Inhaltsverzeichnis

1. Grundlagen: Objekt orientiertes Programmieren

1.1. Ziele

☑ Theorie der OOP kennen

☑ Eigene Datentypen/Klassen erstellen können

☑ Objekte erzeugen und verwalten können

☑ Vererbung anwenden können

Quelle: http://www.cplusplus.com/doc/tutorial/

1.2. Konzepte der OOP

1.2.1. Warum OOP

Ziel ist es der Software-Industrie Mittel, Konzepte in die Hand zu geben, um

☑ **Wartbarkeit**: robuste, wartbare Programme

- ☑ kleine Änderungen am Programm sollen sich nicht auf die Robustheit des gesamten Programmes auswirken

☑ **Wiederverwendbarkeit**: wiederverwendbare Programme bzw. Programmteile

Diese Paradigmen der Softwareentwicklung werden durch

☑ **Informationhiding** mittels Datenabstraktion

☑ **Reuseability** mittels Vererbung

erreicht.

Beide Konzepte, die die wesentlichen Bestandteile der objektorientierten Programmierung sind.

1.2.2. Abstraktion und Kapselung

Mit diesen Konzepten will man obige Ziele erreichen. Man bedient sich folgender Überlegungen:

Der Programmierer/in soll neben den primitiven Datentypen (int, float, char, ...) auch selbst Datentypen erzeugen können.

So könnte man sich im Bereich der

☑ Autoindustrie folgende Datentypen vorstellen:
Auto, Rad, Zylinder, Fahrgestell, ...

☑ Schule
Schueler, Lehrer, Unterrichtsfach, ...

☑ Spiele

Spieler, Held, Gegner, Zaubertrank, Hindernis

Solche neuen Datentypen (auch **Klassen** genannt) haben

☑ **Eigenschaften** und

☑ **Verhalten**

Die Eigenschaften werden durch Daten beschrieben.

So hat z.B. die Klasse Held folg. Eigenschaften:

☑ Name,

☑ Anzahl der Leben,

☑ Energiepunkte,

☑ Farbe

☑ ...

Daneben besitzen Klassen auch ein Verhalten, das durch Funktionen (wir sprechen hier von Methoden) festgelegt wird.

So kann z.B. ein Held

☑ kaempfen

☑ einen Gegenstand aufnehmen oder abgeben

☑ seine Energiepunkte erhöhen

☑ ...

Klassen nennt man auch **Datenkapseln**. D.h. man kann nicht direkt auf die Eigenschaften einer Klasse zugreifen, sondern man muss die zur Verfügung stehenden **Methoden** verwenden.

1.2.3. Wiederverwendung

Damit wird das Prinzip des Informationhiding erfüllt und die Wiederverwendbarkeit bzw. Wartbarkeit von Programmen erhöht. Sollten sich Eigenschaften einer Klasse ändern, ist nicht das gesamte Programm davon betroffen, weil man ohnehin nicht direkt auf die Eigenschaften zugreifen konnte.

1.2.4. Beziehungen

Objekte und Klassen existieren für gewöhnlich nicht völlig alleine, sondern stehen in Beziehungen zueinander. So ähnelt ein Fahrrad beispielsweise einem Motorrad, hat aber auch mit einem Auto Gemeinsamkeiten. Ein Auto ähnelt dagegen einem Lastwagen. Dieser kann einen Anhänger haben, auf dem ein Motorrad steht. Ein Fährschiff ist ebenfalls ein Transportmittel und kann viele Autos oder Lastwagen aufnehmen, genauso wie ein langer Güterzug. Dieser wird von einer Lokomotive gezogen. Ein Lastwagen kann auch einen Anhänger ziehen, muss es aber nicht. Bei einem Fährschiff ist keine Zugmaschine erforderlich, und es kann nicht nur Transportmittel befördern, sondern auch Menschen, Tiere oder Lebensmittel.

Wir wollen ein wenig Licht in diese Beziehungen bringen und zeigen, wie sie sich in objektorientierten Programmiersprachen auf wenige Grundtypen reduzieren lassen:

☑ "is-a"-Beziehungen (**Generalisierung**, **Spezialisierung**)
☑ "part-of"-Beziehungen (**Aggregation**, **Komposition**)
☑ Verwendungs- oder Aufrufbeziehungen

Generalisierung und Spezialisierung: IS-A
Zuerst wollen wir die "is-a"-Beziehung betrachten. "is-a" bedeutet "ist ein" und meint die Beziehung zwischen "ähnlichen" Klassen.

Die "is-a"-Beziehung zwischen zwei Klassen *PERSON* und *SCHUELER* sagt aus,

dass "*SCHUELER* eine *PERSON* ist",

also alle Eigenschaften von *PERSON* besitzt, und vermutlich noch ein paar mehr.

- *SCHUELER* ist demnach eine **Spezialisierung** von *PERSON*.
- Andersherum betrachtet, ist *PERSON* eine **Generalisierung** (Verallgemeinerung) von *SCHUELER*.

"is-a"-Beziehungen werden in objektorientierten Programmiersprachen durch **Vererbung** ausgedrückt. Eine Klasse wird dabei nicht komplett neu definiert, sondern von einer anderen Klasse **abgeleitet**. In diesem Fall **erbt** sie alle Eigenschaften dieser Klasse und kann nach Belieben eigene **hinzufügen**.

In unserem Fall wäre also *SCHUELER* von *PERSON* abgeleitet.

- *PERSON* wird als **Oberklasse**,
- *SCHUELER* als **Unterklasse** bezeichnet.

Aggregation und Komposition : HAS-A
Der zweite Beziehungstyp, die "part-of"-Beziehungen, beschreibt die **Zusammensetzung** einer Klasse aus anderen Klassen (dies wird auch als *Komposition* bezeichnet).

So besteht beispielsweise der Güterzug aus einer (oder manchmal zwei) Lokomotiven und einer großen Anzahl Güterzuganhänger. Der Lastwagen besteht aus der LKW-Zugmaschine und eventuell einem Anhänger. Ein Fahrrad besteht aus vielen Einzelteilen.

Komposition bezeichnet die strenge Form der Aggregation auf Grund einer existentiellen Abhängigkeit. So besteht z.B. ein Fahrrad aus 2 Rädern. Wohingegen der Lastwagen aus der LKW-Zugmaschine und eventuell einem Anhänger besteht. (=Aggregation)

Verwendungs- und Aufrufbeziehungen

Allgemeine Verwendungs- oder Aufrufbeziehungen finden in objektorientierten Programmiersprachen ihren Niederschlag darin, dass Objekte als lokale Variablen oder Methodenargumente verwendet werden. Sie werden auch mit dem Begriff Assoziationen bezeichnet.

