Objektorientierte Systeme 1 - SWB2 & TIB2
Hausaufgabe 6

Aufgabe 1: Casts

Betrachten Sie das nachfolgende Programm. Beantworten Sie die Fragen, die im Programm gestellt werden und schreiben Sie auf, welche Ausgaben Sie erwarten. Überprüfen Sie dann Ihre Vorhersage mit einem Programmdurchlauf.

Sollten Sie beim Kompilieren Fehler bekommen, so erklären Sie diese und kommentieren Sie die fehlerhaften Zeilen aus.

```
1  #include <iostream>
2  #include <typeinfo>
3  using namespace std;
4
5  class A
6  {   char a;
7  public:
8      A(char a='A') : a(a) {}
9  };
10
11 class B1 : public A
12 {
13 public:
14     char b1;
15     B1(char a='A', char b1='1') : A(a), b1(b1) {}
16     virtual void print()
17     {   cout << "int(b1) = " << int(b1) << endl;    }
18 };
19
20 class B2 : public A
21 {   char b2;
22 public:
23     B2(char a='A', char b2='2') : A(a), b2(b2) {}
24 };
25
26 class C : public B1, public B2
27 {   char c;
28 public:
29     C(char a='A', char b1='1', char b2='2', char c='C')
30         : B1(a,b1), B2(a,b2), c(c) {}
31 };
32
33 int main()
34 {
35     B1 * b1_ptr = new B1;
36     A * a_ptr = b1_ptr;
```

```
37      cout << "1. Sind die Pointer b1_ptr und a_ptr identisch?"
38          << endl;
39      cout << boolalpha << (b1_ptr == a_ptr) << endl;
40
41      cout << "2. Wie gross sind *b1_ptr und *a_ptr? " << endl;
42      cout << "Groesse von *b1_ptr: " << sizeof(*b1_ptr) << endl;
43      cout << "Groesse von *a_ptr:  " << sizeof(*a_ptr) << endl;
44      delete b1_ptr;
45
46      C * c_ptr = new C;
47      b1_ptr = c_ptr;
48      B2 * b2_ptr = c_ptr;
49      cout << "3. Sind die Pointer b1_ptr und b2_ptr identisch? "
50          << endl;
51      cout << boolalpha << ((void*)b1_ptr == (void*)b2_ptr)
52          << endl;
53
54      cout << "4. Wie gross sind *b1_ptr und *b2_ptr? " << endl;
55      cout << "Groesse von *b1_ptr: " << sizeof(*b1_ptr) << endl;
56      cout << "Groesse von *b2_ptr: " << sizeof(*b2_ptr) << endl;
57
58      a_ptr = (B2 *)c_ptr;
59      cout << "5. Warum kann a_ptr = c_ptr; nicht kompiliert ";
60      cout << "werden?" << endl;
61      cout << "a_ptr = " << a_ptr << endl;
62      cout << "b2_ptr = " << b2_ptr << endl;
63      delete c_ptr;
64
65      a_ptr = new A;
66      b1_ptr = (B1*)a_ptr;
67      cout << "6. Was ist der Wert der Instanzvariablen b1 von ";
68      cout << "*b1_ptr?" << endl;
69      cout << hex << int(b1_ptr->b1) << endl;
70      b1_ptr->print();
71
72      cout << "7. Welches Ergebnis liefert b1_ptr = ";
73      cout << "dynamic_cast<B1*>(a_ptr); ?" << endl;
74      b1_ptr = dynamic_cast<B1*>(a_ptr);
75      cout << "b1_ptr = " << b1_ptr << endl;
76
77      b1_ptr = new B1;
78      cout << "8. Welches Ergebnis liefert c_ptr = ";
79      cout << "dynamic_cast<C*>(b1_ptr); ?" << endl;
80      c_ptr = dynamic_cast<C*>(b1_ptr);
81      cout << "c_ptr = " << c_ptr << endl;
82      delete b1_ptr;
83
84      c_ptr = new C;
```

```
85      b1_ptr = c_ptr;
86      b2_ptr = dynamic_cast<B2*>(b1_ptr);
87      cout << "9. Zeigen die Pointer c_ptr, b1_ptr und b2_ptr auf";
88      cout << " dasselbe Objekt?" << endl;
89      cout << "c_ptr =  " << c_ptr << endl;
90      cout << "b1_ptr = " << b1_ptr << endl;
91      cout << "b2_ptr = " << b2_ptr << endl;
92
93      void * v_ptr = dynamic_cast <void *>(b1_ptr);
94      cout << "10. Worauf zeigt der Pointer v_ptr?" << endl;
95      cout << "v_ptr = " << v_ptr << endl;
96
97      cout << "11. Von welchem Run-Time-Datentyp sind a_ptr, ";
98      cout << "c_ptr, *b1_ptr und v_ptr?" << endl;
99      cout << "Datentyp von a_ptr:   " << typeid(a_ptr).name()
100         << endl;
101     cout << "Datentyp von c_ptr:   " << typeid(c_ptr).name()
102         << endl;
103     cout << "Datentyp von *b1_ptr: " << typeid(*b1_ptr).name()
104         << endl;
105     cout << "Datentyp von v_ptr:   " << typeid(v_ptr).name()
106         << endl;
107     return 0;
108 }
```

