

## 5. Semester, 1. Prüfung

Name	
------	--

- Die gesamte Prüfung bezieht sich auf die Programmierung in C++!
- Prüfungsdauer: 90 Minuten
- Mit Kugelschreiber oder Tinte schreiben
- Lösungen können direkt auf die Aufgabenblätter geschrieben werden
- PC's sind nicht erlaubt
- Unterlagen und Bücher sind erlaubt
- Achte auf Details wie Punkte, Kommas und Semikolons

Aufgabe	Punkte	
Analyse und Design	12	
Exceptions	14	
Programmverständnis	14	
Weitere Fragen	10	
<i>Total</i>		

## 1. Analyse und Design

- a) Zeichne für folgendes Problemstellung die möglichen Klassen auf (als einfaches Klassendiagramm) und verdeutliche ihre Abhängigkeiten mit Linien untereinander. Du musst keine Methoden, Funktionen oder Datenelemente definieren.

Du arbeitest bei einer Firma, die Mischpulte auf PC-Basis herstellt.

Ein Mischpult hat viele Kanäle. Ein Kanal beinhaltet mehrere

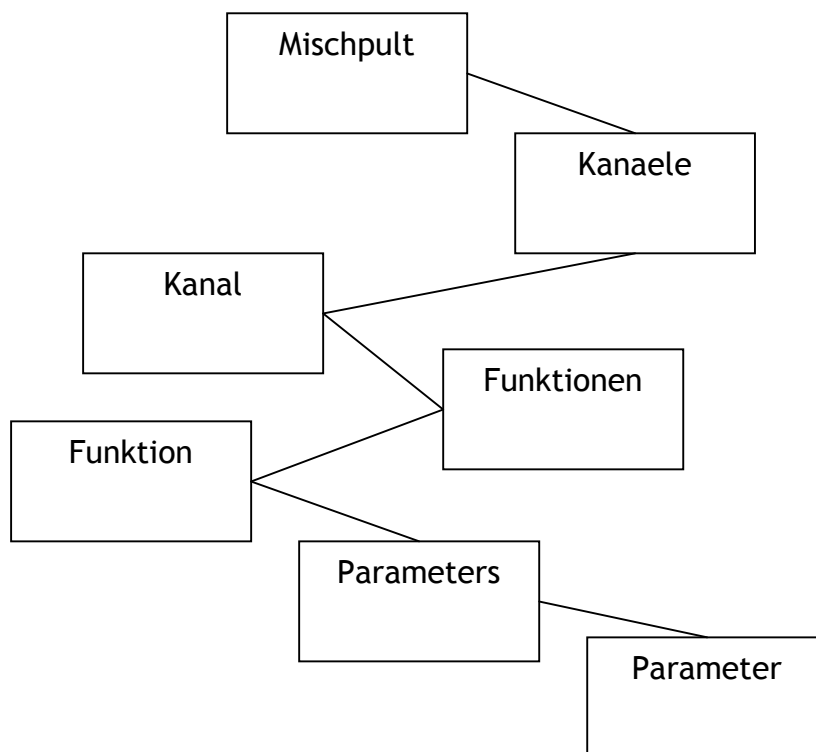
Funktionen. Eine Funktion beinhaltet schlussendlich mehrere

Parameter. Finde aus dieser kurzen Beschreibung die richtigen

Klassen. Dazu gehören auch Klassen, die eine Sammlung von anderen Objekten beinhalten. (7 Punkte)

Das Datenmodell wird beim Start der Software einmal aufgebaut und danach nicht mehr geändert. Was für eine Container-Klasse würdest du für die Sammlungen wählen? (1 Punkt)

(Total 8 Punkte)



vector

- b) Gegeben sei eine Klasse *TransactionError* (Transaktions-Fehler). Diese Klasse wird von einer Bank verwendet um bei Geldtransaktionen die aufgetretenen Fehler zu protokollieren. Diese Fehler werden alle gesammelt und in einem Container einfach angefügt. Diese Einträge werden erst nach einem Jahr alle zusammen gelöscht.

