| | | | |
|---|---|---|---|
| Sommersemester 2022 | | Seite | 1 von 16 |
| Fachbereich: | Informationstechnik | Studiengang: | TIB, SWB, IEP |
| Prüfungsfach: | OOS1 | Prüfungsnummer.: | 1052027 |
| Hilfsmittel: | handschriftliche Notizen (auch gedruckt)<br>2 Blätter DIN A4 beidseitig | Zeit: | 90 min |
| Nachname:<br>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 |
|---|---|---|
| Ein Attribut einer Klasse, das als `static` und `const` deklariert ist, muss über die Initialisierungsliste in einem Konstruktor initialisiert werden.<br>Begründung:<br>Static-Attribute müssen außerhalb der Klasse ¨ahnlich wie eine globale Variable definiert und initialisiert werden. Dabei ist es egal, ob sie const oder non-const sind | | x |
| Die Initialisierungen in einer Initialisierungsliste können alternativ auch als Zuweisungen im Rumpf des Konstruktors geschrieben werden.<br>Begründung:<br>Konstante Instanzvariablen dürfen nicht zugewiesen werden, sie können nur in der Initialisierungsliste initialisiert werden. Auch parametrisierte Konstruktoraufrufe von Basisklassen sind nur in der Initialisierungsliste möglich. | | x |
| Für eine tiefe Kopie benötigt man einen selbstdefinierten Kopierkonstruktor.<br>Begründung:<br>Andernfalls wird nur das erste Element, z.B. der Anker einer verketteten Liste kopiert | x | |
| Der Freund einer Klasse darf auf als `protected` deklarierte Elemente der Klasse zugreifen.<br>Begründung:<br>Der Freund einer Klasse darf auf alles zugreifen. | x | |

| Aussage | wahr | falsch |
|---|---|---|
| Der Zuweisungsoperator `operator=()` kann nur als Methode einer Klasse realisiert werden.<br>Begründung:<br>Eine Realisierung mittels globaler Funktion oder Freundfunktion ist auch möglich | | x |
| Bei mehrfacher Vererbung kann es zu mehrfachem Erbgut kommen. Dies kann durch die Verwendung virtueller Methoden verhindert werden.<br>Begründung:<br>Zur Verhinderung von mehrfachem Erbgut wird virtuelle Vererbung verwendet, nicht virtuelle Methoden | | x |
| In einer `Map` kann effizienter gesucht werden als in einem `Vector`.<br>Begründung:<br>Eine Map organisiert sich selbst. Die Suche ist in O(log N) möglich | x | |
| Klasse `B` ist von Klasse `A` abgeleitet, Klasse `C` ist von Klasse `B` abgeleitet. Ein Zeiger, der auf `C` typisiert ist (`C * ptr;`), kann ohne Probleme auf Objekte der Klasse `A` zeigen (`A a; C * ptr = &a;`).<br>Begründung:<br>Das Objekt a hat nicht alle Eigenschaften, die der Compiler beim Zugriff über ptr erwarten würde. Es kann zu unerwartetem Verhalten bis hin zum Programmabsturz kommen | | x |
| Eine Klasse, die eine oder mehrere virtuelle Methoden enthält, wird damit zu einer abstrakten Basisklasse.<br>Begründung:<br>Es müssen rein virtuelle (oder anders gesagt abstrakte) Methoden sein | | x |
| Ein Handler muss für Exceptions von Basisklassen im Programmcode hinter dem Handler für Exceptions von abgeleiteten Klassen stehen.<br>Begründung:<br>Andernfalls würde nach dem Substitutionsprinzip auch die Exception der abgeleiteten Klasse gefangen. | x | |

# Aufgabe 2: Klassen, Attribute, Methoden, Operatoren (26 min)

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

a) Ein Spieler hat das Attribut `punkte`, das einen ganzzahligen Wert speichert (die im Spiel gesammelten Punkte), sowie das Attribut `name` vom Typ String. Die Attribute sind nicht öffentlich zugänglich. Das Attribut `name` wird durch einen parametrisierten Konstruktor gesetzt und kann danach nicht mehr geändert werden, die Punktzahl wird im Konstruktor mit `0` initialisiert.
b) Die Klasse Spieler hat `get`-Methoden für beide Attribute.
c) Die Methode `subtractPunkte` nimmt einen ganzzahligen Wert entgegen und subtrahiert ihn von der aktuellen Punktzahl des Spielers.
d) Die Methode `resetPunkte` setzt die aktuelle Punktzahl auf `0` zurück.
e) Zur Addition von Punkten definiert die Klasse einen Operator `+=` als Methode. Dieser addiert eine Zahl (2. Operand) zu der aktuellen Punktzahl des Spielers hinzu und gibt eine Referenz auf den Spieler zurück.
f) Die Klasse hat eine Methode `toString()`, die zu einem Spieler einen String zurückgibt:
„`Spieler <name> hat <punkte> Punkte.`“ An den mit `<>` markierten Stellen sind die Werte der Attribute des Objektes einzusetzen. Benutzen Sie in der Implementierung dieser Methode einen Stringstream. In der Beschreibung von Aufgabe 4 finden Sie ein Beispiel für die Ausgabe dieser Methode.

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

a) An einem Skatspiel nehmen drei Spieler teil. Verwalten sie diese im Attribut `spieler`. Verwenden sie hierzu einen geeigneten sequentiellen Container, der Zugriff auf die Inhalte über den index-Operator `[]` erlaubt.
b) Das Attribut `spieler` soll über einen Konstruktor initialisiert werden können, dem drei Parameter vom Typ `Spieler` übergeben werden. Entwerfen Sie die Klasse `Skatspiel` so, dass keine Kopien der Spieler erzeugt werden.
c) Der Destruktor setzt den Punktestand aller Spieler auf `0` zurück. Nutzen Sie bei der Implementierung eine geeignete Schleife und rufen Sie die Methode `resetPunkte()` der Spielerobjekte auf.
d) Die virtuelle Methode `listeSpielstand()` gibt die aktuellen Informationen aller drei am Spiel beteiligten Spieler auf dem Bildschirm aus. Nutzen Sie bei der Implementierung eine geeignete Schleife und rufen Sie die Methode `tostring()` der Spielerobjekte auf. In der Beschreibung von Aufgabe 4 finden Sie ein Beispiel für die Ausgabe dieser Methode.
e) Die rein virtuelle Methode `addErgebnis` nimmt als Parameter eine Spielernummer (Speicherplatz im Container), die gespielten Punkte (ganzzahliger Wert) und die Information ob das Spiel gewonnen wurde in Form eines booleschen Wertes entgegen.

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.

Spieler.h

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

Klassendeklaration der Klasse `Spieler`

```
class Spieler
{
    const string name;
    int punkte;

public:
    Spieler(string name);
    Spieler& operator=(Spieler&) = default;
    string getName() const;
    int getPunkte() const;
    void subtractPunkte(int neuePunkte);
    void resetPunkte();
    string toString() const;
    Spieler& operator+=(int neuePunkte);
};
```

Skatspiel.h

```
// alles was für die Klassendeklaration benötigt wird
#pragma once;
#include "spieler.h"
#include <array>
```

Klassendeklaration der Klasse `Skatspiel`

```
class Skatspiel
{
protected:
    array<Spieler *, 3> spieler;
public:
    Skatspiel(Spieler &sp1, Spieler &sp2, Spieler &sp3);
    virtual void addErgebnis(int spielerNr, int punkte, bool gewonnen) = 0;
    virtual void listeSpielstand() const;
    ~Skatspiel();
};
```

Spieler.cpp

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

Klassendefinition der Klasse `Spieler`

```
Spieler::Spieler(string name) : name(name)
{
    punkte = 0;
}
string Spieler::getName() const { return name; }
int Spieler::getPunkte() const { return punkte; }
void Spieler::subtractPunkte(int neuePunkte)
{
    punkte -= neuePunkte;
}
void Spieler::resetPunkte()
{
    punkte = 0;
};
string Spieler::toString() const
{
    stringstream result;
    result << "Spieler " << name << " hat "
           << punkte << " Punkte.";
    return result.str();
}
Spieler &Spieler::operator+=(int neuePunkte)
{
    punkte += neuePunkte;
    return *this;
}
```

Skatspiel.cpp

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

Klassendefinition der Klasse `Skatspiel`

```
Skatspiel::Skatspiel(Spieler &sp1, Spieler &sp2, Spieler &sp3)
{
    spieler[0] = &sp1;
    spieler[1] = &sp2;
    spieler[2] = &sp3;
}

void Skatspiel::listeSpielstand() const
{
    cout << "Aktueller Spielstand:" << endl;
    for (Spieler *s : spieler)
    {
        cout << s->toString() << endl;
    }
}

Skatspiel::~Skatspiel()
{
    for (Spieler *s : spieler)
    {
        s->resetPunkte();
    }
}
```

# Aufgabe 3: Vererbung (16 min)

Leiten Sie die Klassen `Klassisch` und `Bierlachs` von der Klasse `Skatspiel` ab. Beide repräsentieren unterschiedliche Zählweisen beim Skat.
Die Klasse `Klassisch` soll folgende Anforderungen erfüllen:

a) Ein parametrisierter Konstruktor soll drei Parameter von Typ `Spieler` entgegennehmen und die Initialisierung des Attributs `spieler` an den Konstruktor der Basisklasse delegieren. Auch hier sollen keine Kopien der Spieler erzeugt werden.
b) Die rein virtuelle Methode `addErgebnis` der Basisklasse soll überschrieben werden. Wird das Spiel gewonnen, werden dem Spieler, der gespielt hat (die übermittelte Spielernummer) die gespielten Punkte gutgeschrieben, d.h. die Punkte werden zu seinem Punktestand addiert. Wird das Spiel verloren, so wird dem Spieler, der gespielt hat, die doppelte Anzahl der gespielten Punkte abgezogen.