1.2.5. Polymorphismus

Wird später behandelt. Hier aber nur soviel:

Dynamic Binding und Base Class Pointers/Objektvariablen sind die Elemente die Polymorphismus ermöglichen.

Polymorphismus bedeutet direkt übersetzt etwa "Vielgestaltigkeit" und bezeichnet zunächst einmal die Fähigkeit von Objektvariablen, Objekte unterschiedlicher Klassen aufzunehmen.

Das heißt erst zur Programmlaufzeit wird ermittelt, welche Methoden tatsächlich aufzurufen sind.

1.3. Klassen und Objekte

Der Programierer/in soll neben den primitiven Datentypen (int, float, char, ...) auch selbst Datentypen erzeugen können.

So könnte man sich im Bereich der

☑ Autoindustrie folgende Datentypen vorstellen:
Auto, Rad, Zylinder, Fahrgestell, ...

☑ Schule
Schueler, Lehrer, Unterrichtsfach, ...

☑ Spiele
Spieler, Held, Gegner, Zaubertrank, Hindernis

Objektorientierte Programmiersprachen erlauben das Definieren neuer Datentypen. Diese werden Klassen genannt.

KLASSEN sind Datentypen
☑ Klasse = ein vom Programmierer erstellter Datentyp
☑ d.h. der Programmierer ist in der Lage neben int, float, ...
neue Datentypen zu definieren
☑ zB. kann der Programmierer die Klasse Flugzeug, Auto, ... erstellen.

KLASSE = DATEN + FUNKTIONEN(werden METHODEN genannt)
☑ die neuen Datentypen/Klassen haben auch ihre eigenen Operatoren/Funktionen
☑ zB. kann man Schueler addieren
In diesem Fall könnte man sich vorstellen, eine Schuelerliste zu erhalten. Die selbst wieder eine Klasse ist. Und diese Schulerliste könnte man dann ausdrucken oder per mail verschicken, ...
☑ zB. sich bei der Klasse Date den Wochentag, den MonatsNamen , ... ermitteln.
☑ zB. kann die Klasse Matrix die Matrixmultiplikation

OBJEKTE sind Variablen
☑ Von Klassen können Objekte erzeugt werden.
☑ (vgl. dazu: vom Datentyp int können Variablen erzeugt werden)

Zusammenfassung

☑ Klassen
sind eine Art Beschreibungsplan vergleichbar mit einem Bauplan für einen bestimmten Autotyp (zb: AudiA3).

☑ Die Objekte sind dann die jeweiligen Autos (z.B. meinAuto, deinAuto)

Der Programmierer muss also

☑ Klassen definieren

☑ Objekte erzeugen und bei Objekten Methoden aufrufen

1.3.1. Klassen und Objekte in Cpp

Eine Klassendefinition wird durch das Schlüsselwort **class** eingeleitet. Anschließend folgt innerhalb von geschweiften Klammern eine beliebige Anzahl an Variablen- und Methodendefinitionen.

```
class Rechteck {
     private:
          int m_laenge;
          int m_breite;
};
```

Die Variablen m_laenge und m_breite werden **Membervariablen** bzw. **Felder** genannt.

Um ein Objekt namens meinRechteck zu erzeugen:

```
Rechteck* meinRechteck= new Rechteck(); // zeigerschreibweise
Rechteck wald; // wald ist bereits ein Objekt
```

Mit diesem Rechteck kann man allerdings noch nichts machen, da es kein Verhalten (Methoden) hat. Diese wollen wir jetzt definieren.

1.4. Methoden

Das Verhalten von Objekten wird in Form von Methoden festgelegt. Dabei handelt es sich um Funktionen, die aufgerufen werden können. Man kann also **nicht** direkt auf die Membervariablen zugreifen, sondern nur via **Methoden** die Eigenschaften (Membervariablen) verändern. (=Abstraktion).

1.4.1. Methoden sind Funktionen

Aus Gründen der besseren Strukturierung von Programmen kann man Anweisungen zu Blöcken zusammenfassen und diesen Blöcken Namen geben. Man nennt diese benannten Blöcke Funktionen oder Methoden.

Wenn man nun diese Anweisungen ausführen lassen will, braucht man nur mehr den Namen des Blockes (wir sagen: den Funktionsnamen/Methodennamen) ins Programm an die gewünschte Stelle schreiben. So könnte eine Abfolge von Funktionsaufrufen so aussehen:

```
gehZurCafetria();
issWas();
trinkWas();
kommwieder();
```

Nun könnte man wieder diese vier Anweisungen (die ja selbst jeweils aus einer Vielzahl von Anweisungen bestehen) zu einem benannten Block zusammenfassen. Z.B. machePause()

```
machePause();
```

usw.

Welche Vorteile hat diese Vorgangsweise?

☑ Man kann komplizierten Anweisungsfolgen einen Namen geben und nur mehr diesen Namen verwenden. (zB: sin(30.0))

☑ Man kann Funktionen von jeweils verschiedenen Programmierern programmieren lassen. (Arbeitsteilung)

☑ Man könnte Funktionen einmal schreiben und in anderen Programmen wieder verwenden. (Wiederverwendbare Programmteile)

Ja, dies sind Vorteile, ohne die die moderne Programmierung nicht sein könnte.

D.h. der Progammierer

☑ erstellt Funktionen/Methoden und

☑ ruft Funktionen/Methoden auf.

Aber, was bedeuten die runden Klammern? Nun, um Funktionen/Methoden wirksamer und vielfältiger zu machen, kann man diesen beim Aufruf Werte mitgeben bzw. von Funktionen/Methoden Werte zurückbekommen (z.B.: x= sin(30.0);)

1.4.2. Parameterübergabe und Rückgabe von Funktionen/Methoden

Anhand eines sehr einfachen Beispieles wollen wird dies zeigen:

Datei: rechteck.h

```
// Die Spezifizierung der Klasse Rechteck
//
#ifndef RECHTECK_H
#define RECHTECK_H

class Rechteck {
     private:        // Member bzw. Felder sind private
          int m_laenge;
          int m_breite;
     ...

     public:         //Methoden meist public
          void init(int breite, int laenge);


          int berechneFlaeche();
};
```

```
#endif
```

Datei: rechteck.cpp

```
// Die Implementierung der Klasse Rechteck
//

#include "rechteck.h"

void Rechteck::init(int breite, int laenge){
          m_breite= breite;
          m_laenge= laenge;
}

int Rechteck::berechneFlaeche(){
          int flaeche;
          flaeche= m_laenge * m_breite;
          return flaeche;
}
     ....
```

Hier sehen Sie nun den Aufruf von Methoden.

