Wintersemester 2022/23		Seite	1 von 14
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
Die Initialisierung von konstanten Attributen kann sowohl in der Initialisierungsliste als auch im Block eines Konstruktors erfolgen. Begründung: Falsch. Konstante Attribute müssen in der Initialisierungsliste initialisiert werden, da Zuweisungen an Konstanten nicht erlaubt sind.		x
Im Konstruktor einer abgeleiteten Klasse wird als erstes der Defaultkonstruktor der Basisklasse aufgerufen. Begründung: Falsch. Der Defaultkonstruktor wird nur aufgerufen, wenn in der Initialisierungsliste kein parametrisierter Konstruktor aufgerufen wird.		x
Im folgenden Beispiel handelt es sich um die Aggregation eines Objektes der Klasse KFZ: `class TestKFZ { KFZ testling;...};` Begründung: Falsch. Aggregation bedeutet, dass die Teilobjekte auch ohne das Gesamtobjekt exisitieren können. Das ist hier nicht der Fall, das KFZ wird im Konstruktor erzeugt und im Destruktor entsorgt.		x
In der Klasse `B` verdeckt die Methode `f()` die geerbte Methode `f()` der Klasse `A`: `class A{public: int f() const;} class B: public A {public: int f();}`. Begründung: Richtig. Die beiden Funktionen unterscheiden sich zwar durch das const, da sie den gleichen Namen haben, ist die geerbte Methode verdeckt.	x	

Aussage	wahr	falsch
Beim Überschreiben einer virtuellen Methode muss der Rückgabetyp in der abgeleiteten Klasse exakt derselbe sein. Begründung: Falsch. Ist der Rückgabetyp vom Typ der Basisklasse, dann kann in der abgeleiteten Klasse der Typ der abgeleiteten Klasse stehen.		x
Im folgenden Beispiel wird das Attribut B auf den Wert 0 gesetzt: `class A { int B; public: virtual void setB(int B) = 0;}` Begründung: Falsch. Es handelt sich um die Definition einer abstrakten Methode		x
Der Zuweisungsoperator = kann nicht als const-Methode definiert werden. Begründung: Richtig. Das this-Objekt wird verändert	x	
Wird eine Exception in einem Block nicht gefangen und der Block somit verlassen, werden die mit new im Block erzeugten Objekte nicht gelöscht. Begründung: Richtig. Mit new erzeugte Objekte werden erst durch ein delete-Aufruf gelöscht.	x	
Bei der Instanziierung von Klassentemplates erzeugt der Compiler den Maschinencode für die Instanz. Begründung: Ja, der Maschinencode wird für eine Instanziierung der Typnamen erstellt	x	
Das Erbgut von virtuellen Basisklassen wird über Pointer adressiert. Begründung: Richtig. Dadurch kann das selbe Erbgut über mehrere Pointer angesprochen werden je nach Vererbungspfad	x	

Aufgabe 2: Klassen, Attribute, Methoden, Operatoren (30 min)

Die Klassen `Strecke` und `Route` werden für eine Routenplanung benötigt (vgl. dazu auch Aufgabe 3).
Die Klasse `Strecke` ist die Basisklasse für Strecken mit unterschiedlicher Fortbewegungsart und soll die im Folgenden genannten Anforderungen realisieren:

a) Eine Strecke hat die Attribute `startort` und `zielort`, jeweils vom Typ String, sowie das Attribut `entfernung`, das eine Gleitkommazahl speichert. Die Attribute sind nicht öffentlich zugänglich. Sie werden durch einen parametrisierten Konstruktor gesetzt und können danach nicht mehr geändert werden.
b) Die Klasse `Strecke` hat `get`-Methoden für alle Attribute. Sie müssen lediglich die get-Methode für das Attribut `startort` implementieren.
c) Sie hat eine Methode `toString()`, die zu einer Strecke einen String im folgenden Format zurückgibt: „Von <startort> nach <zielort>: <entfernung> km.“ An den mit <> markierten Stellen sind die Werte der Attribute des Objektes einzusetzen. Bei der Entfernung werden nur die notwendigen Nachkommastellen ausgegeben. Benutzen Sie in der Implementierung dieser Methode einen Stringstream. In der Beschreibung von Aufgabe 3 finden Sie ein Beispiel für die Ausgabe dieser Methode.
d) Die Methode `legeZurueck` ist eine rein virtuelle Methode. Sie wird für die Klasse `Strecke` nicht implementiert.

Die Klasse `Route` soll die folgenden Anforderungen realisieren:

a) Eine Route besteht aus mehreren Strecken. Verwalten Sie Pointer auf Strecken im Attribut `teilStrecken`. Verwenden sie hierzu einen geeigneten sequentiellen Container. Eine Route hat außerdem die Attribute `laenge` und `maxAnzahl`. Das Attribut `laenge` speichert die Länge der Strecke als Summe der Teilstrecken und das Attribut `maxAnzahl` speichert die maximal erlaubte Anzahl von Strecken einer Route.
b) Die Klasse Route hat einen parametrisierten Konstruktor, der die erste Strecke der Route und die maximale Anzahl von Strecken der Route übergeben bekommt, die entsprechenden Attribute mit den Werten belegt und die neue Länge berechnet und speichert.
c) Die Methode `fuegeSteckeHinzu` nimmt einen Pointer auf eine Strecke entgegen, speichert die übergebene Strecke hinter den bisher vorhandenen Teilstrecken der Route ab und berechnet die Länge der Strecke neu.
d) Durch die Methode toString() wird ein String in der folgenden Form zurückgegeben: „Die Route von <Startort der Route> nach <Zielort der Route> enthält <Anzahl der Teilstrecken> Strecken und hat die Länge <Länge der Route> km“. An den mit <> markierten Stellen sind entsprechende Werte aus Attributen des Objektes einzusetzen.
e) Die Methode `starten()` iteriert über die Teilstrecken und ruft für jede Teilstrecke die Methode `legeZurueck()` auf.
f) Als Alternative zur Methode `fuegeStreckeHinzu` definiert die Klasse einen Operator += als Methode. Dieser speichert den übergebenen Pointer auf eine Strecke (2. Operand) hinter den bisher vorhandenen Teilstrecken der Route ab, berechnet die Länge der Strecke neu und gibt eine Referenz auf die Route zurück.
g) Der Operator << dient zur Ausgabe der Informationen einer Route in der String-Form (siehe Beschreibung der Methode toString() oben) in einer Zeile. In der Beschreibung von Aufgabe 3 finden Sie ein Beispiel für die Ausgabe dieser Methode. Implementieren sie den Operator als Freundfunktion.

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.

Strecke.h

```
// alles was für die Klassendeklaration benötigt wird
#pragma once
#include <string>
using namespace std;
```

Klassendeklaration der Klasse `Strecke`

```
class Strecke
{
protected:
    const string startort;
    const string zielort;
    const double entfernung;

public:
    Strecke(const string& _startort, const string& _zielort,
            double _entfernung);
    string getStartort() const;
    string getZielort() const;
    double getEntfernung() const;
    string toString() const;
    virtual void legeZurueck() const = 0;
};
```

Strecke.cpp

```
// alles was für die Klassendefinition benötigt wird
#include "Strecke.h"
#include <sstream>
```

Klassendefinition der Klasse Strecke

```
Strecke::Strecke(const string& _startort, const string& _zielort,
                 double _entfernung): startort(_startort), zielort(_zielort),
                                      entfernung(_entfernung) {};

string Strecke::getStartort() const {
    return startort;
};

string Strecke::toString() const {
    stringstream sstream;
    sstream << "Von " << startort << " nach " << zielort
            << ": " << entfernung << " km.";
    return sstream.str();
};
```

Route.h

```
// alles was für die Klassendeklaration benötigt wird
#include "Strecke.h"
#include <vector>
```

Klassendeklaration der Klasse Route

```
class Route
{
    vector<Strecke*> teilStrecken;
    double laenge;
    int maxAnzahl;

public:
    Route(Strecke* _strecke, int _maxAnzahl);
    void fuegeStreckeHinzu(Strecke* _strecke);
    string toString() const;
    void starten() const;
    Route& operator+(Strecke* _strecke);
    friend ostream& operator<<(ostream& _ostream, const Route& _route);
};
```

Route.cpp

```
// alles was für die Klassendefinition benötigt wird
#include "Route.h"
#include <sstream>
```