Aufgabe 2: Eine eigene Exceptionklasse für Zoos

2.1 Leiten Sie von der Standardexceptionklasse `exception` eine eigene Klasse `MyException` ab. Fügen Sie Ihrer Klasse ein Attribut hinzu, in dem die Zeilennummer gespeichert wird, in der eine Ausnahme auftritt, sowie ein Attribut, in dem der Name der Datei gespeichert wird, in der eine Exception auftritt. (Dazu sollen später die Präprozessormakros `__LINE__` und `__FILE__` im Aufruf des Konstruktors genutzt werden.) Schreiben Sie einen parametrisierten Konstruktor, der mit zwei Parametern die beiden Attribute initialisiert, sowie einen dritten Parameter vom Typ C-String hat. Dieser dritte Parameter enthält den Namen bzw. eine Beschreibung der Ausnahme und wird als Parameter an den Konstruktor der Klasse `exception` übergeben.[1] Programmieren Sie außerdem noch eine Methode `print()`, welche die drei Attribute eines Objektes (Zeile, Datei und Name der Ausnahme) der Klasse ausgibt.

2.2 Ergänzen Sie die untenstehenden Quelldateien an den gekennzeichneten Stellen (//HIER). Binden Sie die noch notwendigen aber fehlenden Dateien aus vorherigen Laboren und Hausaufgaben ein.

2.3 Testen Sie Ihre Programme mit dem gegebenen Hauptprogramm aus. Führen Sie das Programm für jeden der Exception Handler so aus, dass der gewählte Handler aktiviert wird.

