Sommersemester 2023		Seite	1 von 15
Fachbereich:	Informationstechnik	Studiengang:	TIB, SWB, IEP
Prüfungsfach:	OOS1	Prüfungsnummer.:	1052027
Hilfsmittel:	handschriftliche Notizen (auch gedruckt) 2 Blätter DIN A4 beidseitig	Zeit:	90 min
Nachname: Vorname:		Matrikelnummer:	

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.)

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
Referenzvariablen kann man als konstante Pointer ansehen. Begründung:		
Klassenmethoden können nur mit Objekten der Klasse arbeiten, wenn diese als Parameter übergeben werden. Begründung:		
Jeder Operator kann sowohl als Methode der Klasse oder als (globale) Funktion realisiert werden. Begründung:		
Parametrisierte Konstruktoraufrufe von Basisklassen sind nur in der Initialisierungsliste möglich. Begründung:		

Aussage	wahr	falsch
Ein Upcast sollte immer durch einen dynamischen Cast erfolgen, damit eventuelle Fehler abgefangen werden. Begründung:		
Wird der Destruktor einer abgeleiteten Klasse aufgerufen, ruft dieser als erstes den Destruktor der Basisklasse auf. Begründung:		
Wird eine Methode in einer abgeleiteten Klasse überschrieben/redefiniert, dann spricht man von Polymorphie. Begründung		
In einer abstrakten Klasse dürfen nur abstrakte Methoden vorkommen. Begründung:		
Eine geworfene Exception kann an einen Exception-Handler by-value oder by-reference übergeben werden. Begründung:		
Für das Fehlerhandling müssen spezielle, von der Standardexception abgeleitete Fehlerklassen verwendet werden. Begründung:		

Aufgabe 2: Klassen, Attribute, Methoden, Operatoren (35 min)

Die Klasse `Fahrzeug` ist die Basisklasse aller Fahrzeugtypen, die in einer Fuhrpark-Verwaltung unterstützt werden (vgl. dazu auch Aufgabe 3). Sie soll die folgenden Anforderungen realisieren:

a) Die konstante Instanzvariable `_id` vom Typ `int` hält die eindeutige ID für das Fahrzeug fest.
b) Die Instanzvariable `_ist_gebucht` vom Typ `bool` zeigt an, ob das Fahrzeug gebucht wurde oder nicht.
c) Die Klassenvariable `_anzahl` vom Typ `int` speichert die aktuelle Anzahl an Fahrzeugen.
d) Ein Konstruktor bekommt eine ID für das Fahrzeug übergeben und speichert diese in der Instanzvariablen `_id` ab. Die Klassenvariable `_anzahl` wird inkrementiert.
e) Eine Instanzmethode `buche` bucht das Fahrzeug.
f) Die Instanzmethode `ist_gebucht` gibt in Form des Rückgabewertes vom Typ `bool` zurück, ob das Fahrzeug gebucht ist oder nicht.
g) Die Instanzmethode `get_id` gibt die ID des Fahrzeugs als `int` zurück.
h) Die statische Methode `get_anzahl` gibt die aktuelle Anzahl an Fahrzeugen als `int` zurück.
i) Der `operator==` vergleicht zwei Fahrzeuge anhand Ihrer `_id` und gibt in Form des Rückgabewertes vom Typ `bool` zurück, ob die Fahrzeuge identisch sind oder nicht.
j) Die virtuelle Instanzmethode `print` gibt Informationen über das Fahrzeug auf der Konsole aus. Für ein nicht gebuchtes Fahrzeug mit der ID 1 erfolgt die folgende Ausgabe::

```
ID: 1
Ist gebucht: 0
```

Die Klasse `Fuhrpark` wird in einer Anwendung für eine Fuhrpark-Verwaltung benötigt (vgl. dazu auch Aufgaben 3 und 4).
Die Klasse Fuhrpark soll die im Folgenden genannten Anforderungen realisieren:

a) Die Instanzvariable `_anzahl_fahrzeuge` vom Typ `int` hält die Anzahl der im Fuhrpark vorhandenen Fahrzeuge fest.
b) Die Instanzvariable `_name` vom Typ `string` speichert die Bezeichnung des Fuhrparks.
c) Der `vector` mit dem Namen `_fahrzeuge` verwaltet Pointer auf Fahrzeuge
d) Der Konstruktor bekommt die Bezeichnung des Fuhrparks übergeben und speichert ihn in der Instanzvariablen `_name`
e) Die Instanzmethode `fuege_hinzu_fahrzeug` bekommt als Eingabeparameter einen Pointer auf ein Fahrzeug übergeben und fügt das Fahrzeug dem vector `_fahrzeuge` hinzu, unter der Voraussetzung, dass dieses Fahrzeug noch nicht im vector enthalten ist. Sind alle verfügbaren Fahrzeuge dem Fuhrpark hinzugefügt wird die Meldung „`Alle existierenden Fahrzeuge wurden dem Fuhrpark hinzugefügt.`“ auf der Konsole ausgegeben.
f) Die Instanzmethode `entferne_fahrzeug` bekommt als Parameter einen Pointer auf ein Fahrzeug übergeben und löscht dieses Fahrzeug aus dem vector falls es existiert und es nicht gebucht ist.
g) Die Instanzmethode `buche_fahrzeug` bekommt als Parameter einen Pointer auf ein Fahrzeug übergeben und bucht dieses Fahrzeug. Definieren Sie Für diese Methode nur den Prototyp in der Headerdatei. **Erst in Aufgabe 4 wird die Methode implementiert**.

h) Die Instanzmethode `print` gibt die Informationen über den Fuhrpark auf der Konsole aus. Für einen Fuhrpark mit zwei Fahrzeugen erfolgt z.B. die folgende Ausgabe:

```
Anzahl Fahrzeuge: 2

ID: 1
Ist gebucht: 0
ID: 2
Ist gebucht: 1
```

Ergänzen Sie die folgenden Programmgerippe. Schützen Sie die Datenelemente vor Zugriff durch klassenfremde Methoden; erlauben Sie aber abgeleiteten Klassen den Zugriff. Verwenden Sie – falls möglich – konstante Methoden.
Trennen sie die Umsetzung sinnvoll in Headerdateien und Implementierungsdateien.

`Fahrzeug.h`

```
// alles was für die Klassendeklaration benötigt wird
```

Klassendeklaration der Klasse `Fahrzeug`

```
// Instanzvariablen



// Konstruktor, Methoden und Operator
```

Fuhrpark.h

```
// alles was für die Klassendeklaration benötigt wird
```

Klassendeklaration der Klasse Fuhrpark

```
// Instanzvariablen



// Konstruktor und Methoden
```

Fahrzeug.cpp

```
// alles was für die Klassendefinition benötigt wird
```

Klassendefinition der Klasse Fahrzeug

```
// Konstruktor



// Methoden und Operator
```

Fuhrpark.cpp

```
// alles was für die Klassendefinition benötigt wird
```

Klassendefinition der Klasse Fuhrpark

```
// Konstruktor



// Methoden
```

Aufgabe 3: Vererbung, Verwendung der Klassen (14 min)

Die Klassen `Elektro` und `Verbrenner` sind Konkretisierungen der Klasse `Fahrzeug`.
Leiten Sie die Klasse `Elektro` von der Klasse `Fahrzeug` ab. Sie soll folgende Anforderungen erfüllen:

a) Die Instanzvariable `_batterie` vom Typ `int` hält den Batteriefüllstand in % fest.
b) Ein parametrisierter Konstruktor bekommt die ID für die Basisklasse Fahrzeug und den Batteriestatus übergeben und setzt die Werte entsprechend.
c) Die Instanzmethode `print` gibt Informationen über die Batterie und die Art des Fahrzeugs auf der Konsole aus. Bei einem nicht gebuchten Elektrofahrzeug mit der ID 1 und 90% Füllstand ergibt sich die folgende Ausgabe:

```
ID: 1
Ist gebucht: 0
Art: Elektro
Batterie 90%
```

Die Klasse Verbrenner muss **nicht** implementiert werden.

Implementieren Sie eine Funktion `teste_fuhrpark` mit folgenden Eigenschaften:

a) ein Fuhrpark mit dem Namen „Musterfirma“ wird angelegt,
b) zwei (!) Fahrzeuge werden angelegt, mit folgenden Eigenschaften:
    a. Typ = Verbrenner / ID = 1000 / Tankfüllung = 80 Liter
    b. Typ = Elektro / ID = 1001 / Batterie = 100 %
c) die angelegten Fahrzeuge werden dem Fuhrpark hinzugefügt,
d) die Fahrzeuge mit folgenden IDs werden gebucht:
    a. 1000
    b. 1001
e) der Fuhrpark wird ausgegeben,
f) das Fahrzeug mit der folgenden ID wird entfernt:
    a. 1000

Ergänzen Sie die folgenden Programmgerippe. Verwenden Sie – falls möglich – konstante Methoden. Nutzen Sie die sprachlichen Möglichkeiten von C++, um den Compiler prüfen zu lassen, ob virtuelle Methoden korrekt überschrieben wurden.
Trennen sie die Umsetzung sinnvoll in Headerdateien und Implementierungsdateien.

Elektro.h

```
// alles was für die Klassendeklaration benötigt wird
```

Klassendeklaration der Klasse `Elektro`

```
class Elektro
{
// Konstruktor



// Konstruktor und Methode









}
```

Elektro.cpp

```
// alles was für die Klassendefinition benötigt wird
```

Klassendefinition der Klasse `Elektro`

```
// Konstruktor



// Methode
```

main.cpp

```
// alles was für die die Funktion teste_fuhrpark benötigt wird
```

```
void teste_fuhrpark()
{









}

int main()
{
    teste_fuhrpark();
    return 0;
}
```

Aufgabe 4: Exception Handling (11 min)

Implementieren Sie folgende Anforderungen:

a) In der Datei `BereitsGebuchtAusnahme.cpp` implementieren sie den Konstruktor und die `what`-Methode. Die zugehörige Headerdatei ist unten abgebildet. Für die ID 125 soll die `what`-Methode beispielsweise bei bereits gebuchtem Fahrzeug den folgenden Text auf der Konsole ausgeben:
„Das Fahrzeug mit der ID 125 ist bereits gebucht.“

b) In der Datei `Fuhrpark.cpp` (siehe Seite 12) nehmen Sie die Implementierung der Methode `buche_fahrzeug` vor. Das Fahrzeug kann nur gebucht werden, wenn es bisher nicht gebucht ist. Wenn es bereits gebucht ist, soll die Ausnahme `BereitsGebuchtAusnahme` geworfen werden.

a) Ergänzen Sie die Methode `teste_fuhrpark` (siehe Seite 13) so, dass alle möglichen Ausnahmen behandelt werden. Die Ausnahmebehandlung soll wie folgt funktionieren:
   - Tritt die Standardausnahme `exception` auf, wird eine Meldung mit dem folgenden Format auf der Konsole ausgegeben:
     „Standardausnahme: [Rückgabewert der what-Methode]“
   - Tritt die Ausnahme `BereitsGebuchtAusnahme` auf, wird der Rückgabewert der `what`-Methode auf der Konsole ausgegeben.
   - Tritt irgendeine Ausnahme (alle außer `exception` und `BereitsGebuchtAusnahme`) auf, wird die folgende Meldung auf der Konsole ausgegeben:
     „Eine unerwartete Ausnahme ist aufgetreten.“

BereitsGebuchtAusnahme.h

```
// alles was für die Klassendeklaration benötigt wird

#include<exception>
#include<string>
using namespace std;
```

Klassendeklaration der Klasse `BereitsGebuchtAusnahme`

```
class BereitsGebuchtAusnahme: public exception {
    string fehlernachricht;
public:
    BereitsGebuchtAusnahme(int id);
    const char* what() const noexcept override;
};
```

BereitsGebuchtAusnahme.cpp

```
// alles was für die Klassendefinition benötigt wird
```

Klassendefinition der Klasse BereitsGebuchtAusnahme

```
// Konstruktor



// Methode
```

In den folgenden Dateien Fuhrpark.cpp und main.cpp müssen Sie keine weiteren #includes angeben.

Fuhrpark.cpp

```
// buche_fahrzeug
```

main.cpp

```
// Alle benötigten includes vorhanden
void teste_fuhrpark()
{




// Hier steht Ihr Code aus Aufgabe 3.
// Sie müssen den Code nicht erneut aufschreiben.
```

```
}

int main()
{
    teste_fuhrpark();
    return 0;
}
```

Aufgabe 5: Polymorphie (10 min)

Analysieren Sie das nachfolgende Programm und schreiben Sie die Ausgabe des Programms unterhalb von „//Ausgabe:“.

```
#include <iostream>
#include <string>
using namespace std;
class A {
public:
    virtual void f() { cout << "A::f()->"; }
    void f(int i) { cout << "A::f(" << i << ")->"; }
    void g() { cout << "A::g()" << endl; }
    void g() const { cout << "A::g() const" << endl; }
    void h()
    {
        f();
        f(1);
        g();
    }
    void h() const { g(); }
};
class B : public A {
public:
    void f() override { cout << "B::f()->"; }
    void f(int i) { cout << "B::f(" << i << ")->"; }
    virtual void g() const { cout << "B::g() const" << endl; }
    void h()
    {
        f();
        f(2);
        g();
    }
    void h() const { g(); }
};
class C : public B {
public:
    void f() override { cout << "C::f()->"; }
    virtual void f(int i) { cout << "C::f(" << i << ")->"; }
    void g() { cout << "C::g()" << endl; }
    void g() const override { cout << "C::g() const" << endl; }
    void h()
    {
        f();
        f(3);
        g();
    }
    void h() const { g(); }
};
```

```
int main()
{
    C o_c;
    A *p_c = &o_c;
    B o_b;
    A *p_b = &o_b;
    A o_a;
    A *p_a = &o_a;
    const C o_c_const;
    const A *p_c_const = &o_c_const;
    o_c.h();
    o_b.h();
    o_a.h();
    o_c_const.h();
    p_c->h();
    p_b->h();
    p_a->h();
    p_c_const->h();
    return 0;
}
```

// Ausgabe