Klassendefinition der Klasse Route

```
Route::Route(Strecke *_strecke, int _maxAnzahl)
{
    teilStrecken.push_back(_strecke);
    maxAnzahl = _maxAnzahl;
    laenge = _strecke->getEntfernung();
}

void Route::fuegeStreckeHinzu(Strecke* _strecke)
{
    teilStrecken.push_back(_strecke);
    laenge = laenge + _strecke->getEntfernung();
}
string Route::toString() const
{
    stringstream ss;
    ss << "Die Route von " << teilStrecken[0]->getStartort();
    ss << " nach " << teilStrecken[teilStrecken.size()-1]->getZielort(); //
    ss << " enthält " << teilStrecken.size() << " Strecken und";
    ss << " hat die Laenge " << laenge << " km.";
    return ss.str();
}

void Route::starten() const
{
    for (Strecke* _strecke : teilStrecken) {
        _strecke->legeZurueck();
    }
};

Route& Route::operator+(Strecke* _strecke)
{
    fuegeStreckeHinzu(_strecke);
    return *this;
}
ostream &operator<<(ostream& _ostream, const Route& _route)
{
    _ostream << _route.toString() << endl;
    return _ostream;
}
```

Aufgabe 3: Vererbung (8 min)

Die Klassen `Fussweg` und `Busstrecke` sind Konkretisierungen der Klasse `Strecke`.
Leiten Sie die Klasse `Fussweg` von der Klasse `Strecke` ab. Sie soll folgende Anforderungen erfüllen:

a) Ein parametrisierter Konstruktor soll die drei Parameter `_startort`, `_zielort` und `_entfernung` entgegennehmen und die Initialisierung der entsprechenden Attribute an den Konstruktor der Basisklasse delegieren. Es sollen keine Kopien der Attribute erzeugt werden.

b) Die rein virtuelle Methode `legeZurueck` der Basisklasse soll überschrieben werden. Nutzen Sie die Fähigkeit des Compilers um sicherzustellen, dass sie die Methode korrekt überschreiben. Es soll folgender Text auf der Konsole ausgegeben werden: „`Laufen Sie die Strecke '<startort>' nach '<zielort>' zu Fuss.`“

Die Klasse Busstrecke muss nicht implementiert werden.

Das Hauptprogramm am Ende der Aufgabe zeigt eine Anwendung der Klassen `Strecke`, `Route`, `Fussweg` und `Busstrecke`.

Ergänzen Sie die folgenden Programmgerippe. Verwenden Sie – falls möglich – konstante Methoden.
Trennen sie die Umsetzung sinnvoll in Headerdateien und Implementierungsdateien.

`Fussweg.h`

```
// alles was für die Klassendeklaration benötigt wird
#include "Strecke.h"
```

Klassendeklaration der Klasse `Fussweg`

```
class Fussweg : public Strecke
{
public:
    Fussweg(const string& _startort, const string& _zielort,
            double _entfernung);
    void legeZurueck() const override;
};
```

Fussweg.cpp

```
// alles was für die Klassendefinition benötigt wird
#include "Fussweg.h"
#include <iostream>
```

Klassendefinition der Klasse Fussweg

```
Fussweg::Fussweg(const string& _startort, const string& _zielort, double
_entfernung) : Strecke(_startort, _zielort, _entfernung){};


void Fussweg::legeZurueck() const
{
    cout << "Laufen Sie die Strecke '" << startort
         << " nach " << zielort << "' zu Fuss." << endl;
}
```

main.cpp

```
#include "Route.h"
#include "Busstrecke.h"
#include "Fussweg.h"
#include <iostream>
using namespace std;

int main()
{
    Fussweg* s0 = new Fussweg("Wohnung", "Haltestelle Serrach Bärenwiesen",
                              0.8);
    Busstrecke* s1 = new Busstrecke("Haltestelle Serrach Bärenwiesen",
                                    "Haltestelle Hochschulzentrum", 12);
    Fussweg* s2 = new Fussweg("Haltestelle Hochschulzentrum",
                              "Hochschule Esslingen", 0.4);
    Route wegZurHochschule(s0, 5);
    wegZurHochschule.fuegeStreckeHinzu(s1);
    wegZurHochschule = wegZurHochschule + s2;
    cout << wegZurHochschule;
    cout << endl;
    cout << "Die folgenden Teilstrecken liegen auf dieser Route:";
    cout << endl;
    cout << s0->toString() << endl;
    cout << s1->toString() << endl;
    cout << s2->toString() << endl;
    cout << endl;
    wegZurHochschule.starten();
    return 0;
}
```

Der anschließende Text gibt die Ausgabe des Programms wieder.

```
Die Route von Wohnung nach Hochschule Esslingen enthält 3 Strecken und hat die
Laenge 13.2 km.

Die folgenden Teilstrecken liegen auf dieser Route:
Von Wohnung nach Haltestelle Serrach Bärenwiesen: 0.8 km.
Von Haltestelle Serrach Bärenwiesen nach Haltestelle Hochschulzentrum: 12 km.
Von Haltestelle Hochschulzentrum nach Hochschule Esslingen: 0.4 km.

Laufen Sie die Strecke 'Wohnung nach Haltestelle Serrach Bärenwiesen' zu Fuss.
Fahren Sie die Strecke 'Haltestelle Serrach Bärenwiesen nach Haltestelle
Hochschulzentrum' mit dem Bus.
Laufen Sie die Strecke 'Haltestelle Hochschulzentrum nach Hochschule
Esslingen' zu Fuss.
```

Aufgabe 4: Exception Handling (13 min)

Die Klassen `Strecke` und `Route` aus Aufgabe 2 sollen für eine Urlaubsplanung verwendet werden. Folgende Änderungen bzw. Erweiterungen werden als gegeben angesehen und müssen nicht implementiert werden:
Die Klasse `Strecke` enthält keine rein virtuelle Methode `legeZurueck`, d.h. es können Objekte der Klasse erzeugt werden.
Die Klasse `Route` enthält zusätzlich einen Konstruktor, der lediglich einen Wert für die maximale Anzahl von Teilstrecken entgegennimmt und das entsprechende Attribut setzt.

Implementieren Sie folgende Anforderungen:

a) In der Datei `Route.cpp` (s. unten) erweitern Sie die Implementierung der Methode `fuegeStreckeHinzu`. Bevor die Strecke hinzugefügt wird, sollen 2 Prüfungen erfolgen.
   - Ist die maximale Anzahl von Teilstrecken erreicht, soll die (Standard-) Exception `out_of_range` ausgelöst werden.
   - Falls der Startpunkt der neuen Strecke nicht gleich dem Endpunkt der zuletzt gespeicherten Teilstrecke ist, soll die (Standard-) Exception `invalid_argument` ausgelöst werden.

b) Ergänzen Sie den Quellcode der in der Datei `planeUrlaub.cpp` (siehe nächste Seite) so, dass alle möglichen Ausnahmen behandelt werden. Die Ausnahmebehandlung soll wie folgt funktionieren:
   - Tritt die Ausnahme `out_of_range` auf, wird eine Meldung ausgegeben und die Variable `esReicht` auf `true` gesetzt. Dadurch wird die Schleife und damit auch die Funktion beendet.
   - Tritt die Ausnahme `invalid_argument` auf, wird eine Meldung ausgegeben und die Schleife kann fortgesetzt werden, d.h. es werden neue Argumente eingelesen.
   - Tritt irgendeine Ausnahme (alle außer `out_of_range` und `invalid_argument`) auf- dies kann z.B. durch die Ein- und Ausgabeaktionen passieren - wird eine Meldung ausgegeben und die Ausnahme soll weitergereicht werden an den Aufrufer der Funktion `planeUrlaub()`.

`Route.cpp`

```
// fuegeStreckeHinzu
{
    if (route.size() == anzahl)
        throw out_of_range("Zu viele Strecken");

    if (route.size() > 0 &&
        route[route.size() - 1]->getZielort() != s->getStartort())
        throw invalid_argument("Zielort nicht gleich Startort");




    // Implementierung aus Aufgabe 2
}
```