[1] Dies geht in Visual Studio, aber nach dem C++-Standard gibt es einen solchen Konstruktor `exception(char *)` nicht. Bei anderen Kompilern müssen Sie also evtl. ein eigenes Attribut dafür in Ihrer Klasse `MyException` anlegen.

```
#pragma once

#include <string>
#include "MyList.hpp"
#include "MyException.hpp"
using namespace std;

// Klasse der Tiere
class Tier : public MyData {
    // Name des Tiers
    string name;
public:
    // Konstruktor
    Tier(string = "");
    // virtuelle print-Funktion
    virtual void print(bool = true) const;
};

class Elefant : public Tier {
public:
    // Konstruktor
    Elefant(string = "");
    // virtuelle clone-Funktion
    virtual Elefant * clone() const;
};

class Tiger : public Tier {
public:
    // Konstruktor
    Tiger(string = "");
    // virtuelle clone-Funktion
    virtual Tiger * clone() const;
};

class Maus : public Tier {
public:
    // Konstruktor
    Maus(string = "");
    // virtuelle clone-Funktion
    virtual Maus * clone() const;
};

class Zoo {
    // Name des Zoos
    string name;
    // Die Tiere werden in einer MyList gespeichert
    MyList tiere;
```

```
public:
    // Konstruktor
    Zoo(string);
    // Eigene Exceptionklasse "ElefantTrifftAufMaus"
    // abgeleitet von MyException
    class ElefantTrifftAufMaus /*HIER*/
        // HIER

    };
    // Ein Tier dem Zoo hinzufügen
    void hinzu(const Tier &);
    // Alle Zootiere ausgeben
    void print() const;
};
```

```
#include <iostream>
#include "Zoo.hpp"
using namespace std;

////////////////////////////////////////////////////
// Klasse Tier

// Konstruktor
Tier::Tier(string n) : name(n) {}

// Methode print
void Tier::print(bool nl) const {
    cout << string(typeid(*this).name()).substr(6) << " " << name;
    if (nl) { cout << endl; }
}

////////////////////////////////////////////////////
// Klasse Elefant

// Konstruktor
Elefant::Elefant(string n) : Tier(n) {}

// virtuelle clone-Funktion
Elefant * Elefant::clone() const {
    return new Elefant(*this);
}

////////////////////////////////////////////////////
// Klasse Tiger

// Konstruktor
Tiger::Tiger(string n) : Tier(n) {}
```

```
// virtuelle clone-Funktion
Tiger * Tiger::clone() const {
    return new Tiger(*this);
}

//////////////////////////////////////////////////////
// Klasse Maus

// Konstruktor
Maus::Maus(string n) : Tier(n) {}

// virtuelle clone-Funktion
Maus * Maus::clone() const {
    return new Maus(*this);
}

//////////////////////////////////////////////////////
// Klasse Zoo

// Konstruktor für Ausnahmeklasse ElefantTrifftAufMaus
// HIER


// Konstruktor
Zoo::Zoo(string n) {
    // Name zuweisen
    name = n;
    // Wenn der String kürzer als 4 Zeichen ist,
    // dann MyException werfen
    // Nutzen Sie die Präprozessormakros __LINE__ und __FILE__
    // HIER

    // Ansonsten, den 5. Buchstaben des Namens groß machen
    name.at(4) = toupper(name.at(4));
}

// Tier dem Zoo hinzufügen
void Zoo::hinzu(const Tier & t) {
    // Wenn ein Elefant hinter/nach einer Maus eingefügt wird,
    // dann Ausnahme werfen
    // Nutzen Sie die Präprozessormakros __LINE__ und __FILE__
    // HIER

    // sonst Tier hinzufügen
    // HIER

}
```

```
// Alle Zootiere ausgeben
void Zoo::print() const {
    tiere.print();
}
```

```
#include <iostream>
#include "Zoo.hpp"
using namespace std;

int main() {
    string str;
    char c = 'X';
    // Ausnahmeprüfung aktivieren
    // HIER

        cout << "Bitte Name des Zoos eingeben: ";
        cin >> str;
        Zoo z(str);
        while (c != 'e') {
            cout << endl << "Bitte Tierart auswaehlen:" << endl;
            cout << "1 = Elefant" << endl;
            cout << "2 = Tiger" << endl;
            cout << "3 = Maus" << endl;
            cout << "e = Ende mit Eingabe" << endl;
            cout << "Eingabe: ";
            cin >> c;
            if (c != 'e') {
                cout << "Bitte Namen des Tieres eingeben: ";
                cin >> str;
                switch (c) {
                case '1': z.hinzu(Elefant(str)); break;
                case '2': z.hinzu(Tiger(str)); break;
                case '3': z.hinzu(Maus(str)); break;
                case 'e': break;
                default: // Einen String als Ausnahme werfen
                    // HIER

                }
            }
            cout << endl;
            z.print();
        }

    // Ausnahmen auffangen
    // Speziellste Ausnahme auffangen und ausgeben
    // HIER

    // MyException auffangen und ausgeben
```

```
    // HIER

    // Alle anderen Standardausnahmen auffangen und ausgeben
    // HIER

    // Alle anderen Ausnahmen auffangen
    // HIER

    return 0;
}
```

Aufgabe 3: Stack Unwinding

Überlegen Sie zuerst die Konstruktor- und Destruktoraufrufe sowie die Ausgaben in dem folgenden Programm bei verschiedenen Eingaben. Verifizieren Sie dann Ihre Ergebnisse an Hand von Programmläufen und beantworten sie folgende Fragen:

a) Wann werden die Variablen eines Blockes gelöscht (freigegeben), wenn in dem Block eine Ausnahme geworfen wird.
   i. Direkt nach dem Werfen der Ausnahme.
   ii. Direkt nach dem Auffangen der Ausnahme.
   iii. Am Ende des Programms.

