m_title << endl;
    cout << "Interpret  : " << m_interpret << endl;
    cout << "Länge      : " << m_length << endl;
}
```

```
////////////////////////////////////
```

Eine andere Abstraktion hätte zu einem völlig anderen Ergebnis führen können. Ein Ingenieur bei einem CD-Presswerk wäre möglicherweise zu folgender Abstraktion, also Klasse gekommen :

```
class CD
{
public:
    void init(double weight, double thickness,
              string color, string material);
private:
    double m_weight;
    double m_thickness;
    string m_color;
    string m_material;
};
```

Obwohl diese Klasse nicht viel mit der CD Klasse von oben gemeinsam hat, ist es eine absolut richtige Abstraktion einer CD, aber für einen völlig anderen Problem-bereich.

Zurück zum Beispiel von oben. Schreiben wir nun also eine kleine Anwendung, die ein Objekt der Klasse CD erzeugt und diese verwendet. Diese Anwendung wird nur eine kleine main-Funktion sein. Diese Funktion ist sozusagen ein "Client" der Klasse CD und kann nur auf Dinge zugreifen, die im öffentlichen Teil der Klasse definiert sind.

```
// Wir wollen diese
// tolle CD Klasse verwenden
#include "CD.h"

#include <string>

using std::string;

int main()
{
    CD eineCD;
    // eine Variabe/Objekt erzeugen
```

```
string elvis("Elvis Presley");
string titel("Viva Las Vegas");

// Elementfunktionen aufrufen
// um die Daten einzutragen
eineCD.setInterpret(elvis);
eineCD.setTitle(titel);
// Länge in sekunden
eineCD.setLength(2130);

eineCD.display();

return 0;
}
```

CD ist ein selbstdefinierter Datentyp. Um eine Variable zu erzeugen können wir vorgehen wie bis anhin und schreiben:

```
Datentyp VariablenName;
also

CD eineCD;
```

Unsere Klasse CD verwaltet im Moment die Länge als **long**, als die Anzahl Sekunden, die die CD dauert. Der Programmierer der Klasse CD kann jetzt aber beschliessen die Länge der CD als **double** Wert zu verwalten mit Minuten und Sekunden als Nachkommastellen. Die interne Darstellung das heisst die privaten Datenelemente werden also geändert :

```
class CD
{
    // öffentliche Schnittstelle der Klasse
    // CD
    public:
        // mit folgenden Funktionen
        // können wir die Datenelemente
        // jeweils setzen
        void setInterpret(const string& interpret);
        void setTitle(const string& title);
        void setLength(long length);

        // diese Funktion gibt die
        // Daten der CD aus
        void display();

    // Datenelemente im private-Teil verstecken
    private:
        string m_interpret;
        string m_title;
        double m_length;
}; // wichtig ! Semikolon nicht vergessen
```

Übung

Schreibe die Methoden "setLength" und "display" der Klasse CD so um, dass m_length nach dem Aufruf der Funktion setLength die Länge als Minute.Sekunden speichert und auch so ausgibt :

```
-----
Title      : Viva Las Vegas
Interpret  : Elvis Presley
Laenge     : 35.30
```

Um in der Funktion "display" die Ausgabe für double Zahlen so einzurichten, dass sie zwei Nachkommastellen ausgibt, musst du die Datei <iomanip> includieren. Du kannst dann folgenden Aufruf machen :

```
cout << fixed << setprecision(2);
```

Siehe hierfür auch im Buch das Beispiel auf Seite 268 an.

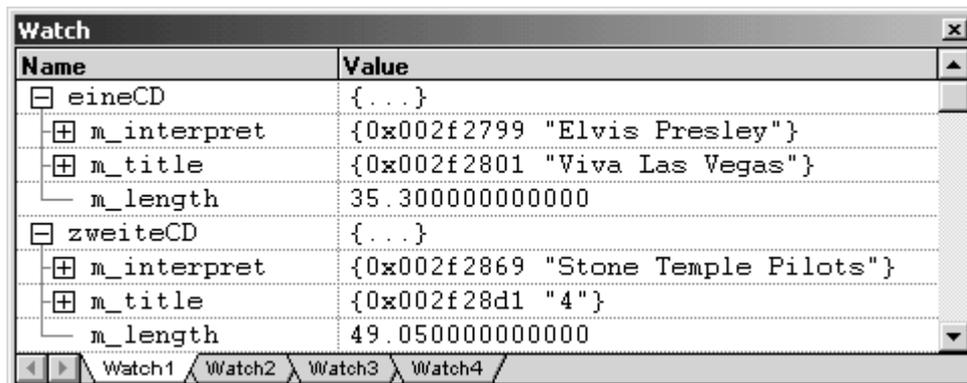
Objekte im Speicher

Wenn wir mehrere Objekte also Variablen vom Datentyp CD erzeugen, hat jedes Objekt einen eigenen Satz an Datenelementen. Folgendes main-Programm erzeugt zwei Objekte der Klasse CD. Jedes hat für sich seine eigenen Datenelemente, was man mit dem Debugger leicht überprüfen kann.

```
int main()
{
    CD eineCD;
    eineCD.setInterpret("Elvis Presley");
    eineCD.setTitle("Viva Las Vegas");
    eineCD.setLength(2130);

    CD zweiteCD;
    zweiteCD.setInterpret("Stone Temple Pilots");
    zweiteCD.setTitle("4");
    zweiteCD.setLength(2945);

    return 0;
}
```



The screenshot shows a debugger's Watch window with two tabs: Watch1 and Watch2. The Watch1 tab is active and displays the following data:

Name	Value
[-] eineCD	{...}
[+] m_interpret	{0x002f2799 "Elvis Presley"}
[+] m_title	{0x002f2801 "Viva Las Vegas"}
[-] m_length	35.30000000000000
[-] zweiteCD	{...}
[+] m_interpret	{0x002f2869 "Stone Temple Pilots"}
[+] m_title	{0x002f28d1 "4"}
[-] m_length	49.05000000000000

Beide Objekte werden in diesem Beispiel nicht explizit initialisiert, wie wir und das von anderen Datentypen gewohnt sind, z.B. bei einem **long** :

```
long value = 0;
```

Bei Klassen geschieht dies durch einen sogenannten Konstruktor. Wir werden dies aber erst später genauer ansehen und kennenlernen.

Zugriff auf Objekte

Wie man in unserer main-Funktion bestens sieht, kann man mittels des **Punktoperators** auf Elemente eines Objektes zugreifen. Das gilt für Datenelemente genauso wie für Elementfunktionen (Methoden) :

```
objekt.element
```

Bei unserer Klasse CD könnte auch der Wunsch bestehen auf die privaten Datenelemente zuzugreifen :

```
int main()
{
    CD eineCD;
    eineCD.setInterpret("Elvis Presley");
    eineCD.setTitle("Viva Las Vegas");
    eineCD.setLength(2130);

    CD zweiteCD;
    zweiteCD.setInterpret("Stone Temple Pilots");
    zweiteCD.setTitle("4");
    zweiteCD.setLength(2945);

    cout << zweiteCD.m_length << endl; // Fehler !!!

    return 0;
}
```

Der Compiler lässt es nicht zu, dass ausserhalb der Klasse auf private Datenelemente zugegriffen wird und erzeugt eine Fehlermeldung. Das ist hier auch absolut erwünscht, denn gerade `m_length` haben wir ja während der Entwicklung der Klasse geändert. Hätten wir das Datenelement `m_length` in der Klasse CD im public-Teil der Klasse definiert, hätte der Code hier zwar compiliert, aber der Anwender der Klasse CD wäre **abhängig** davon, dass `m_length` immer gleich definiert bleibt.

Solche **Abhängigkeiten** sind **schlecht**. Von jetzt an merkt ihr euch, dass in C++ Abhängigkeiten Teufelszeug ist, Amen.

Genauso wie ein long einem anderen long zugewiesen werden kann :

```
long a = 10;
long b = a;
```

Können wir einem Objekt der Klasse CD ein anderes Objekt der Klasse CD zuweisen :

```
int main()
{
    CD eineCD;
    eineCD.setInterpret("Elvis Presley");
    eineCD.setTitle("Viva Las Vegas");
    eineCD.setLength(2130);

    CD copy = eineCD;

    return 0;
}
```

Was dabei zu beachten ist, ist dass "**copy**" ein eigenes Objekt ist, also selber Speicherplatz beansprucht und vom Objekt "eineCD" unabhängig ist. Ich erinnere an die letzte Lektion wo wir gelernt haben, wann Daten kopiert werden und wie man mit Referenzen und Zeigern umgehen kann. Wir können nämlich auch hier eine Kopie verhindern, so wie wir es in der letzten Lektion gelernt haben :

```
// Ref ist nun eine
// Referenz auf das
// originale Objekt !
CD& ref = eineCD;
```

Hier ist ref also nur eine anderer Name für ein Objekt, das sich bereits im Speicher befunden hat ! Im Gegensatz zum "copy"-Beispiel von vorhin wo copy ein eigenständiges Objekt im Speicher ist. Genauso sind auch Zeiger möglich. Da ein solches Objekt auch irgendwo im Speicher unserer Rechners belegt ist es auch möglich diese Adresse zu finden und als Zeiger zu speichern :

```
// finde die Adresse
// von eineCD !
CD* pEineCD = &eineCD;
```

Zeiger auf CD : **CD***.

Adresse von einem CD Objekt : address-of operator **&**
Zugriff über einen Zeiger

Wenn wir einen Zeiger auf ein Objekt haben können wir nicht mehr mit dem Punktoperator auf Elemente des Objektes zugreifen. Wir müssen zuerst dereferenzieren !

```
int main()
{
    CD eineCD;
    eineCD.setInterpret("Elvis Presley");
    eineCD.setTitle("Viva Las Vegas");
    eineCD.setLength(2130);

    // finde die Adresse
    // von eineCD !
    CD* pEineCD = &eineCD;

    // dereferenzieren und
    // danach den Punktoperator !
    (*pEineCD).display();

    return 0;
}
```

Diese Schreibweise ist nicht ganz schön und leserlich. Da aber Zeiger in C++ doch recht häufig verwendet werden und man oft nur einen Zeiger auf gewisse Objekte hat, hat man eine andere Schreibweise erfunden : den **Pfeiloperator** Das Beispiel von oben kann dann auch so geschrieben werden:

```
    // finde die Adresse
    // von eineCD !
    CD* pEineCD = &eineCD;

    // Pfeiloperator !
    pEineCD->display();
```

struct

In unserer Klasse CD haben wir die Daten, die zu einer CD gehören zusammengefasst. Zusätzlich haben wir aber der CD auch ein gewisses Verhalten gegeben, indem wir Elementfunktionen in der Klasse eingefügt haben. In C kannte man im allgemeinen nur das Zusammenfassen von Daten, die logisch zueinander gehörten. Man spricht auch von Datensätzen. Solche Datensätze können mit dem Schlüsselwort **struct** definiert werden.

```
struct Length
{
    long Minutes;
    long Seconds;
};
```

Ein struct entspricht einer Klasse mit einem kleinen Unterschied. Die Datenelemente in einem struct sind automatisch **public** ! Das heisst folgende Klasse Length ist genau gleich wie die Struktur Length von oben :

```
class Length
{
public:
    long Minutes;
    long Seconds;
};
```

Hier ein kleines main-Beispiel :

```
struct Length
{
    long Minutes;
    long Seconds;
};

int main()
{
    Length theLength;

    theLength.Minutes = 34;
    theLength.Seconds = 30;
    // da die Elemente automatisch
    // public sind, ist der Zugriff
    // erlaubt

    return 0;
}
```

Structs werden manchmal von früheren C-Programmierern eingesetzt wenn man zeigen will, dass es sich wirklich nur um einige zusammengehörende Daten handelt, die kein Verhalten haben.

Übung

1.

Ändere die Klasse CD so, dass sie anstatt des **doubles** oder **longs** für die Länge eine Struktur CDLength verwendet. Definiere in der Datei CD.h eine Struktur CDLength mit den Elementen Minutes und Seconds. Ändere danach den Datentypen des Datenelementes **m_length** von double auf CDLength. Schreibe nun die Funktion "setLength(long length)" so um, dass sie die Minuten und Sekunden berechnet und im Datenelement m_length speichert. Ändere nun auch die Funktion "display" so, dass sie die Elemente der Struktur m_length einzeln ausgibt, etwa so :

```
-----  
Title       : Viva Las Vegas  
Interpret   : Elvis Presley  
Laenge      : 35 min 30 sec
```

2.

Du bist ein junger Mensch und interessierst dich möglicherweise für Autos. Möglicherweise vergleichst Du manchmal die Fahrzeuge Deiner Kollegen mit Deinem. Nimm an Du willst eine Datenbank haben von allen Autos Deiner Kollegen.

Schreibe eine Klasse Auto.

Abstrahiere das ein Auto so, wie es Dir für die geschilderte Problemstellung richtig erscheint. Füge also einige Datenelemente in Deine Klasse ein und gib der Klasse auch ein Verhalten indem du einige Methoden definierst, sicher um die Datenelemente zu setzen wie setPS(long ps) etc. Schreibe auch eine Methode display() um die Daten anzuzeigen.

Definiere die Klasse in einer eigenen Header und Quellcodedatei (z.B. Auto.h, Auto.cpp).

Schreibe eine Methode "staerker" : bool staerker(Auto* anderesAuto) in der Klasse Auto. Diese Funktion soll die PS Zahl (m_PS) mit der PS-Zahl des anderen Autos vergleichen und true zurückgeben falls die eigene PS-Zahl grösser ist als die des anderen Autos.