Datei: testrechteck.cpp

```
/* N.N., Datum
   testrechteck.cpp
 */

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

int main(){
     ...
     int flaeche;
     int l;
     int b;

     Rechteck* wiese= new Rechteck(); //ein Rechteck erzeugen
     Rechteck wald;
     .....
     //laenge, breite einlesen
     ...

     // Rechteck laenge, breite setzen
     wiese->init(l, b);
     wald.init(l,b);
```

```
    // flaeche berechnen durch Aufruf der Funktion
    flaeche= wiese->berechneFlaeche();
    flaeche= wald.berechneFlaeche();

    //flaeche ausgeben
    ....

    delete wiese;
}
```

1.4.3. lokale Variablen

Variablen, die innerhalb einer Funktion/Methode vereinbart werden, sind nur dort gültig und sichtbar also dort nutzbar.

1.4.4. Methoden in Cpp

Es gibt folgende Arten von Methoden:

☑ **Konstruktoren**
heißt wie die Klasse und wird beim Erzeugen eines Objektes automatisch aufgerufen (=Initialisierung der Membervariablen)

☑ **Destruktoren**
heißt wie die Klasse, allerdings mit einem vorangestellten Tilde-zeichen (z.B: ~Rechteck()) und wird beim Zerstören eines Objektes aufgerufen. Hat in Java wenig Bedeutung. In C++ hingegen sehr wichtig.

☑ **Get/Set-Methoden**
zum lesenden/schreibenden Zugriff auf Membervariablen

☑ **Allgemeine Methoden**

Merke:
Methoden, die nur lesend auf die Member zugreifen, sollen das Schlüsselwort **const** verwenden.

1.5. Die fertige Klasse Rechteck

Zur Erstellung von Cpp-Klassen verwendet man folgendes Vorgehen:

☑ Die Spezifikation der Klasse kommt in die Header-Datei.
☑ Die Implementierung der Klasse kommt in die Cpp-Datei.

1.5.1. rechteck.h

```
/* a.hofmann, 2007
   rechteck.h
*/

#ifndef RECHTECK_H
#define RECHTECK_H

#include <string>
using namespace std;

class Rechteck {
     private:
          int m_laenge;
          int m_breite;

     public:
          //Default Konstruktor
          Rechteck();

          //Allgemeiner Konstruktor
          Rechteck(int laenge, int breite);

          // set/get Methoden
          void setBreite(int breite);

          void setLaenge(int laenge);

          int getBreite()const;

          int getLaenge() const;

          //allg. Methoden
          int flaeche() const;

          string toString() const;

          //Destruktor
          ~Rechteck();
};
#endif
```

1.5.2. rechteck.cpp

```
/*
  a.hofmann, 2007
  rechteck.cpp
*/
#include <iostream>
```

```
#include <string>
#include "rechteck.h"
using namespace std;


//Default Konstruktor
Rechteck::Rechteck(){
          m_laenge= 10;
          m_breite= 5;
}

//Allgemeiner Konstruktor
Rechteck::Rechteck(int laenge, int breite){
          m_laenge= laenge;
          m_breite= breite;
}

// set/get Methoden
void Rechteck::setBreite(int breite){
          m_breite= breite;
}

void Rechteck::setLaenge(int laenge){
          m_laenge= laenge;
}

int Rechteck::getBreite()const{
          return m_breite;
}

int Rechteck::getLaenge()const{
          return m_laenge;
}

//allg. Methoden
int Rechteck::flaeche()const{
          return m_laenge*m_breite;
}

string Rechteck::toString()const{
          //?!?!?!ERROR
          return "Rechteck: laenge="+m_laenge+" breite="+m_breite;
          // Lösung: siehe Programm: rechteck

}

Rechteck::~Rechteck(){
     //hier in diesem Fall ist nichts zu tun
}
```

Jetzt kann man Objekte erzeugen und mit diesen arbeiten.

1.5.3. testRechteck.cpp

```
/* a.hofmann, 2007
  testRechteck.cpp
*/
#include <iostream>
#include <string>
#include "rechteck.h"
using namespace std;

int main(){

     //Default Konstruktor wird aufgerufen
     Rechteck* meinRechteck= new Rechteck();

     //Allg. Konstruktor wird aufgerufen
     Rechteck* wiese= new Rechteck(20, 5);

     cout<<"Wiese hat die Länge: " << wiese->getLaenge() << endl;

     //?!?!?
     cout<<"Wiese hat die Flaeche: " + wiese->flaeche() << endl;

     cout<< wiese->toString();


     delete wiese;
     delete meinRechteck;

     return 0;
}
```

Hier sehen Sie nun das obige Beispiel ohne Zeigerschreibweise.
testRechteck.cpp

```
/* a.hofmann, 2007
  testRechteck.cpp
*/
#include <iostream>
#include <string>
#include "rechteck.h"
using namespace std;

int main(){

     //Default Konstruktor wird aufgerufen
     Rechteck wiese;
```

```
    //Allg. Konstruktor wird aufgerufen
    Rechteck wald(20, 5);

    cout<<"Wiese hat die Länge: " << wiese.getLaenge() << endl;

    cout<<"Wiese hat die Flaeche: " << wiese.flaeche() << endl;
    cout<<"Wald hat die Flaeche: " << wald.flaeche() << endl;


    cout<< "Meine Wiese: \n" << wiese.toString();

    return 0;
}
```

1.5.4. Hinweis: stringstream und sstream.h

Bei der obigen Lösung sind Fehler enthalten.