Was für eine Container-Klasse, die du kennst würdest du hier einsetzen? (1 Punkt)

Definiere einen neuen Datentypen mit *typedef*, der solche Elemente *TransactionError* in einem Container beinhaltet. Vergiss nicht die nötigen *#include*-Anweisungen und die *namespace*-Spezifizierer (2 Punkte)

Erzeuge eine Variable von diesem Datentypen. (1 Punkt)

(Total 4 Punkte)

```
vector
```

```
#include <vector>
```

```
typedef std::vector<TransactionError> Errors;
```

```
Errors allErrors;
```

## 2. Exceptions

- a) Schau dir die folgenden Klassen *Graph* und *GraphException* an. Die Klasse *Graph* dient zur grafischen Ausgabe von einigen Elementen. Falls etwas nicht funktioniert, wird ein Objekt der Klasse *GraphException* erzeugt und geworfen. Schreibe ein kleines Programm (eine *main*-Funktion), das eine Linie, ein Quadrat und einen Kreis zeichnet und verwende hierfür ein Objekt der Klasse *Graph*. Verwende Exception-Handling und mache etwas sinnvolles mit den „gefangenen“ Fehlern. Beachte hierfür aber die Klasse *GraphException* genau. Die Lösung hat etwa 10 Zeilen Code. (8 Punkte)

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

class GraphException
{
public:
    GraphException(const string& text)
        : m_text(text)
    {
    }

    string GetErrorText() const
    {
        return m_text;
    }
private:
    string m_text;
};

class Graph
{
public:
    Graph()
    {
        if(!OpenGraphicalWindow())
        {
            throw GraphException("Could not open");
        }
    }

    ~Graph()
    {
        CloseGraphicalWindow();
    }

    void Line(int x1, int y1, int x2, int y2)
    {
        if(!DrawLine(x1, y1, x2, y2))
        {
            throw GraphException("Could not draw line");
        }
    }
    // siehe nächste Seite mit mehr Code zur Klasse Graph
```

```
void Quad(int x, int y, int a)
{
    if(!DrawQuad(x, y, a))
    {
        throw GraphException("Could not draw line");
    }
}

void Circle(int x, int y, int r)
{
    if(!DrawCircle(x, y, r))
    {
        throw GraphException("Could not draw circle");
    }
}

};

int main()
{
    // Dei Klasse Graph wirft Exceptions also try-catch
    try
    {
        Graph einGraph;
        einGraph.Line(0,0,10,10);
        einGraph.Quad(10,10,10);
        einGraph.Circle(50,50,50);
    }
    catch(GraphException& e)
    {
        cout << „Fehler : „ << e.GetErrorText() << endl;
    }
    return 0;
}
```

- b) Schreibe eine Funktion *Division*, die als Argument zwei *doubles* hat. Das erste ist der Dividend, das zweite der Divisor. Das Ergebnis soll der Rückgabewert (auch *double*) sein (Ergebnis = Dividend/Divisor). Die Funktion soll prüfen, ob der Divisor gleich 0.0 ist und in diesem Fall eine Ausnahme (Exception) werfen. Nimm als Exception-Klasse, die geworfen wird eine Klasse deiner Wahl. (4 Punkte)  
Schreibe eine kleine main-Funktion, die diese Funktion in einem *try-catch*-Block aufruft und allenfalls die Ausnahme fängt. Vergiss nicht die nötigen *#include*- und namespace -Anweisungen. (2 Punkte) (Total 6 Punkte)

```
#include <string>
#include <iostream>

using namespace std;

// Funktion mit double als Rückgabewert, und zwei double
// Argumente, name : Division
double Division(double dividend, double divisor)
{
    // prüfe ob dividend gleich 0
    if(0.0 == dividend)
    {
        // werfe Exception z.B. string
        throw string(„Division durch null“);
    }
}

int main()
{
    try
    {
        double test = Division(3.6, 0.0);
    }
    catch(string& error)
    {
        cout << error << endl;
    }
    return 0;
}
```

### 3. Programmverständnis

- a) Gegeben sei eine Klasse *Objekt*. Definiere einen Vektor (vector aus der C++Standard-Library) von *Objekt* mit dem Namen *Objekte*. Vergiss nicht die #includes und die namespace-Spezifizierer. (2 Punkte)

```
#include <vector>

std::vector<Objekt> Objekte; // das ist eine Variable

// oder nur einen Typen definieren wäre auch korrekt
// dai die Aufgabe nicht klar gestellt war
typedef std::vector<Objekt> Objekte;
// und eine Variable
Objekte meineObjekte;
```

- b) Was gibt folgendes Programm aus? (2 Punkte)

```
#include <vector>
#include <iostream>

using namespace std;

typedef vector<int> Zahlen;

int main()
{
    Zahlen einigeZahlen;
    for(int i = 0; i < 10; ++i)
    {
        einigeZahlen.push_back(i);
    }
    Zahlen::iterator it = einigeZahlen.begin();
    Zahlen::iterator end = einigeZahlen.end();