b) Wann werden die Variablen eines äußeren Blockes gelöscht (freigegeben), wenn in einem inneren Block eine Ausnahme geworfen wird, i.e. `{ int i; { throw ... }}`.
   i. Direkt nach dem Werfen der Ausnahme.
   ii. Direkt nach dem Auffangen der Ausnahme.
   iii. Am Ende des Programms.

c) Wann wird ein geworfenes Objekt gelöscht?

d) Wann wird ein anonym aufgefangenes Objekt gelöscht?

e) Wann wird ein referenziertes aufgefangenes Objekt gelöscht?

f) Wann wird ein im Handlerparameter kopiertes Ausnahmeobjekt gelöscht?

```
1  #include <iostream>
2  #include <sstream>
3  #include "ObjectCounter.hpp"
4  using namespace std;
5
6  class Objekt : public ObjectCounter {
7      string name;
8  public:
9      Objekt(string n) : name(n) {
10         cout << "Konstruktor der Klasse Objekt  ";
11         print();
12     }
13     Objekt(const Objekt & o) : name(o.name) {
14         cout << "Kopierkonstruktor der Klasse Objekt  ";
15         print();
16     }
17     ~Objekt() {
```

```
18          cout << "Destruktor der Klasse Objekt  ";
19          print();
20      }
21      void print() {
22          cout << "Objekt-Id = " << getId();
23          cout << "   Name = " << name << endl;
24      }
25  };
26
27  unsigned int ObjectCounter::maxId = 0;
28  unsigned int ObjectCounter::number = 0;
29
30  string iToS(int i) {
31      stringstream ss;
32      ss << i;
33      return ss.str();
34  }
35
36  void f(int i) {
37      cout << "Aufruf f(" << i << ")" << endl;
38      Objekt o("f(" + iToS(i) + ")");
39      if (i > 1) {
40          f(--i);
41      }
42      if (i == 1) {
43          cout << "Ausnahme in f(" + iToS(i) + ") werfen ..." << endl
                ;
44          throw Objekt("Ausnahmeobjekt in f(" + iToS(i) + ")");
45      }
46      cout << "Am Ende der Funktion f(" + iToS(i) + ")";
47  }
48
49  int main() {
50      cout << "Am Beginn Hauptfunktion" << endl;
51      Objekt o("main");
52      int i = 0;
53      cout << "Eine ganze Zahl >= 0 eingeben: " << endl;
54      cin >> i;
55      try {
56          try {
57              try {
58                  f(i);
59              }
60              catch (Objekt) {
61                  cout << "Handler 1: " << endl;
62                  cout << "Anonymes Objekt aufgefangen und
                        weitergeworfen ..." << endl;
63                  throw;
```

```
64              }
65          }
66          catch (Objekt & e) {
67              cout << "Handler 2: " << endl;
68              cout << "Referenziertes Objekt aufgefangen und
                    weitergeworfen ..." << endl;
69              throw e;
70          }
71      }
72      catch (Objekt e) {
73          cout << "Handler 3: " << endl;
74          cout << "Kopiertes Objekt aufgefangen ..." << endl;
75          e.print();
76      }
77      catch (...) {
78          cout << "Defaulthandler" << endl;
79      }
80      cout << "Am Ende der Hauptfunktion" << endl;
81      return 0;
82  }
```

Aufgabe 4: Ausnahmen für MyVector

In der Klasse `MyVector` aus Hausaufgabe 5 kann mit der Methode `at()` theoretisch auch außerhalb des Vektors zugegriffen werden. Da die Methode einen Wert zurückgeben muss, mussten wir bisher auch bei zu großem `i` ein Objekt des Vektors zurückgeben. Mit Ausnahmen können wir das nun ändern.

Ändern Sie Ihre Methode `at(...)` so ab, dass eine Ausnahme geworfen wird, wenn der Index `i` zu groß ist. Schreiben Sie ein Hauptprogramm, wo Sie dies austesten und die Ausnahme abfangen.

Aufgabe 5: Ausnahmen in Konstruktoren

Es kann passieren, dass in einem Konstruktor eine Ausnahme auftritt. Schreiben Sie in dem kleinen Hauptprogramm einen Handler, der die verursachte Ausnahme auffängt.

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

class Test {
    char * ptr;
public:
    Test() {
        ptr = new char[500000000];
    }
};

int main() {
    try {
```

```
        int i = 1;
        while (true) {
            cout << i++ << endl;
            new Test;
        }
    }
    // HIER

    return 0;
}
```

Aufgabe 6: File Streams

Die Datei `wetterdaten.csv` enthält Wetterdaten des Deutschen Wetterdienstes in etwas abgespeckter Form. Die folgende Tabelle zeigt den prinzipiellen Aufbau der Daten. Die erste Zeile enthält Metadaten und zeigt den Inhalt der einzelnen Spalten an. Jede weitere Zeile enthält einen Datensatz mit durch Strichpunkte getrennten Wetterdaten. Am Ende jeder Zeile ist der Text `eor` (end of record) gespeichert. In der Datei kann die Spaltenbreite, im Gegensatz zu der gezeigten Tabelle, variieren. Wichtig ist die bei einer CSV-Datei üblichen Trennung der Spalten. In diesem Falle werden als Trennzeichen Strichpunkte eingesetzt.

```
StationsID;Messdatum;Qualität;RelFeuchte;LuftTemp;LuftDruck;Wind;Niederschlag;eor
      4931; 20121026;       3;     89.88;     8.0;    959.68; 2.6;         8.1;eor
      4931; 20121027;       3;     96.67;     1.2;    953.98; 4.9;         6.5;eor
      4931; 20121028;       3;     90.04;     0.3;    966.28; 3.3;         0.0;eor
      4931; 20121029;       3;     87.21;    -0.2;    969.28; 1.6;         1.3;eor
      4931; 20121030;       3;     79.13;     4.4;    962.00; 2.2;         0.0;eor
      4931; 20121031;       3;     79.54;     4.4;    955.50; 1.3;         4.7;eor
```

In dieser Aufgabe sollen Sie die Wetterdaten einlesen, die Mittelwerte pro Monat berechnen und die Ergebnisdaten in einer weiteren Datei `monatswetter.csv` speichern.

6.1 Die Klasse `Datum` ist eine Hilfsklasse. Ihre wichtigste Aufgabe ist das Konvertieren von Strings der Form "JJJJMMTT", wie sie in den Wetterdaten vorkommen. In der Klasse werden keine Prüfungen durchgeführt, ob ein Datum korrekt ist. Daher kann beispielsweise auch ein Tag 0 vorkommen, um dadurch einen Monat zu speichern: "JJJJMM00". Das wird bei der Speicherung der Monatsmittelwerte ausgenutzt. Implementieren Sie die Methoden der Klasse.

```
// Datei: Datum.hpp

#ifndef DATUM
#define DATUM

#include <string>
using namespace std;

/* Die Klasse Datum speichert Tag, Monat und Jahr
   und erlaubt den Direktzugriff auf diese Variablen.
   Verschiedene Konstruktoren erlauben das Erzeugen
```

```
   von Objekten. Ausserdem gibt es eine Methode, um
   einen String zu parsen und in die Datums-Komponenten
   zu zerlegen, sowie eine Methode, um die Stringform
   eines Datumsobjektes zu erzeugen.
*/

class Datum {
public:

    int tag, monat, jahr;

    // Datum muss als int in Form einer 8-stelligen Zahl
    // vorliegen: JJJJMMTT
    Datum (int d = 0);

    // Datum kann auf einen beliebigen Wert gesetzt werden.
    // Mit t=0 kann man einen Monat und mit t=0 und m=0
    // kann man ein Jahr bestimmen.
    Datum (int j, int m, int t);

    // Der String muss in der Form einer 8-stelligen Zahl
    // vorliegen: JJJJMMTT. Aus diesem String werden die
    // Attribute des aktuellen (this) Objektes berechnet.
    void toParse(string s);

    // Die Werte des aktuellen Objektes werden in einen
    // String in der Form einer 8-stelligen Zahl
    // JJJJMMTT gewandelt.
    string toString() const;
};

#endif
```

6.2 Die Klasse `WetterDaten` dient dazu, einen Datensatz aus der Datei aufzunehmen. Dabei sind von Ihnen die beiden Konstruktoren und die Methode `toString()` zu implementieren.