Lösung: verwenden Sie die Klasse stringstream, um verschiedene Daten auszugeben.

```
#include <sstream>
…
stringstream os;
os << "Dies ist eine C-Zeichenkette " << nummer << endl;
...
string s= os.str();
```

1.6. Beispiele

1.6.1. Beispiel: oop1-Rechteck

Projektname: oop1-Rechteck

Erstellen Sie nach obigem Beispiel die Klasse Rechteck und ändern sie die Methode toString() derart ab, dass zusätzlich die Fläche in Form von Sternchen angezeigt wird.
Beispiel:
Rechteck: laenge=5 breite=3
*****
*****
*****

1.6.2. Projekt: oop1-Banksysteme – Die Klasse Konto

Projektname: oop1-Banksysteme

Erstellen Sie das Projekt oop1-Banksysteme
und erstellen Sie darin die Klasse Konto mit folg. Aufbau:
Member:
☑ int m_nummer;
☑ string m_inhaber;
☑ double m_betrag;

Methoden: (Parameter,Rückgaben selbst überlegen)
☑ Konstruktor/Destruktor
☑ get/set-Methoden
☑ toString()
☑ abheben()
☑ einlegen()

Erstellen Sie dazu ein kleines Testprogramm.

1.6.3. Hinweis: Double Werte mit 2 Nachkommastellen ausgeben

```
string Konto::toString() const{
	stringstream os;

	// Hinweis: double Werte mit 2 Nachkommastellen ausgeben
	//
http://www.java2s.com/Code/Cpp/Data-Type/doublevaluecoutformat.htm

	os.precision(2);

	os << "KONTO: " << endl;
	os << "  Nummer:\t" << m_nummer << endl;
	os << "  Inhaber:\t" << m_inhaber << endl;
	os << "  Betrag:\t" << fixed << m_betrag << endl;

	return os.str();
}
```

1.6.4. Hinweis: Default Parameter festlegen

Funktionen/Methoden können sogenannte Default-Parameter besitzen.

Bsp:

```
class Konto{
	…

	Konto(string inhaber, double betrag, int nummer=-1);
	….
};
….
```

```
…
Konto::Konto (string inhaber, double betrag, int nummer){
      m_inhaber= inhaber;
      m_betrag= betrag;
      m_nummer= nummer;
}
```

```
…
Konto meinKonto("Max", 100.00, 4711);
Konto deinKonto("Moritz", 100.00);
```

```
Das Objekt meinKonto hat dann die Membervariable m_nummer mit dem Wert 4711.
Das Objekt deinKonto hat dann die Membervariable m_nummer mit dem Wert -1.
```

1.6.5. Projekt: oop1-Banksysteme - Die Klasse Bank

Projektname: oop1-Banksysteme

Erstelle die Klasse Bank
Member:
☑ static const int maxanz_konten;

☑ int m_nummer; // BLZ
☑ string m_name;
☑ Konto* m_konten[maxanz_konten];

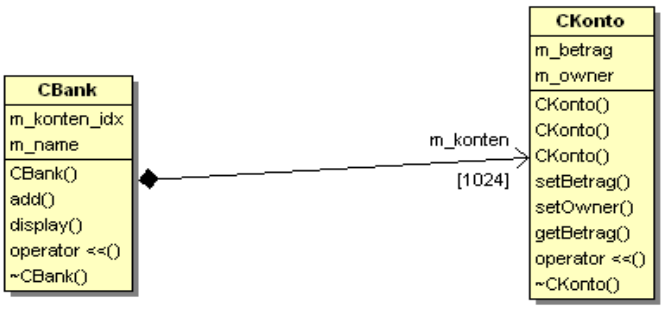
Methoden: (Parameter,Rückgaben selbst überlegen)
☑ Konstruktor/Destruktor
☑ get/set-Methoden
☑ toString()
☑ addKonto(), delKonto()
☑ addBetrag(double) ... addiert zu jedem Konto einen Betrag
☑ minBetrag(double)

Erstellen Sie dazu ein kleines Testprogramm.

```
....
#include <iostream>
using namespace std;

#include "konto.h"
#include "bank.h"


int main() {
      cout << "!!!Banksysteme!!!" << endl;

      Bank* meinB= new Bank("Hofmann Unlimited");
```

```
    Konto* k1= new Konto("Anton Hofmann",1000.0);
    Konto* k2= new Konto("Beta Hofmann", 1000.0);
    Konto* k3= new Konto("Gamma Hofmann",1000.0);


    meinB->addKonto(*k1);
    meinB->addKonto(*k2);
    meinB->addKonto(*k3);

    cout <<"Die Bank: VOR der Spesenberechnung *********\n";

    cout << meinB->toString() << endl;

    cout <<"Die Bank: NACH der Spesenberechnung *********\n";
    meinB->minBetrag(10.0);
    cout <<meinB->toString();

    cout <<"Die Bank: NACH dem Löschen von k3 (Gamma Hofmann)
*********\n";
    meinB->delKonto(*k3);
    cout <<meinB->toString();

    delete k1;
    delete k2;
    delete k3;

    delete meinB;

    return 0;
}
```

Das UML Diagramm:



1.7. vector: Objekte in Containern speichern

Die so genannte STL (Standard Template Library) ist eine C++ Bibliothek, die u.a. das komfortable Speichern/Verwalten von vielen Objekten ermöglicht.

Wenn man viele Objekte zu verwalten hat (wie z. B.: Die Klasse Bank speichert viele Konten, ...), weiß man oft nicht wie viele Objekte (z.B. Konten) tatsächlich zur Laufzeit des Programms gespeichert werden sollen. Arrays haben den Nachteil, dass man bereits zur Übersetzungszeit (=statisch) die maximale Anzahl der Arrayelemente angeben muss. Hier hilft uns die Klasse vector aus der STL. Sie kann zur Laufzeit beliebig viele Objekte speichern. Und man kann mit vectoren gleich arbeiten wie mit den bekannten Arrays.

Hier nun ein einfaches Beispiel:

```
// a.hofmann 2007
// vector1.cpp
// dem STL

#include <string>
#include <vector>
#include <iostream>
using namespace std;

int main(){
    vector<int> iVector;
    vector<string> sVector;

    // Iterator, um auf vector-elemente
    // lesend und schrfeibend zugreifen zu können
    vector<int>::iterator it;
    vector<string>::iterator st;

    // Iterator, um auf vector-elemente
    // NUR lesend zugreifen zu können
    vector<int>::const_iterator cit;
    vector<string>::const_iterator cst;


    iVector.push_back(1);
    iVector.push_back(2);
    iVector.push_back(3);

    sVector.push_back("eins");
    sVector.push_back("zwei");
    sVector.push_back("drei");



    cout << "\nSTL: vector Beispiele ..............." << endl<<endl;

    // Elementweiser Zugriff
    cout << iVector[0] << endl;
```

```
    cout << sVector[0] << endl;

    for (it= iVector.begin(); it!= iVector.end(); it++)
        cout << *it << endl;

    for (st= sVector.begin(); st!= sVector.end(); st++)
        cout << *st << endl;




    cout<< "Size von SVector: " << sVector.size() << endl;

    // löschen
    st= sVector.begin();
    st++;
    sVector.erase(st);
    cout<< "Size von SVector (nach erase): " << sVector.size() << endl;

    // const Iterator demonstration
    for (cst= sVector.begin(); cst!= sVector.end(); cst++)
        cout << *cst << endl;


    sVector.pop_back(); // Das letzte Element wird gelöscht

    return 0;
}
```

1.7.1. Hinweis: vector und das Löschen von Objekten mit erase

Man beachte, dass nach dem Löschen eines Elementes die Iteratoren ungültig werden.

http://www.cplusplus.com/reference/stl/vector/erase/

…

Because vectors keep an array format, erasing on positions other than the vector end also moves all the elements after the segment erased to their new positions, which may not be a method as efficient as erasing in other kinds of sequence containers (deque, list).

This invalidates all iterator and references to *position* (or *first*) and its subsequent elements.

...

1.7.2. Projekt: oop1-Banksysteme-STL-Vector

Projektname: oop1-Banksysteme-STL-Vector

Kopieren Sie das Projekt oop1-Banksysteme und

ändern Sie die Klassen so, dass als Container für die Konten die Klasse vector aus der STL verwendet wird.

1.7.3. Hinweis: c++11 standard und die for-each Schleife

Wenn man folgende Einstellung vornimmt:

```
g++ -std=c++11 source.c -o source.exe
```

kann man folgende Schleife verwenden:

```
for(Konto& k: m_konten)
     cout << k.toString();
```

1.8. this-Zeiger

Bei this handelt es sich um einen Zeiger, der beim Anlegen eines Objekts automatisch generiert wird. this ist ein Zeiger, der auf das aktuelle Objekt zeigt und dazu verwendet wird, die eigenen Methoden und Instanzvariablen anzusprechen.

```
....
void Konto::setBetrag(double betrag){
          this->betrag= betrag;
}
....
```

1.9. Klassenmember und Klassenmethoden: static

Klassenmember und Klassenmethoden werden durch das vorangestellte Wort **static** gekennzeichnet.

Es handelt sich dabei um sogenannte „**Klassenvariablen**“. Es sind Variablen, die nicht bei jedem Instanziieren eines Objektes neu angelegt werden. Nein, Klassenvariablen existieren praktisch **nur einmal** für alle Objekte. Dadurch kann man auf elegante Weise zum Beispiel einen Zähler erzeugen, der angibt wieviele Objekte einer Klasse bereits instanziiert wurden.

Hier ein Beispiel

rechteck.h

```
class Rechteck{
     private:
          ...
          static int m_anzahl; // Deklaration der Klassenvariablen

     public:
          ....
          static int getAnzahl(); // Deklaration der Klassenmethode
```

```
...};
```

rechteck.cpp

```
// Definition und Initialisierung der KLASSENvariablen (static members)

int Rechteck::m_anzahl=0;

…
// Die programmierte Methode (Achtung: Hier fehlt das Wort static)
int Rechteck::getAnzahl(){
     return m_anzahl;
}
```

Der Aufruf einer KLASSEN-Methode:

```
int wieviele= Rechteck::getAnzahl();
```

1.9.1. Projekt: oop1-Rechteck-static

Projektname: oop1-Rechteck-static

Bringen Sie das folgende Programm zum Laufen.
02-uebung\u-rechteck-static

Hinweis:
Erweitere die Klasse Rechteck um einen Klassenmember m_anzahl, der mitzählt wieviel Rechteck-Objekte gerade im Einsatz sind. (=Instanzenzähler). D.h. bei jedem Konstruktor wird der member erhöht. Bei jedem Destruktor wird der member erniedrigt.

1.9.2. Projekt: oop1-Banksysteme-STL-Vector-static

Projektname: oop1-Banksysteme-STL-Vector-static

Bringen Sie das folgende Programm zum Laufen.
02-uebung\03-u-Banksysteme-STL-vector-static

Ziel:
...ändern Sie die Klassen so, dass die Kontonummern als static MemberVariablen der Klasse Konto verwaltet werden.
D.h.

☑ Das Erzeugen von Konten wird in die Klasse Bank verlegt durch die Methode Bank::addKonto(...).
☑ Die Klasse Konto verwaltet eine static Variable, um immer neue Kontonummern beim Anlegen eines Kontos zu bekommen.

Hier ein Beispiel für die Verwendung:
main.cpp

```
// a.hofmann 10.06
// main.cpp
// demo: klasse und objekt und container
// g++ main.cpp Konto.cpp Bank.cpp -o main.exe

#include "Konto.h"
#include "Bank.h"
#include <iostream>
using namespace std;

int main(){

     Bank *meineBank;
     meineBank = new Bank("Hofmann unlimited");

     cout << "\n\n"<<endl;
     cout << "-------------------------------------------------"<<endl;
     cout << " DEMO: static member "<<endl;
     cout << " DEMO: Die KontoNummer werden als static member "<<endl;
     cout << " DEMO: in der Klasse Konto gehalten "<<endl;
     cout << " DEMO: "<<endl;
     cout << " DEMO: Dadurch werden beim Hinzufügen, immer neue
KontoNummern erzeugt"<<endl;
     cout << " DEMO: "<<endl;
     cout << " DEMO: Testen Sie, ob IMMER eine neue KontoNummer erzeugt
wird????"<<endl;
     cout << " -------------------------------------------------"<<endl;


     cout << "\n\nDIE BANK -----------------"<<endl;
     cout << ">>> hier noch mit allen 3 Konten: BEACHTE die
KontoNummer!!!"<<endl;
     meineBank->addKonto("Anton Hofmann", 100.0);
     meineBank->addKonto("Beta Hofmann", 200.0);
     meineBank->addKonto("Gamma Hofmann",300.0);

     cout << meineBank->toString() << endl;



     cout << "\n\nDIE BANK -----------------"<<endl;
     cout << ">>> nach dem Löschen v. Konto Anton Hofmann: : BEACHTE die
KontoNummer!!!"<<endl;
     meineBank->delKonto("Anton Hofmann");
```

```
    cout << meineBank->toString() << endl;
    cout << endl;

    cout << "\n\nDIE BANK -----------------"<<endl;
    cout << ">>> nach dem Einfügen v. Konto Omega Hofmann: BEACHTE die
KontoNummer!!!"<<endl;
    meineBank->addKonto("Omega Hofmann", 900.0);

    cout << meineBank->toString() << endl;
    cout << endl;

    delete meineBank;

   return 0;
}
```

1.10. Weitere Übungsaufgaben

Siehe CPP/02-oop1-klasse-vererbung/02-ueben/

1.10.1. Projekt: oop1-Pruefziffer

Erstellen Sie das Projekt: oop1-Pruefziffer und lösen Sie die in
02-ueben/
gestellten Aufgaben.

1.10.2. Projekt: oop1-Pseudorand

Erstellen Sie das Projekt: oop1-Pseudorand und lösen Sie die in
02-ueben/
gestellten Aufgaben.

1.11. Vererbung

Oft kann man bei auf den ersten Blick unterschiedlichen Klassen doch viele **Gemeinsamkeiten** finden. So haben z.B.: Lehrer und Schüler gemein, dass Sie einen Vornamen, Nachnamen, ein bestimmtes Alter, usw. haben.

Durch die so genannte Vererbung braucht der Programmierer beim Anlegen einer neuen Klasse (Schueler, Lehrer, Direktor, Abteilungsvorstand, ...) nicht jedesmal den gleichen Programmcode zur Verwaltung dieser Gemeinsamkeiten schreiben.

Ein enormer Gewinn, den alle Programmierer nutzen wollen.

Mann kann nicht nur Membervariablen vererben, sondern auch Methoden.

Hier zunächst einige Begriffe:

1.11.1. Oberklasse, Unterklassen

Oberklasse
wird auch Basisklasse oder "**super** class" genannt und beinhaltet alle Methoden und Member, die an später zu schreibende Klassen vererbt (zur dortigen Wiederverwendung) werden.

Unterklasse
wird auch Subklasse genannt. Sie enthält ihre eigenen Member und Methoden und zusätzlich automatisch alle Member und Methoden ihrer Oberklasse. Sie kann geerbte Methoden auch überschreiben.

1.11.2. public, private, protected

Eines ist aber zu beachten:
Auch Oberklassen haben ihre "Privatsphäre", d.h. eine Unterklasse darf nicht direkt auf die private Member ihrer Oberklasse zugreifen, sondern muss genauso wie jeder andere die Methoden der Basisklasse verwenden.

Ausnahme sind Member, die **protected** vereinbart wurden. Hier gilt, dass **nur Unterklassen** direkt auf diese Member zugreifen können.

1.11.3. Projekt: oop1-Firma – Die Oberklasse CPerson

Hier ein kleines Beispiel:

Projekt: oop1-Firma
Die Oberklasse: CPerson
cperson.h

```
// a.hofmann, 2007
// cperson.h
// Vererbung

#include <string>
using namespace std;

#ifndef CPERSON
#define CPERSON

class CPerson{
    private:
        // Private member. NIEMAND darf direkt drauf zugreifen
        int alter;
```

```
        // Protected member.
        // NUR Unterklassen dürfen direkt drauf zugreifen
     protected:
        string name;

    public:
        // Konstruktor
        CPerson(int alter, string name);

        // Zugriffsmethoden
        int getAlter() const;

        string toString() const;
};
#endif
```

cperson.cpp

```
// a.hofmann, 2007
// cperson.cpp
// Vererbung

#include <iostream>
#include <sstream>  // um int, double, .. in strings umzuwandeln

#include "cperson.h"
using namespace std;

CPerson::CPerson(int alter, string name){
    this->alter= alter;
    this->name= name;
}

int CPerson::getAlter() const{
     return alter;

     // return this->alter;
     // ist auch möglich, wird aber kaum verwendet
}

string CPerson::toString() const{
    ostringstream out;
    out << "\nNAME: " << name << "\nAlter: " << alter << "\n";
    return out.str();
}
```

1.11.4. Projekt: oop1-Firma – Die Unterklasse CArbeiter

Die Unterklasse: CArbeiter

Ein Arbeiter **IST-EINE PERSON** und **erbt** demnach die member der Klasse CPerson

carbeiter.h

```
// a.hofmann, 2007
// carbeiter.h
// Vererbung

#include <string>
#include "cperson.h"


#ifndef CARBEITER
#define CARBEITER

class CArbeiter : public CPerson
{
     private:
        int stunden;
        double stundenLohn;

     public:
        // Konstruktor
        CArbeiter( int alter, string name,
                   int stunden, double stundenLohn );

        // Zugriffsmethoden
        int getAlter() const;

        string toString()const;

        double getGehalt() const;
};
#endif
```

carbeiter.cpp

```
// a.hofmann, 2007
// carbeiter.cpp
// Vererbung

#include "carbeiter.h"

#include <string>
#include <sstream>
using namespace std;

// Konstruktor
```

```
CArbeiter::CArbeiter(  int alter, string name,
                       int stunden, double stundenLohn )
          // Kunstruktor der Oberklasse aufrufen
          : CPerson(alter, name)
{
     this->stunden= stunden;    // this ist hier notwendig
     this->stundenLohn = stundenLohn;
}



// Zugriffsmethoden
int CArbeiter::getAlter() const
{
     // return alter;
     // ist NICHT möglich, weil in Oberklasse private

     // return getAlter();
     // FEHLER; rekursiver Aufruf

     return CPerson::getAlter();
}

string CArbeiter::toString()const
{
    ostringstream out;

    out << CPerson::toString();

    out << "\nSTUNDEN: ";
    out <<  stunden;
    out << "\nSTUNDENLOHN: " ;
    out << stundenLohn;
    out << "\n";

    return out.str();
}


double CArbeiter::getGehalt() const
{
    return stunden * stundenLohn;
}
```

main.cpp

```
// a.hofmann 2007
// testcmitarbeiter.cpp
// Vererbung
// CPERSON
```

```
//     CARBEITER

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

int main(int argc, char *argv[]){
    // Objekt ist eine Zeigervariable
     CArbeiter* ich;

     ich= new CArbeiter(80, "Heinz Altermann", 40, 50.0);

     cout<< "\n*****************************************" << endl;;
     cout<< ich->toString()<< endl;

     cout<< "Mein Alter: " << ich->getAlter()<< endl;;
     cout<< "Mein Lohn: " << ich->getGehalt()<< endl;;


     cout<< "\n*****************************************" << endl;;
    // Objekt ist keine Zeigervariable
     CArbeiter du(20, "Karl Jungmann", 20, 25.0);

     cout<< du.toString()<< endl;

     cout<< "Dein Alter: " << du.getAlter()<< endl;;
     cout<< "Dein Lohn: " << du.getGehalt()<< endl;;


     cout << endl<<endl;

     delete ich;

    return EXIT_SUCCESS;
}
```

1.11.5. Projekt: oop1-Firma – Die Unterklassen CVerkaeufer, CManager

☑ Klasse: CVerkaeufer ist ein Arbeiter
Bringen Sie das obiges Programm zum Laufen und fügen Sie noch die Klasse CVerkaeufer hinzu:

Member:
int anzahlVerkaeufe;
double verkaufsKommission;

Methoden:
Konstruktor
getGehalt()
toString()
getAlter() //muss man diese Methode wirklich hinzufügen?

Erweitern Sie das obige Testprogramm entsprechend

☑ Klasse: CManager ist eine Person
Fügen Sie auch noch die Klasse CManager hinzu

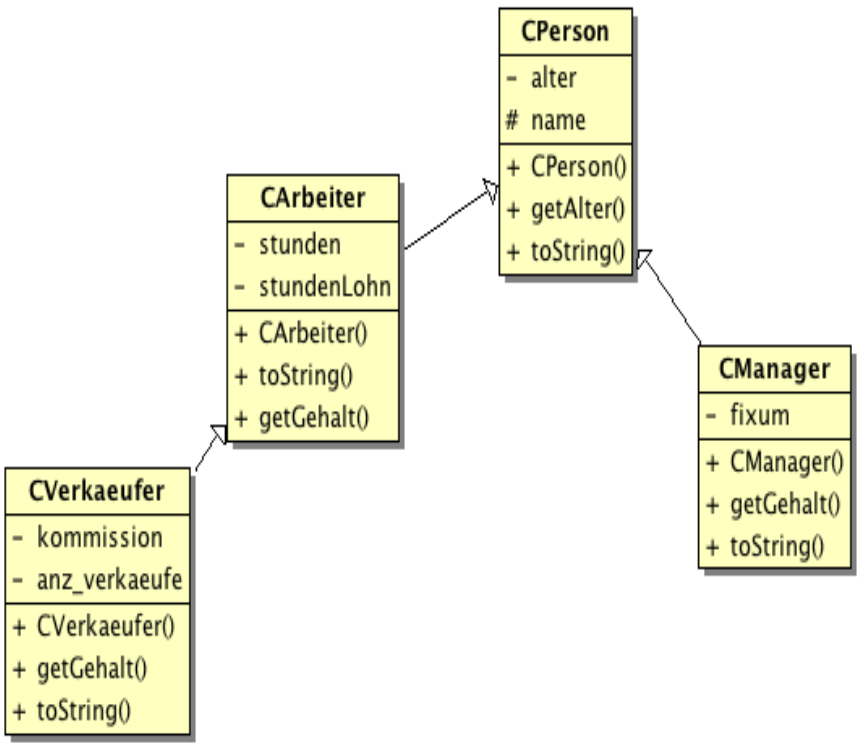
Member:
double fixum;

Methoden:
Konstruktor
getGehalt()
toString()
getAlter() //muss man diese Methode wirklich hinzufügen

1.12. Polymorphismus (Late Binding)

IST-Zustand: Projekt: oop1-Firma

Vererbungshierarchie



1.12.1. Projekt: oop1-Firma - Die Klasse CAbteilung ist ein Container

Wir wollen nun in unserem Projekt alle Personen zentral verwalten können. Zum Beispiel wollen wir von allen Personen das Gehalt berechnen können.

Dazu erstellen wir die

☑ Klasse CAbteilung, die
- ☐ alle Arten von Mitarbeitern (Arbeiter, Verkäufer, Manager, …) verwalten kann
- ☐ das gesamtGehalt, das für alle Mitarbeiter aufzubringen ist, berechnet.

☑ Lösungsansatz 1: (falsch)
- ☐ Einen vector für alle Arbeiterobjekte: vector<CArbeiter> arbeiter;
- ☐ Einen vector für alle Verkäuferobjekte: vector<CVerkaeufer> verkaeufer;
- ☐ usw.

d.h. Wenn es eine neuen Angestellten-Typ (zB: CProgrammierer) gibt, muss man die Klasse CAbteilung verändern. DAS IST „NOT SO GOOD“

Deshalb hier die
einzig richtige Lösung, die man als Anwendung des Polymorphismus kennt.

☑ Lösungsansatz 2: (richtig)
- ☐ **NUR EINEN** vector **für ALLE** Personentypen durch Verwendung des sogenannten BASE-CLASS-POINTER:

vector<CPerson*> mitarbeiter;

TODO:

☑ Erstellen Sie die Klasse CAbteilung:
☑ und erstellen Sie ein Testprogramm, dass
☑ einen Arbeiter, Manager,Verkäufer erzeugt
☑ eine Abteilung erzeugt
☑ die Arbeiter, Manager,Verkäufer zur Abteilung hinzufügt und
☑ die Methode toString() vom Abteilungsobjekt aufruft.

☑ toString() gibt alle Mitarbeiter aus und
☑ zum Schluss das zu bezahlende GesamtGehalt aller Mitarbeiter.

Hier ist die Datei abteilung.h

```
#ifndef CABTEILUNG_H
#define CABTEILUNG_H

#include <vector>
using namespace std;
```

```
#include "cperson.h"

class CAbteilung {

private:
    vector<CPerson*>  mitarbeiter;
    string name;

public:
    CAbteilung (string name);

    void addMitarbeiter(CPerson* p);
    void removeMitarbeiter(int alter);

    double getGesamtGehalt() const;

    string toString() const;
};

#endif // CABTEILUNG_H
```

Hinweis:
Um die obige Aufgabe wirklich lösen zu können, müssen Sie folgendes noch tun/wissen.

☑ Die Klasse **CPerson** braucht noch die Methode double **getGehalt**(), die zunächst 0.0 zurückgibt.
☑ Da im member mitarbeiter nur Zeiger auf die Oberklasse CPerson sind und wir
aber in Wirklichkeit darin die Adressen von Arbeiterobjekten, Verkäuferobjekten, …
speichern, muss **zur Laufzeit** des Programmes ermittelt werden, welches getGehalt()
tatsächlich aufzurufen ist. Dies wird durch das sogenannte **späte/Late Binding** realsiert.

1.12.2. Das Schlüsselwort virtual

☑ **Methoden werden durch die Angabe virtual zur Laufzeit des Programmes gebunden**.
d.h. Zur Laufzeit wird erst ermittelt welche virtuelle Methode tatsächlich aufgerufen werden soll.

☑ Die **abstrakte** Klasse CPerson
enthält u.a. folg. Virtuelle Methode, die von den Unterklassen überschrieben werden muss.

```
...
virtual double getGehalt() const =0;

...
```

Dadurch wird die Klasse CPerson eine **ABSTRAKTE KLASSE**:
d.h. Von ihr werden **keine Objekte** erzeugt, sondern sie dient dazu

☑ das gemeinsame Verhalten/Eigenschaften für ihre Unterklassen festzulegen
☑ als BASE-CLASS-POINTER kann sie verwendet werden.

Die Methoden getGehalt() wird in allen Unterklassen ja überschrieben (s.o.). Diese Methoden müssen nicht mehr virtual definiert werden, weil sie diese Eigenschaft ja bereits von der Oberklasse CPerson erbt.

1.12.3. Projekt: oop1-Firma – Die Klasse CPerson als abstrakte Klasse (virtual)

☑ Machen Sie aus der Klasse CPerson eine abstrakte Klasse:
virtual double getGehalt() const =0;

☑ Frage: sollte toString() auch virtual definiert werden?

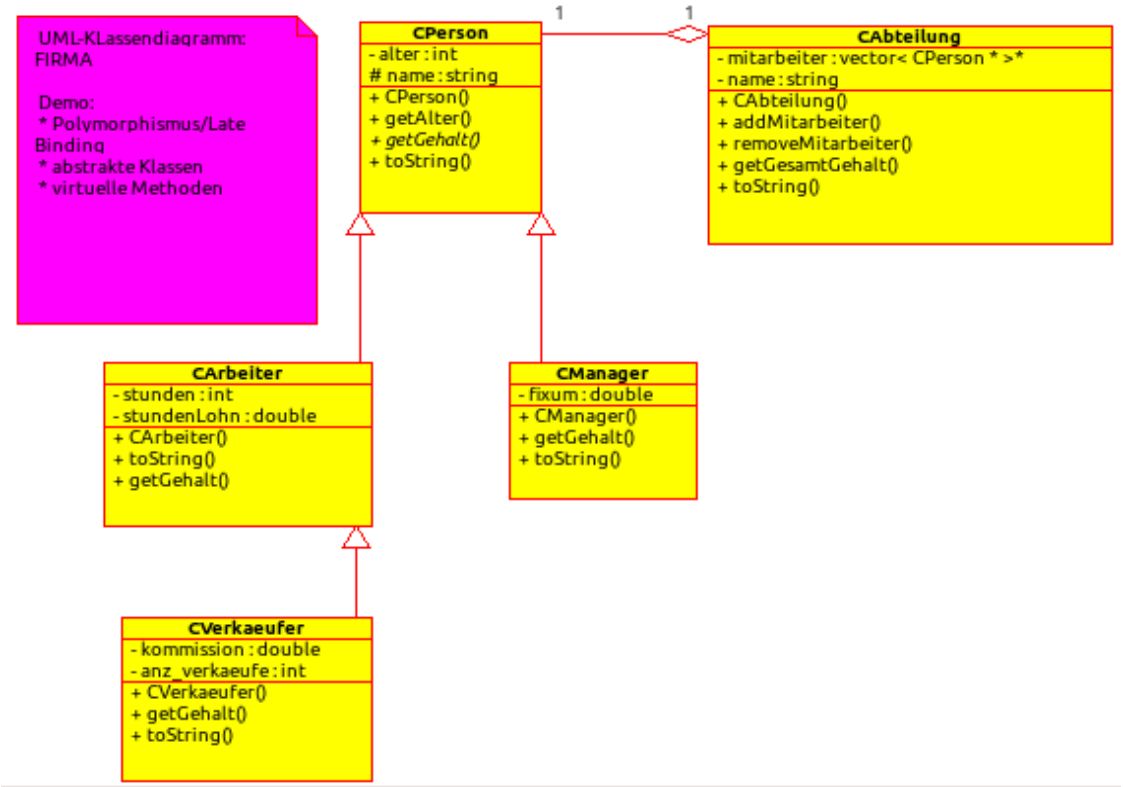
☑ Passen Sie eventuell alle weiteren Klassen aus dem Projekt CFIRMA an:
CPerson
CArbeiter
CVerkauefer
CManager

CAbteilung

☑ schreiben Sie ein Testprogramm: main.cpp
Erzeugt: Arbeiterobjekte, Verkäuferobjekte, Managerobjekte
Erzeugt: ein Abteilungsobjekt
Fügt: Arbeiterobjekte, Verkäuferobjekte, Managerobjekte in das Abteilungsobjekt ein
Ruft: vom Abteilungsobjekt die Methode getGesamtGehalt() auf.

1.12.4. Aufgabe: UML-Klassendiagramm: CFIRMA

Fügen Sie hier das Klassendiagramm ein.



1.12.5. Virtuelle Destruktoren?

Annahmen:
☑ Es gibt eine Vererbungshierarchie.
☑ Man verwendet einen Container mit BASE-CLASS-POINTERN, um verschiedene Unterklassenobjekte verarbeiten zu können.
☑ Die Oberklasse ist eine ABSTRAKTE KLASSE, d.h. Sie hat virtuelle Methoden.
☑ Die Unterklasse verwenden in ihren Konstruktoren new, weil sie zusätzlich Speicher brauchen.

```
CPerson* base;

CProgrammierer* ich= new CProgrammierer(“Max Mustermann“, 40, 1000.0);
…

base= ich;

…
delete base;
```

☑ Frage:

Welcher Destruktor wird bei delete base aufgerufen?
☑ Antwort:
a) Destruktor der Klasse CPerson
b) Destruktor der Klasse CProgrammierer

Das Problem ist, wenn a) der Destruktor der Oberklasse aufgerufen wird, dann wird der vom Konstruktor der Unterklasse (CProgrammierer) alloziierte Speicher nicht freigegeben.

☑ LÖSUNG:
Der Destruktor MUSS virtual definiert sein.

cperson.h

```
virtual ~CPerson();
```

1.13. Weitere Übungen

Bearbeiten Sie die Übungen aus dem Ordner 02-uebungen

1.14. Fragen/Test

Siehe Ordner 03-wissen

1.15. Ausblick

Wir wollen nun die Möglichkeiten der dynamischen Speicherverwaltung in C++ kennenlernen.