Die Klasse `Bierlachs` soll folgende Anforderungen erfüllen:

a) Ein parametrisierter Konstruktor soll drei Parameter von Typ `Spieler` entgegennehmen und die Initialisierung des Attributs `spieler` an den Konstruktor der Basisklasse delegieren. Auch hier sollen keine Kopien der Spieler erzeugt werden.
b) Die rein virtuelle Methode `addErgebnis` der Basisklasse soll überschrieben werden. Wird das Spiel gewonnen, so wird den beiden Spielern, die nicht gespielt haben, die gespielten Punkte abgezogen. Wird das Spiel verloren, so wird dem Spieler, der gespielt hat, die doppelte Anzahl der gespielten Punkte abgezogen.
c) Die virtuelle Methode `listeSpielstand` soll überschrieben werden. Zunächst soll der aktuelle Spielstand durch einen Aufruf der Methode `listeSpielstand` der Basisklasse ausgegeben werden. Anschließend soll geprüft werden, ob mindestens einer der Spieler bereits mehr als 301 Minuspunkte gesammelt hat. Ist dies der Fall, soll die folgende Meldung ausgegeben werden: „Das Spiel ist beendet.“

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

Klassisch.h

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

Klassendeklaration der Klasse `Klassisch`

```
class Klassisch: public Skatspiel {
public:
    Klassisch(Spieler& sp1, Spieler& sp2, Spieler& sp3);

    void addErgebnis(int spielerNr, int punkte, bool gewonnen) override;

};
```

Bierlachs.h

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

Klassendeklaration der Klasse `Bierlachs`

```
class Bierlachs: public Skatspiel {
public:
    Bierlachs(Spieler& sp1, Spieler& sp2, Spieler& sp3);

    void addErgebnis(int spielerNr, int punkte, bool gewonnen) override;

    void listeSpielstand() const override;

};
```

Klassisch.cpp

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

Klassendefinition der Klasse `Klassisch`

```
Klassisch::Klassisch(Spieler& sp1, Spieler& sp2, Spieler& sp3): Skatspiel(sp1,
sp2, sp3) {}

void Klassisch::addErgebnis(int spielerNr, int punkte, bool gewonnen)
{
    if (gewonnen)
    {
        *spieler[spielerNr] += punkte;
    }
    else
    {
        spieler[spielerNr]->subtractPunkte(punkte * 2);
    }
}
```

Bierlachs.cpp

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

Klassendefinition der Klasse `Bierlachs`

```
Bierlachs::Bierlachs(Spieler& sp1, Spieler& sp2, Spieler& sp3): Skatspiel(sp1,
sp2, sp3) {}

void Bierlachs::addErgebnis(int spielerNr, int punkte, bool gewonnen)
{
    if (gewonnen)
    {
        for (Spieler *s : spieler)
        {
            if (s->getName() != spieler[spielerNr]->getName())
                s->subtractPunkte(punkte);
        }
    }
    else
    {
        spieler[spielerNr]->subtractPunkte(punkte * 2);
    }
}

void Bierlachs::listeSpielstand () const
{
    Skatspiel::listeSpielstand();
    for (Spieler *s : spieler)
    {
        if (s->getPunkte() < -301)
        {
            cout << "Das Spiel ist beendet." << endl;
            break;
        };
    }
}
```

# Aufgabe 4: Polymorphie (4 min)

Das unten angegebene Programm erzeugt drei Spieler, und jeweils eine Spielrunde klassischer Zählweise und eine Spielrunde Bierlachs.
Für beide Spielrunden werden jeweils drei Ergebnisse erzeugt, der Spielstand ausgegeben und das Objekt, welches die Spielrunde repräsentiert wieder zerstört.

```
#include "spieler.h"
#include "bierlachs.h"
#include "klassisch.h"

int main() {
    Spieler a("Tick");
    Spieler b("Trick");
    Spieler c("Track");

    Bierlachs bl(a, b, c);
    bl.addErgebnis(0, 54, true);
    bl.addErgebnis(1, 18, false);
    bl.addErgebnis(2, 144, false);
    bl.listeSpielstand();
    bl.~Bierlachs();

    Klassisch kl(a,b,c);
    kl.addErgebnis(0, 54, true);
    kl.addErgebnis(1, 18, false);
    kl.addErgebnis(2, 144, false);
    kl.listeSpielstand();
    kl.~Klassisch();

    return 0;
}
```

Es wird die folgende Ausgabe erzeugt:

```
Aktueller Spielstand:
Spieler Tick hat 0 Punkte.
Spieler Trick hat -90 Punkte.
Spieler Track hat -342 Punkte.
Das Spiel ist beendet.
Aktueller Spielstand:
Spieler Tick hat 54 Punkte.
Spieler Trick hat -36 Punkte.
Spieler Track hat -288 Punkte.
```

Lagern Sie die Erzeugung der Ergebnisse, das Ausgeben des Spielstandes und die Zerstörung des Objekts in die Funktion `testeVerwaltung` aus. Nutzen Sie dafür Polymorphie.

```
#include "spieler.h"
#include "bierlachs.h"
#include "klassisch.h"
// Hier testeVerwaltung implementieren


void testeVerwaltung(Skatspiel& skat) {
    skat.addErgebnis(0, 54, true);
    skat.addErgebnis(1, 18, false);
    skat.addErgebnis(2, 144, false);

    skat.listeSpielstand();
    skat.~Skatspiel();
}


int main() {
    Spieler a("Tick");
    Spieler b("Trick");
    Spieler c("Track");

    Bierlachs bl(a, b, c);
    // Hier testeVerwaltung für bl aufrufen
    testeVerwaltung(bl);

    Klassisch kl(a,b,c);
    // Hier testeVerwaltung für kl aufrufen
    testeVerwaltung(kl);

    return 0;
}
```

# Aufgabe 5: Templates (11 min)

Schreiben Sie ein Klassentemplate mit Namen `Knoten` für eine baumartige Datenstruktur, die in jedem Knoten ein Datenelement speichern kann. Jeder Knoten des Baumes soll beliebig (endlich) viele Kinder haben können. Nutzen Sie das Klassentemplate `vector`, um alle Kinder eines Knotens zu speichern. Stellen Sie sicher, dass das folgende Hauptprogramm ausgeführt werden kann. Sie können dabei davon ausgehen, dass der Ausgabeoperator `<<` für die Daten, die in den Knoten gespeichert werden, definiert ist.

```
int main()
{
    Knoten<int> k1(1);
    cout << "Datum in k1 : " << k1.getData() << endl;
    Knoten<int> k2(2);
    Knoten<int> k3(3);
    Knoten<int> k4(4);
    k3.plusKind(k4);
    k1.plusKind(k2);
    k1.plusKind(k3);
    cout << "Anzahl der Kinder von k1 : " << k1.anzKinder() << endl;
    k1.print();
    Knoten<string> c1("Hallo");
    Knoten<string> c2("Du");
    Knoten<string> c3("da");
    c1.plusKind(c2);
    c1.plusKind(c3);
    c1.print();
    return 0;
}
```

Die Ausgabe der Hauptfunktion ist:

```
Datum in k1 : 1
Anzahl der Kinder von k1 : 2
1 - 2 - 3 - 4 -
Hallo - Du - da -
```

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


// Templatedefinition

template <typename T>

class Knoten
{
// Instanzvariablen

    T data;
    vector<Knoten> kinder;


// Konstruktor

public:
    Knoten(T data) : data(data) {}
    T getData()
    {
        return data;
    }

// Methoden

    int anzKinder()
    {
        return kinder.size();
    }
    void plusKind(Knoten &k)
    {
        kinder.push_back(k);
    }
    void print(bool nl = true)
    {
        cout << data << " - ";
        for (unsigned int i = 0; i < kinder.size(); i++)
        {
            kinder[i].print(false);
        }
        if (nl)
            cout << endl;
    }


}
```

# Aufgabe 6: Konstruktoren, Destruktoren, Exceptions (13 min)

Analysieren Sie das nachfolgende Programm und schreiben Sie die Ausgabe des Programms unterhalb von „Ausgabe:“ (nächste Seite).

```
#include <iostream>
using namespace std;
class Base
{
protected:
    int id;
public:
    Base(int _id): id(_id) { cout << "create base " << id << endl; }
    ~Base() { cout << "destroy base " << id << endl; }
};

class Derived: public Base {
public:
    Derived(int _id): Base(_id) { cout << "create derived " << id << endl; }
    ~Derived() { cout << "destroy derived " << id << endl; }
};

void func(int n)
{
    cout << "enter func, n=" << n << endl;
    Derived derived(n);
    if (n == 0)
    {
        throw 100;
    }
    func(n - 1);
    cout << "leave func, n=" << n << endl;
}
int main()
{
    try
    {
        Base base(5);
        func(1);
    }
    catch (int e)
    {
        cout << "caught " << e << endl;
    }
}
```

Ausgabe:

```
create base 5
enter func, n=1
create base 1
create derived 1
enter func, n=0
create base 0
create derived 0
destroy derived 0
destroy base 0
destroy derived 1
destroy base 1
caught 100
destroy base 5
```