12. Abend, Methoden

Ziel :

- Wir lernen heute Objekte von Klassen zu initialisieren wenn wir sie erzeugen, und wie wir aufräumen wenn ein Objekt abgebaut/zerstört wird.

Konstruktoren

Um ein Objekt von einer Klasse zu initialisieren, wenn es erzeugt wird, bietet C++ die Möglichkeit einen Konstruktor zu definieren. Als erstes lernen wir den **default-Konstruktor** kennen. Der default-Konstruktor ist ein Konstruktor, der keine Parameter hat. (Weiter unten sehen wir Konstruktoren, die Parameter haben)

CD.H:

```
#ifndef CD_H
#define CD_H

#include <string>

using std::string;

class CD
{
    public:
        // Default Konstruktor
        CD();

        void setzeTitel(const string& titel);
        void setzeInterpret(const string& interpret);
        void setzeLaenge(long sekunden);

    private:
        string m_Titel;
        string m_Interpret;
        long   m_Laenge;

};

#endif
```

CD.CPP

```
#include "CD.h"


// Default Konstruktor
CD::CD()
{
    m_Laenge = 0;
    // m_Titel und m_Interpret
    // müssen nicht initialisiert
    // werden, da sie selber Objekte
    // sind und einen eigenen
    // Konstruktor haben
}

void CD::setzeInterpret(const string& interpret)
{
    m_Interpret = interpret;
}

void CD::setzeTitel(const string& titel)
{
    m_Titel = titel;
}

void CD::setzeLaenge(long sekunden)
{
    m_Laenge = sekunden;
}
```

Hier noch ein Beispiel mit einer Klasse Complex. Eine Klasse, die eine komplexe Zahl kapselt.

```
class Complex
{
    public:
        double GetReal();
        double GetImag();

        double GetAbsolute();


    private:
        double m_Real;
        double m_Imag;
};
```

Ein **Objekt** (eine Variable) dieser Klasse besitzt zwei Datenelemente : m_Real und m_Imag. Um auf die Elemente zugreifen zu können gibt es die beiden **Methoden** GetReal() und GetImag(). Zusätzlich gibt es noch die **Methode** GetAbsolute(), welche den Absolutewert der Komplexen Zahl zurückgeben soll. Ein vollständiges Beispiel sieht so aus :

```
#include <iostream>
#include <string>
#include <cmath>

using namespace std;

class Complex
{
    public:

        double GetReal();
        double GetImag();

        double GetAbsolute();

    private:
        double m_Real;
        double m_Imag;
};


double Complex::GetReal()
{
    return m_Real;
}

double Complex::GetImag()
{
    return m_Imag;
}

double Complex::GetAbsolute()
{
    double Absolute = sqrt(m_Real * m_Real + m_Imag * m_Imag);
    return Absolute;
}
```

Hier noch ein passendes main :

```
int main()
{
    Complex c;

    double Real = c.GetReal();

    return 0;
}
```

In der **main**-Funktion erzeugen wir ein Objekt der Klasse Complex mit dem Namen "c". Danach rufen wir die Funktion GetReal von diesem Objekt c auf. Der Wert den wir zurückerhalten kann irgend ein Wert sein, denn das Datenelement m_Real von c hat nie einen Wert erhalten ! Eigentlich müssten wir die Datenelemente von c irgendwie initialisieren. Um Datenelemente zu initialisieren gibt es in C++ den sogenannten **Konstruktor**.

```
class Complex
{
    public:
        // Default-Konstruktor
        Complex();

        double GetReal();
        double GetImag();

        double GetAbsolute();

    private:
        double m_Real;
        double m_Imag;
};
```

Hier der Code, der in der cpp - Datei steht:

```
Complex::Complex()
{
    m_Real = 0.0;
    m_Imag = 0.0;
}
```

Der Konstruktor sieht aus wie eine Funktion (wegen den Klammern) hat aber keinen Rückgabewert. Zuätzlich muss der Konstruktor immer so wie die Klasse heissen, hier also Complex. Dass der Konstruktor wie eine Funktion ist, sieht man auch daran, dass im Konstruktor ganz normaler Code steht. Dieser Konstruktor wird jedesmal

aufgerufen, wenn wir ein Objekt der Klasse Complex erzeugen, das heisst jedesmal wenn im Code eine Variable vom Datentyp Complex erzeugt wird !

Destruktor

Genauso wie wir mittels eines Konstruktors genau definieren können was geschieht, wenn eine Variable von einer selbstdefinierten Klasse erzeugt wird, können wir wenn Code ausführen lassen, wenn unsere Variable zerstört wird. Diese Funktion heisst **Destruktor**. Der Destruktor sieht beinahe aus wie ein Konstruktor, nur dass vor dem Klassennamen eine ~ (Tilde) steht.

```
class Complex
{
    public:
        Complex();
        Complex(double Real, double Imag);
        Complex(const Complex& c);

        ~Complex(); // Destruktor

        double GetReal();
        double GetImag();

        double GetAbsolute();

    private:
        double m_Real;
        double m_Imag;
};
```

Hier der Code für die .cpp - Datei

```
Complex::~Complex()
{
    cout << "Destruktor" << endl;
}
```

Der Desktruktor dient dazu, gewisse Dinge "aufzuräumen". In unserer Klasse Complex gibt es nichts aufzuräumen, aber man könnte sich vorstellen, dass in in einem Programm, das mit Dateien arbeitet in einem Destruktor die Datei wieder geschlossen werden kann. Wichtig wird der Destruktor, wenn wir lernen Speicher dynamisch zur Laufzeit anzufordern. Dynamisch allozierter (angeforderter) Speicher wird häufig im Destruktor wieder freigegeben.
Wann wird der Destruktor aufgerufen ?
Der Destruktor wird dann aufgerufen wenn die Variable nicht mehr gültig ist.

Konstruktor mit Parametern

Die Ähnlichkeit des Konstruktors mit einer Funktion geht noch weiter ! Es ist nämlich möglich einen Konstruktor mit Parametern zu schreiben. Betrachten wir noch einmal die Klasse komplex :

```
class Complex
{
    public:         // Default Konstruktor
        Complex();        // Konstruktor mit Parametern
        Complex(double Real, double Imag);
        double GetReal();
        double GetImag();

        double GetAbsolute();

    private:
        double m_Real;
        double m_Imag;
};
```

Der Code aus der .cpp-Datei

```
// Standardkontruktor
Complex::Complex()
{
    m_Real = 0.0;
    m_Imag = 0.0;
}

// Konstruktor mit Parametern
Complex::Complex(double Real, double Imag)
{
    m_Real = Real;
    m_Imag = Imag;
}
```

Auch dieser neue Konstruktor heisst genau wie die Klasse, verlangt aber im Gegensatz zum Standardkontruktor zwei Parameter. Im Code sieht man, dass diese beiden Parameter verwendet werden um die beiden Datenelemente zu initialisieren. Sehen wir uns ein wenig Code an, der eine Variable **c** vom Datentyp Complex mit dem Standardkontruktor erstellt und eine weitere Variable **z** mit dem zweiten Konstruktor :

```
int main()
{
    Complex c;  // Standardkontruktor
    Complex z(1.0, 2.0); // Konstruktor mit Parametern


    double Realc = c.GetReal(); // Realc wird den Wert 0.0 haben
    double Realz = z.GetReal(); // Realz wird den Wert 1.0 haben


    return 0;
}
```

Es ist möglich einen Haltepunkt auf die Zeilen zu setzen in denen die Variablen jeweils definiert werden, was bei einem Datentyp wie **int** im Normalfall nicht möglich ist. Versuche also einen Haltepunkt auf die Zeile in der **z** erzeugt wird zu setzen und spring dann mit F11 in den Konstruktor der Klasse Complex !

Kopierkonstruktor

Der Kopierkontruktor ist ein spezieller Konstruktor mit Parametern ! Das besondere an ihm, ist das der Parameter, der diesem Konstruktor übergeben wird ein anderes Objekt der gleichen Klasse ist :

```
class Complex
{
    public:
        // Default Konstruktor
        Complex();        // Konstruktor mit Parametern
        Complex(double Real, double Imag);        //
Kopierkonstruktor
        Complex(const Complex& c);

        double GetReal();
        double GetImag();

        double GetAbsolute();
    private:
        double m_Real;
        double m_Imag;
};
```

Hier der Code aus der .cpp-Datei

```
Complex::Complex(const Complex& c)
{
    m_Real = c.m_Real;
    m_Imag = c.m_Imag;
}
```

Der Code ist auch hier ziemlich einfach. Da der übergebene Parameter c von der gleichen Klasse ist, können wir auf die privaten Member des anderen Objektes zugreifen und die Datenelemente einfach unseren Datenelementen zuweisen. Der Kopierkontruktor wird verwendet indem man als Parameter eine andere bereits erzeugte Variable als Parameter übergibt. Die Variable **x** im Beispiel unten wird erzeugt indem der Kopierkonstruktor aufgerufen wird. Überprüfe das am besten gleich mit dem Debugger !

```
int main()
{
    Complex c;           // Standardkontruktor
    Complex z(1.0, 2.0); // Konstruktor mit Parametern
    Complex x(z);        // Kopierkonstruktor


    double Realc = c.GetReal();
    double Realz = z.GetReal();

    return 0;
}
```

Inline

Funktionen können aus Effizienz-Gründen **inline** definiert werden, indem man den Code für eine Funktion direkt in die Klassendefinition schreibt (implizit). Der Code aus der .cpp-Datei wandert in die Headerdatei :

```
class Complex
{
    public:
        // Default Konstruktor
        Complex();
        // Konstruktor mit Parametern
        Complex(double Real, double Imag);
        // Kopierkonstruktor
        Complex(const Complex& c);
        // inline Funktionen
        double GetReal()
        {
           return m_Real;
        }
        double GetImag()
        {
            return m_Imag;
        }
```

```
        double GetAbsolute();

    private:
        double m_Real;
        double m_Imag;
};
```

Die Funktionen GetReal und GetImag sind so inline definiert und müssen nicht mehr in der .cpp-Datei definiert werden. Der Compiler erzeugt dadurch schnelleren Code, dafür wird das .exe-File grösser. Man kann Methoden auch explizit inline machen :

```
class Complex
{
    public:
        // Default Konstruktor
        Complex();        // Konstruktor mit Parametern
        Complex(double Real, double Imag);        //
Kopierkonstruktor
        Complex(const Complex& c);

        double GetReal();
        double GetImag();

        double GetAbsolute();

    private:
        double m_Real;
        double m_Imag;
};
// explizit inline
inline double GetReal()
{
    return m_Real;
}
inline double GetImag()
{
    return m_Imag;
}
```

Zugriff auf Datenelemente

Zurück zur Klasse CD von oben : Bei dieser Klasse können wir Daten nur setzen indem wir die verschiedenen "setze"-Funktionen aufrufen. Um auf die Datenelemente (m_Titel, m_Interpret, m_Laenge) zuzugreifen. Wollen wir diese Daten später wieder abrufen müssen wir etwas wie "hole"-Funktionen definieren:

```
#ifndef CD_H
#define CD_H

#include <string>

using std::string;

class CD
{
    public:
        // Default Konstruktor
        CD();

        void setzeTitel(const string& titel);
        void setzeInterpret(const string& interpret);
        void setzeLaenge(long sekunden);
        // lesende Funktionen (inline)
        string holeTitel()
        {
            return m_Titel;
        }
        string holeInterpret()
        {
            return m_Interpret;
        }
        double holeLaenge()
        {
            return m_Laenge;
        }

    private:
        string m_Titel;
        string m_Interpret;
        long   m_Laenge;

};

#endif
```

Es mag mühsam erscheinen, für jedes Datenelement eine "setze"- und eine "hole"-Methode **(Get/Set)** zu definieren (das gibt zu viel Arbeit). Durch definieren der Datenelemente im public-Teil der Klasse könnte man sich das sparen. **Tu das nicht !!!!!!** Wer bei mir ohne Grund eine Variable public macht bekommt 234 (zweihundertvierundreissig !) Punkte Abzug ! Folgende Gründe (lese auch im Buch Seite 295):

- Überprüfung der Werte in der "setze"-Funktion. z.B. Bereichsüberprüfung, oder Umrechnungen
- Kapselung der internen Daten, wie wir bereits gesehen haben, können wir nachträglich den Datentypen von Datenelementen ändern (CDLaenge von letzter Lektion). Die Zugriffsmethoden müssen dann zwar Umrechnungen anstellen, aber immerhin müssen nicht alle, die die Klasse verwenden ihren Code neu schreiben.
- In grossen Projekten, in denen von vielen Orten aus auf ein Objekt zugegriffen wird ist es **fast unmöglich** zu Laufzeit festzustellen von wo aus ein Datenelement geändert wurde, wenn das Datenlement public ist. Gibt es eine "setze"-Methode können wir einfach einen Haltepunkt setzen (Breakpoint).

const-Objekte

Genau wie wir Variablen von eingebauten Datentypen konstant, also nur zum Lesen anlegen können, können wir Objekte Konstant machen :

```
#include "cd.h"

int main()
{
    const CD eineCD;

    return 0;
}
```

Damit wollen wir klarmachen, dass wir nur noch lesend auf das Objekt "eineCD" zugreifen wollen.
Unsere "hole"-Funktionen erfüllen genau diese Anfoerderung, sie ermöglichen den Lese-Zugriff auf das eineCD-Objekt :

```
#include "cd.h"
#include <iostream>
using namespace std;

int main()
{
    const CD eineCD;
    cout << eineCD.holeTitel() << endl;

    return 0;
}
```

Der Compiler wird das aber nicht kompilieren ! Er kann nicht von selber herausfinden ob, die Funktion unser Objekt "eineCD" unverändert lässt. Um dem Compiler zu zeigen, dass eine Methode das Objekt nicht ändert, müssen wir diese **const** definieren :

```
ifndef CD_H
#define CD_H

#include <string>

using std::string;

class CD
{
public:
    // Default Konstruktor
    CD();

    void setzeTitel(const string& titel);
    void setzeInterpret(const string& interpret);
    void setzeLaenge(long sekunden);
    // lesende Funktionen (inline)
    string holeTitel() const
    {
        return m_Titel;
    }
    string holeInterpret() const
    {
        return m_Interpret;
    }
    double holeLaenge() const
    {
        return m_Laenge;
    }

private:
    string m_Titel;
    string m_Interpret;
    long   m_Laenge;

};

#endif
```

Unsere "hole"-Methoden sind nun **const.** Der Compiler überprüft nun, dass in den Methoden das Objekt selber nicht geändert wird : also folgende Methode wird vom Compiler **nicht kompiliert !**

```
        ...
        double holeLaenge() const
        {
            m_Titel = "Test"; // Fehler !!!
            return m_Laenge;
        }
        ...
```

Wichtige Bemerkung am Rande :
Als Anfänger ertappt man sich häufig dabei, dass man für seine Klassen für jedes Datenelement Get- und Set-Methoden ("hole"/"setze"). Überlege dir immer folgendes : musst Du wirklich auf das Datenelement zugreifen ? Stelle Dir immer die Frage was willst Du von deinem Objekt. Beispiel CD-Klasse :
Wir können bei der CD-Klasse nun auf jedes Datenelement zugreifen. Also könnten wir in unserem main folgenden Code zum Ausgeben der CD schreiben :

```
#include "cd.h"
#include <iostream>
using namespace std;

int main()
{
    const CD eineCD;
    cout << eineCD.holeTitel() << endl;
    cout << eineCD.holeInterpret() << endl;
    cout << eineCD.holeLaenge() << endl;

    return 0;
}
```

Wenn wir wollen, dass die CD auf der Console ausgegeben wird, dann solch die CD sich selber darum kümmern ! Eigentlich wollen wir die CD definieren können, die Elemente mit den "setze"-Methoden definieren und schlussendlich die CD ausgeben.
Die **bessere** Abstraktion der CD für unser Problem ist also wie folgt :

```
#ifndef CD_H
#define CD_H

#include <string>

using std::string;

class CD
{
    public:
        // Default Konstruktor
        CD();
```

```
        void setzeTitel(const string& titel);
        void setzeInterpret(const string& interpret);
        void setzeLaenge(long sekunden);

        // gibt Inhalt auf der Console
        // aus
        void ausgabe();
    private:
        string m_Titel;
        string m_Interpret;
        long   m_Laenge;

};

#endif
```

Noch besser wäre hier, dass die CD die Daten selber mit einer einlesen()-Methoden von der Console liest !

Weiteres:

- **this**, Objektidentität (Seite 301)
- call by Value, by Reference
- Rückgabe von Objekten (als Referenz und als Zeiger)
- globale Funktionen vs. Klassen-Methoden
- static

Übung

1. Schreibe die Klasse CD um. Sie soll folgende Punkte erfüllen

- Konstruktor (mit Ausgabe, dass ein Objekt erzeugt wurde)
- Destruktor (mit Ausgabe, dass ein Objekt abgebaut wurde)
- Konstruktor mit Parametern (Titel, Interpret und Länge)
- Schreibe konstante Zugriffsfunktionen für alle Datenelemente
- Schreibe set-Methoden für alle Datenelemente
- Schreibe eine Funktion einlesen, die über cin den Benutzer direkt nach dem Titel, dem Interpreten und der Laenge fragt
- Schreibe eine Funktion zur Ausgabe der CD-Informationen

Verwende eine Header und eine Quellcode-Datei. Schreibe eine main-Funktion, mit der die Methoden getestet werden können.

2. Ergänze den Kopierkonstruktor

3. Erzeuge im main 6 CD-Objekte und lass diese durch den Benutzer definieren (am besten mit der "einlesen"-Methode). Danach soll eine Sortierfunktion aufgerufen werden, die die CD's der Länge nach sortiert. Du wirst dafür eine Methode in der Klasse CD brauchen, die eine CD mit einer anderen vergleicht. Du brauchst auch eine Swap-Funktion für CD's und eine Sortierfunktion wie aus der vorletzten Übung.