planeUrlaub.cpp

```
#include "Route.h"
#include <iostream>
#include <exception>
using namespace std;
void planeUrlaub()
{
    Route urlaub(3);
    string start, ziel;
    double distanz;
    bool esReicht = false;
    while (!esReicht)
    {

        try
        {

            cout << "Naechste Etappe : " << endl;
            cout << "Start: ";
            cin >> start;
            cout << endl
                 << "Ziel: ";
            cin >> ziel;
            cout << endl
                 << "Entfernung: ";
            cin >> distanz;
            urlaub.fuegeStreckeHinzu(new Strecke(start, ziel, distanz));

        }
        catch (out_of_range e)
        {
            cout << "Urlaub wird zu lang" << endl;
            esReicht = true;
        }
        catch (invalid_argument e)
        {
            cout << "Falsche Strecke" << endl;
        }
        catch (...)
        {
            cout << "Sonstiger Fehler: Abbruch" << endl;
            throw;
        }




    }
}
```

Aufgabe 5: Templates (5 min)

Die unten angegebene Funktion `sortiere` kann die in einem Array gespeicherten integer-Werte aufsteigend sortieren. Schreiben Sie die Funktion in eine Templatefunktion um, sodass sie beliebige primitive Datentypen oder Objekte sortieren kann, die den „<“-`Operator` implementieren. Führen sie die Änderungen direkt im unten angegebenen Quellcode durch. Die `main`-Funktion müssen Sie nicht anpassen.

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

const unsigned int laenge = 10;


template <typename T>
array<int, laenge> sortiere(array<int, laenge> _feld)
{     T                              T
    bool sortiert = false;

    while (!sortiert)
    {
        sortiert = true;
        for (size_t i = 0; i < _feld.size() - 1; i++)
        {
            if (_feld[i] > _feld[i + 1])
            {   T
                int temp = _feld[i];
                _feld[i] = _feld[i + 1];
                _feld[i + 1] = temp;

                sortiert = false;
            }
        }
    }

    return _feld;
}

int main(int argc, char *argv[])
{
    array<int, laenge> int_feld = {10, 2, 7, 5, 8, 3, 4, 1, 9, 6};
    array<int, laenge> int_feld_sortiert = sortiere(int_feld);

    for (size_t i = 0; i < int_feld_sortiert.size(); i++)
    {
        cout << i << ": " << int_feld_sortiert[i] << endl;
    }
}
```

Aufgabe 6: Polymorphie (14 min)

Ergänzen Sie die folgenden Klassen und das Hauptprogramm so, dass als Ausgabe erscheint:

```
B::print
A::druck
A::print
```

Es dürfen keine neuen Variablen oder Objekte erzeugt werden!

```
#include <iostream>
using namespace std;
class A
{
public:
    virtual void print() { cout << "A: : print " << endl; }
    void druck() { cout << "A: : druck " << endl; }
};
class B : public A
{
public:
    void print() { cout << "B: : print " << endl; }
    void druck() { cout << "B: : druck " << endl; }
};
int main()
{
    A *a = new B();


    a->print();
    a->druck();
    a->A::print();


    return 0;
}
```

Schauen Sie sich das nachfolgende Programm an und geben Sie darunter an, was es ausgibt.

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

class WeAll {
public:
    void who() { cout << "Alle koennen programmieren "; }
    void canDo() { cout << "in C++ "; }
    void cantDo() { cout << "aber nicht in C#." << endl; }
};
class WeBoth : public WeAll {
public:
    void who() { cout << "Wir beide koennen programmieren "; }
    virtual void canDo() { cout << "in C++ und Java "; }
    void cantDo() { cout << "aber nicht in PHP." << endl; }
};
class Myself : public WeBoth {
public:
    void who() { cout << "Ich kann programmieren "; }
    void canDo() override { cout << "in C++ und Rust "; }
    virtual void cantDo() { cout << "aber nicht in Javascript." << endl; }
};
int main() {
    Myself ms;
    WeAll* waPtr = &ms;
    WeBoth* wbPtr  = &ms;
    Myself* msPtr = &ms;

    waPtr->who(); waPtr->canDo(); waPtr->cantDo();
    wbPtr->who(); wbPtr->canDo(); wbPtr->cantDo();
    msPtr->who(); msPtr->canDo(); msPtr->cantDo();

    return 0;
}
```

Ausgabe:

```
Alle koennen programmieren in C++ aber nicht in C#.
Wir beide koennen programmieren in C++ und Rust aber nicht in PHP.
Ich kann programmieren in C++ und Rust aber nicht in Javascript.
```