    // neue Variable summe!
    int summe = 0;

    for( ; it != end; ++it)
    {
        // jedes Element einzeln ausgeben, mit eigener Zeile
        cout << (*it) << endl;
        // jedes Element wird zur summe addiert
        summe += (*it);
    }

    double d = (double)summe / (double)einigeZahlen.size();
    cout << d << endl;

    return 0;
}
alle Zahlen von 0 bis 9 kommen in den vector und werden
ausgegeben
0
1
2.... etc.
9
4.5 // Ergebnis von summe durch anzahl Elemente als double
```

## c) Was gibt folgender Code aus? (2 Punkte)

```
#include <string>
#include <iostream>
#include <list>

using std::string;
using std::list;

class Message
{
public:
    Message(const string& text) : _text(text)
    {
    }
    void send() const
    { // Hier ist die Ausgabe
      std::cout << "sending : " << _text << std::endl;
    }
private:
    string _text;
};

typedef list<Message> MessageList;
// Die Klasse Messages ist eine MessageList hat
// aber zusätzlich eine Methode sendAll()
class Messages : public MessageList
{
public:
    void sendAll()
    {
        iterator it = begin();
        iterator endIt = end();

        for( ; it != endIt; ++it)
        {
            it->send();
        }
    }
};

int main()
{
    // Message Liste erzeugen und einzelne Message-Objekte
    Messages someMessages;
    Message m1("Nachricht");
    Message m2("Hallo");
    Message m3("Botschaft");
    // Message Objekte einfügen
    someMessages.push_back(m1);
    someMessages.push_back(m2);
    someMessages.push_back(m3);
    // spezielle Methode aufrufen
    someMessages.sendAll();

    return 0;
}

sending : Nachricht
sending : Hallo
sending : Botschaft
```



- d) Gegeben sei folgende Klasse *CD*. Definiere eine Liste (list aus der C++Standard-Library) *CDListe* für Elemente der Klasse *CD*. (2 Punkte)  
Schreibe eine Funktion *PlayMusic*, die alle CD-Elemente in einem *CDListe*-Objekt, das als Argument übergeben wird abspielt.  
(3 Punkte) Achte auf **effiziente Parameterübergabe** und ergänze die nötigen **#include-** und **namespace-Angaben**. (Total 5 Punkte)

```
// Hier include ergänzen
#include <list>

class CD
{
public:
    void Play() const;
};

typedef std::list<CD> CDListe;

// Funktion PlayMusic
void PlayMusic(CDListe& liste)
{
    CDListe::iterator it = liste.begin();
    CDListe::iterator end = liste.end();
    for( ; it != end; ++it)
    {
        it->Play();
    }
}
```

- e) Schreibe folgende *for*-Schleife als *while*-Schleife. Drei Stichworte :  
Initialisierung, Schleifenbedingung, Reinitialisierung (3 Punkte)

```
typedef vector<int> Werte;
Werte meineWerte;

Werte::iterator it = meineWerte.begin();
Werte::iterator end = meineWerte.end();

for( ; it != end; ++it)
{
    cout << *it << endl;
}

// Initialisierung
Werte::iterator it = meineWerte.begin();
Werte::iterator end = meineWerte.end();

// Schleifenbedingung
while(it != end)
{
    // Code
    cout << *it << endl;
    // Reinitialisierung
    ++it;
}
```

## 4. Weitere Fragen

- a) Schreibe eine Funktion *SwapChar*, die zwei Variablen vom Typ *char* vertauscht (Verwende bitte keine Pointer). (3 Punkte)

```
void SwapChar(char& a, char& b)
{
    char temp = a;
    a = b;
    b = temp;
}
```

- b) Schreibe nun eine template-Funktion *Swap*, die man für beliebige Datentypen verwenden kann. (4 Punkte)

```
template<class T>
void Swap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

- c) Was machst Du wenn Du neuen Code schreibst?  
(falsche Vorschläge durchstreichen) (1 Punkt)

- ~~verkaufen und reich werden~~ ist zwar auch gut
- ~~assert aufrufen~~ Nicht direkt
- schrittweise im Debugger abarbeiten

- d) Was würdest Du in dieser Funktion in die assert-Anweisung schreiben?  
(1 Punkt)

```
int Div(int a, int b)
{
    // nur mit zweitem Argument ungleich 0 aufrufen
    // das zweite Argument ist b, also
    _ASSERT( b != 0 )
    return a / b;
}
```

- e) Willst Du fehlerfreie Programme schreiben? (1 Punkt)

Ja