```
// Datei: WetterDaten.hpp

#ifndef WETTERDATEN
#define WETTERDATEN

#include <string>
#include "Datum.hpp"

using namespace std;

/* Ein Objekt der Klasse Wetterdaten speichert einen
   Satz von gemessenen Daten. Dazu wird die Identitaet
```

```
   der Wetterstation und das Messdatum gespeichert.
*/

class WetterDaten {
    int stationsID;
    Datum messDatum;
    int qualiNiveau;
    double relFeuchte;
    double luftTemp;
    double luftDruck;
    double windGeschw;
    double niederschlag;

public:
    int getStationsID() const {return stationsID;}
    Datum getMessDatum() const {return messDatum;}
    int getQualiNiveau() const {return qualiNiveau;}
    double getRelFeuchte() const {return relFeuchte;}
    double getLuftTemp() const {return luftTemp;}
    double getLuftDruck() const {return luftDruck;}
    double getWindGeschw() const {return windGeschw;}
    double getNiederschlag() const {return niederschlag;}


    // Der Eingabestring enthaelt die einzelnen
    // Messdaten getrennt durch Semikolon. Am Ende
    // des Satzes muss ein "eor" stehen.
    WetterDaten(std::string toParse);
    WetterDaten(int Id, Datum datum, int quali,
        double relF, double luftT, double ldruck,
        double wind, double regen);

    // Das Ergebnis ist ein String mit den Wetterdaten
    // getrennt durch Semikolon und abgeschlossen durch
    // ein "eor"
    std::string toString() const;

    // Nur fuer Testzwecke zur Ausgabe auf die Konsole:
    void print() const;
};

#endif
```

6.3 Die Klasse `WetterMittel` unterstützt die Berechnung der Mittelwerte. Dabei werden hauptsächlich statische Variablen und Methoden benutzt.

```
// Datei: WetterMittel.hpp

#ifndef WETTERMITTEL
```

```
#define WETTERMITTEL

#include "WetterDaten.hpp"

/* Die Klasse dient dazu die Mittelwerte von Messdaten
   zu berechnen. Dazu werden die Daten mit einem Wert
   initialisiert. Weitere Werte werden aufsummiert.
   Bei Abfrage des Ergebnisses werden die Summen
   noch durch die Anzahl der aufsummierten Datensaetze
   dividiert. Das Datum des Ergebnis-Datensatzes kann vor
   der Abfrage auf einen beliebigen Wert gesetzt werden.
*/

class WetterMittel {
    static int stationsID;
    static Datum messDatum;
    static int qualiNiveau;
    static double relFeuchte;
    static double luftTemp;
    static double luftDruck;
    static double windGeschw;
    static double niederschlag;
    static int anzahl;

public:

    // Initialisiert die Klassenvariablen mit einem
    // Wetterdatensatz. Setzt die Anzahl auf 0.
    static void initialisiereWerte (const WetterDaten * wd);

    // Addiert die Daten des Wetterdatensatzes zu den
    // Klassenvariablen hinzu. Stationsid und Messdatum
    // werden hierbei nicht veraendert
    static void summiereWerte (const WetterDaten * wd);

    // Berechnet die Mittelwerte und gibt das Ergebnis
    // als Wetterdatensatz zurueck.
    static WetterDaten * getMittelwerte();

    // Setzt das Datum fuer den Ergebnis-Datensatz,
    // um beispielsweise festzuhalten, dass die
    // Mittelwerte fuer einen bestimmten Monat berechnet
    // wurden (d.tag == 0) oder fuer ein bestimmtes Jahr
    // (d.tag == 0) && (d.monat == 0)
    static void setDatum (Datum d) {messDatum = d;}

};
```

```
#endif;
```

6.4 Schreiben Sie ein Hauptprogramm, das die Datei `wetterdaten.csv` zeilenweise einliest. