Wintersemester 2018/19		Zahl der Blätter: Blatt Nr:	17 1
Fachbereich:	Informationstechnik	Semester:	SWB/TIB/2
Prüfungsfach:	OOS 1	Prüfungsnr:.	1052027
Hilfsmittel:	keine elektronischen Hilfsmittel	Zeit:	90 min
Name:		Matrikel-Nr.:	

Hinweis: Der auf den Blättern jeweils freigelassene Raum reicht im Allgemeinen vollständig für die stichwortartige Beantwortung der Fragen, bzw. für die Lösungen aus. Tragen Sie daher auf **jedem** Blatt Ihren Namen und Ihre Matrikelnummer ein und nutzen Sie diese Blätter zur Abgabe Ihrer Antworten und Lösungen.

Aufgabe 1: Allgemeine Fragen (ca. 20 Min.)

Bitte beurteilen Sie die folgenden allgemeinen Aussagen. Machen Sie jeweils ein Kreuzchen in der Spalte „wahr“ oder „falsch“. Begründen Sie jeweils Ihre Wahl.

Aussage	wahr	falsch
Bei der folgende Anweisung in Zeile 2 wird der Zuweisungsoperator aufgerufen: **1 Person erika;** **2 Person markus = erika** **Begründung:**		
Strings von der c++ String Bibliothek sind im Vergleich zu C-String einfacher zu handhaben, da sie nützliche Funktionen, Methoden und Operationen anbieten um mit Zeichenketten zu arbeiten. **Begründung:**		
Die Zugriffsmodifizierer **protected** und **private** kapseln alle **public**-Elemente der Basisklasse. **Begründung:**		

Aussage	wahr	falsch
Der Datentyp der Variablen charArray1 und charArray2 ist EIGENESCHAR **1 typedef char* EIGENESCHAR;** **2 EIGENESCHAR charArray1, charArray2;** **Begründung:**		

Aussage	wahr	falsch
Objekte einer polymorphen Klasse ohne Attribute besitzen keinen Zeiger auf die VMT. **Begründung:**		
Ist in einer Klasse ein einziger Konstruktor definiert, der kein Default-Konstruktor ist, so kann der Compiler den Default-Konstruktor nicht mehr aufrufen. **Begründung:**		
Klassenmethoden haben kein this-Objekt. **Begründung:**		

Aussage	wahr	falsch
Im Destruktor einer abgeleiteten Klasse wird als erstes der Destruktor der Basisklasse aufgerufen. **Begründung:**		
Die Initialisierungsliste muss verwendet werden für eingebettete Objekte, für die ein parametrisierter Konstruktor aufgerufen werden soll. **Begründung:**		
Eine Klasse enthalte eine konstante und eine nicht konstante Version einer Methode. Die nicht konstante Version kann dann als überladene Version der ersteren angesehen werden. **Begründung:**		

Aufgabe 2: Klassendefinition (ca. 20 Min.)

ACHTUNG: Lesen Sie die folgende Aufgabe KOMPLETT durch, bevor Sie mit der Lösung beginnen. Bitte trennen Sie **Deklaration** und **Implementierung** der Klassen.

Die Teilaufgaben von Aufgabe 2 können auch unabhängig voneinander gelöst werden.

Im Folgenden sollen nun Klassen für eine Bankverwaltung definiert werden. Folgende Abbildung zeigt das dazugehörende Klassendiagramm.

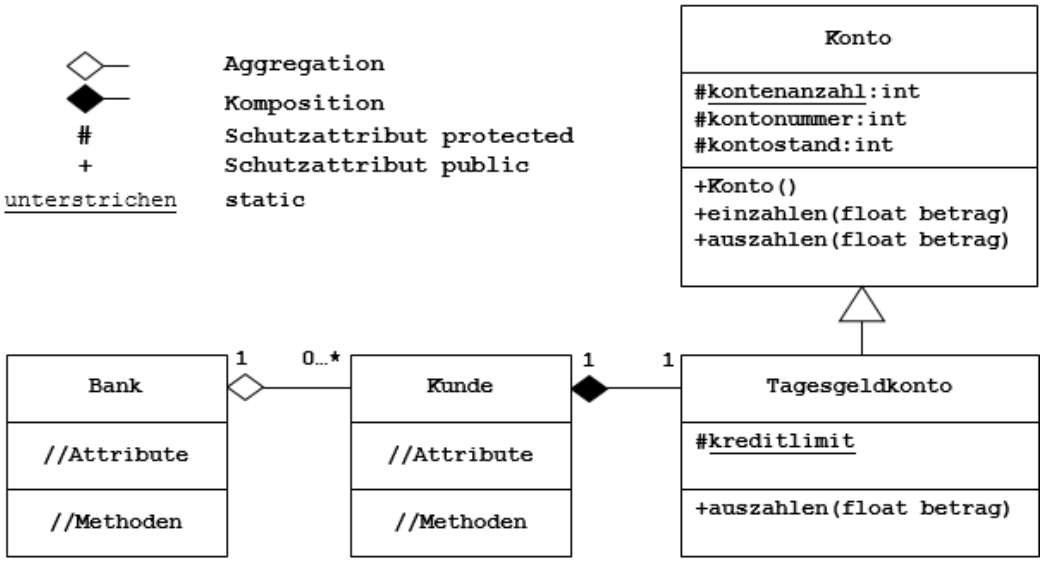

Abbildung 1: Klassendiagramm – Bankverwaltung

2.1 Eine Klasse **`Bank`** mit folgenden Elementen soll deklariert werden.

a) eine Instanzvariable **`name`** vom Typ **`string`**, die die Bezeichnung der Bank speichert (z.B. „Commerzbank“),

vector<Kunden*>Kunden

b) ein **VectorArray** mit dem Namen **`kunden`**, der Pointer auf **Kunden** verwaltet,

c) einen Konstruktor, der die Bezeichnung der Bank übergeben bekommt,

d) eine Instanzmethode **kundeAnlegen**, die als Parameter einen Pointer auf einen Kunden bekommt und einen Kunden zum **VectorArray** hinzufügt, falls dieser noch nicht existiert (sonst darf ein Kunde nur einmal existieren!). Die Methode hat einen Rückgabewert **bool**,

2.2 Eine Klasse **`Kunde`** mit folgenden Elementen soll deklariert werden.

a) eine Instanzvariable **`kundennummer`** vom Typ **`int`**, die die Kundennummer des Kunden speichert,

b) Ein Pointer als Instanzvariable **pTagesgeldkonto**, der auf ein **Tagesgeldkonto** zeigt,

c) eine Instanzvariable **`name`** vom Typ **`string`**, die den Namen (Vor- und Nachnamen) des Kunden speichert,

d) einen **Konstruktor**, der als Parameter die Kundennummer und den Namen des Kunden übergeben bekommt,

e) eine konstante Instanzmethode **kontozugriff**, die als Rückgabewerte den Pointer auf das Tagesgeldkonto des Kunden zurückgibt,

f) einen **`Operator==`** mit dem zwei Kunden anhand ihrer Kundennummer verglichen werden können (z.B. kunde1 == kunde2),

g) einen Destruktor, der den Speicher auf **pTagesgeldkonto** wieder freigibt

2.3 Eine Klasse **`Konto`** mit folgenden Elementen soll deklariert werden.

a) eine Klassenvariable **kontoanzahl** vom Typ **int**, die die Anzahl aller Konten-Objekte festhält,

b) eine Instanzvariable **kontostand** vom Typ **float**, die den aktuellen Kontostand des Kontos festhält. Der Startwert des Kontostandes ist 50.0,

c) eine konstante Instanzvariable **kontonummer** vom Typ **int**, die eindeutig ist und dessen Wert sich aus der Summe der momentanen Anzahl aller Konten-Objekte + 5000 zusammensetzt,

d) eine Instanzmethode **`einzahlen`,** die als Parameter einen **float** mit dem Namen **betrag** übergeben bekommt und den Kontostand um diesen Betrag erhöht. Einen Rückgabewert der Methode gibt es nicht,

e) eine Instanzmethode **auszahlen**, die als Parameter einen **float** mit dem Namen **betrag** übergeben bekommt und den Kontostand um diesen Betrag erniedrigt. Einen Rückgabewert der Methode gibt es nicht,

2.4 Eine Klasse **Tagesgeldkonto**, die von der Klasse Konto erbt, soll mit folgenden Elementen deklariert werden.

a) eine Klassenvariable **kreditlimit** vom Typ **float**, die das allgemeine Kreditlimit für alle Tagesgeldkonto-Objekte speichert,

b) eine Instanzmethode **auszahlen**, die die Instanzmethode **auszahlen** der Klasse Konto überschreibt. Es wird erst ein Betrag ausgezahlt, wenn die Endsumme das Kreditlimit nicht übersteigt. Der Rückgabewert ist **true** oder **false.**

2.5 Eine Funktion **`void testBankverwaltung()`** mit folgenden Eigenschaften:

a) eine Bank wird angelegt,

b) drei (!) Kunden werden angelegt,

c) die angelegten Kunden, werden der Bank hinzugefügt,

d) Einzahlung/Auszahlung der Kunden von:

(1) Kunde 1: einzahlen von 100.0

(2) Kunde 2: auszahlen von 1200.0

(3) Kunde 3: auszahlen von 2500.0

Ergänzen Sie das folgende Programmgerippe. Schützen Sie die Datenelemente vor Zugriffen durch klassenfremde Methoden; erlauben Sie aber abgeleiteten Klassen den Zugriff. Verwenden Sie – falls möglich – konstante Methoden.

Trennen Sie **Header** (Aufgabe 2) und **Implementierung** (Aufgabe 3).

Prototypen der Klassen (Aufgabe 2)

```
#include<iostream>
#include<string>
#include<vector>
#pragma once
using namespace std;
```

Klassendeklaration der Klasse Bank

```
class Bank {
// Instanzvariablen
String name;
vector<Kunden*> Kunden


// Konstruktor
public:
Bank(string_name);


// Methoden
bool kundenAnlegen(Kunden* kunden);



};
```

//include

```
#include <iostream>
#include <string>
#include <vector>
#pragma once
using namespace std;
```

Klassendeklaration der Klasse Kunde

```
class Kunde {
// Instanz- und Klassenvariablen

int kundennummer;
String name;
Tagesgeldkonto* pTagesgeldkonto;


// Konstruktor
public:
Kunde(string _name, int _kundennummer);



//Destruktor
~Kunde();



// Methoden
Tagesgeldkonto* kontozugriff() const;
Kunde operator==();


};
```

//include

```
#include <iostream>
#include <string>
#include <vector>
#pragma once
using namespace std;
```

Klassendeklaration der Klasse Konto

```
class Konto {
// Instanzvariablen
protected:
static int kontoanzahl;
float kontostand = 500 ;
const int kontonummer = kontoanzahl + 5000;




// Methoden
public:
void einzahlen(float betrag);
void auszahlen(float betrag);

Konto();




};
```

Klassendekl. Tagesgeldkonto

```
class Tagesgeldkonto: public Konto{
protected:
  static float kreditlimit;
public:
 bool auszahlen(float betrag) const override;
```

Aufgabe 3: Implementierung der Methoden der Klassen (ca. 22 min)

Programmieren Sie bitte hier und auf den folgenden Seiten außerhalb der Klassen **`Bank, Kunde`**, **`Konto`** und **`Tagesgeldkonto`** die angegebenen Elemente aus:

`//include`

```
#include <iostream>
#include <string>
#include <vector>
#progma once
using namespace std;
```

Klassendefinition der Klasse Bank

```
// Bank Konstruktor




// Bank Methoden
```

```
//Include
```

Klassendefinition der Klasse Kunde

```
// Kunde Konstruktor




// Kunde Destruktor




// Kunde Methoden
```

```
//Include
```

Klassendefinition der Klasse Konto

```
// Konto Methoden
```

```
//Include
```

Klassendefinition der Klasse Tagesgeldkonto

```
//Tagesgeldkonto Methoden
```

Implementierung der Methode testBankverwaltung

```
void testBankverwaltung () {

// Legen Sie eine Bank an mit dem Namen „Commerzbank“
   Bank* Commerzbank= new Bank ("Commerzbank" );

// Legen Sie 3 Kunden an
      // Kunde 1: Kundennummer: 3001 / Name: Markus Muster
      // Kunde 2: Kundennummer: 3002 / Name: Erika Muster
      // Kunde 3: Kundennummer: 3003 / Name: Helene Fischer
         Kunde* markus = new Kunde(3001, "Markus Muster");
         Kunde* erika = new Kunde(3002, "Erika Muster");
         Kunde* helene = new Kunde(3003, "Helene Fischer");



// Fügen Sie die Kunden der Bank hinzu
   bank.kundeAnlegen ("Markus Muster");
   bank.kundeAnlegen("Erika Muster");
   bank. kundeAnlegen ("Helene Fischer");

// Machen sie folgende Ein-und Auszahlungen
      //Kunde 1: einzahlen von 100.0
      //Kunde 2: auszahlen von 1200.0
      //Kunde 3: auszahlen von 2500.0

   Markus -> kontozugriff() -> einzahlen (100.0);
   erika -> kontozugriff() -> einzahlen (1200.0);
```

Aufgabe 4: Polymorphie (ca. 11 min)

Analysieren Sie die nachfolgende Funktion **`main`** und schreiben Sie die Ausgabe der Funktion **direkt hinter die Zeile**.

```
class Eins {
public:
    1 void x() { a(); y(90.0), z(9000.0); }
    2 virtual void a() { cout << "Eins::a()" << endl; }
    3 void y(int i) { cout << "Eins::y(" << i << ")" << endl; }
    4 void z(int i) { cout << "Eins::z(" << i << ")" << endl; }
};

class Zwei: public Eins {
public:
    5 void x() { a(); y(90.0), z(9000.0); }
    6 void a() const { cout << "Zwei::a()" << endl; }
    7 void y(int i) const { cout << "Zwei::y(" << i << ")" << endl; }
    8 void z(double i) { cout << "Zwei::z(" << i << ")" << endl; }
};

class Drei : public Zwei {
public:
    9 void x() { a(); y(90.0), z(9000.0); }
   10 void a() { cout << "Drei::a()" << endl; }
   11 virtual void y(int i) { cout << "Drei::y(" << i << ")" << endl; }
   12 void z(int i) { cout << "Drei::z(" << i << ")" << endl; }
};

int main()
{
      Zwei zwei;
      Drei drei;

      Eins * peins = new Eins;
      Eins * pzwei = new Zwei;
      Eins * pdrei = new Drei;

      zwei.a();
      pzwei->a();
      pdrei->a();
      drei.y(12);
      drei.x();
      peins->y(10);
      pdrei->y(12);
      peins->z(1000);
      pdrei->z(1002);
      pdrei->x();
}
```

Drei::y(<< 12 <<")

Aufgabe 5: Ausnahmen (ca. 15 Min.)

Gegeben ist eine Klasse **Auto** mit der Methode **motorStarten**. Diese überprüft vorhandene interne Fehler-Attribute (**unbekannterFehler**, **keinBenzinFehler**, **motorFehler** und **motorElektronikFehler**) vom Typ **bool**. Falls ein Fehlerattribut auf **true** gesetzt ist, wird eine entsprechende Exception geworfen. Ein Mehrfachauftreten von Fehlern ist hier nicht berücksichtigt.

```
class Auto {
private:
      bool unbekannterFehler;
      bool keinBenzinFehler;
      bool motorFehler;
      bool motorelektronikFehler;
public:
      …
      Bool motorStarten() {
            if (motorFehler) {
                  throw MotorException(200);
            }
            if (motorelektronikFehler) {
                  throw MotorElektronikException(201);
            }
            if (keinBenzinFehler) {
                  throw "Kein Benzin! ";
            }
            if (unbekannterFehler) {
                  throw -1;
            }
      }
};
```

Es existiert eine eigene Exceptionklasse **MotorException**, die von der Klasse **Exception** erbt. Die **MotorException**-Klasse überschreibt die **what-**Methode und hat zusätzlich eine Instanzvariable **errorCode** vom Typ **int**, die über die Methode **getErrorCode** ausgelesen werden kann.

```
class MotorException: public exception {
private:
      int errorCode;
public:
      MotorException(int errorCode):errorCode(errorCode){}
      const char * what() const throw() {
            return "Motorfehler";
      }
      ostream& writeErrorCode() {
            return cout << " Error code: " << errorCode << endl;
      }
};
```

Implementieren Sie folgende Zusätze (Eine **Trennung** von Deklaration und Implementierung ist **NICHT** notwendig):

5.1 Die Exceptionklasse **MotorElektronikException**, die von der Klasse **MotorException** erbt. Die **what**-Methode soll den Text „Motorenelektronikfehler“ zurückliefern.

```
class ElektronikException:public MotorException{
ElektronikException(int _errorcode):errorCode(_errorCode){};

const char* what() const throw(){
  return "Motorenelektronikfehler"
}
}
```

5.2 In der **testProgramm**-Methode erzeugen Sie ein Auto und rufen die **motorStarten**-Methode auf. Fangen Sie alle möglichen Fehler über **try/catch** ab und geben Sie Folgendes je nach Exceptionart auf der Konsole aus.

a) Unbekannter Fehler: **„Unbekannter Fehler“**

b) MotorException: **„Motorfehler / Error code: 200“**

c) MotorElektronikException: **„Motorelektronikfehler / Error code: 201“**

d) Kein Benzin mehr: Fehler: „**Kein Benzin!“**

```
testProgramm(){









}
```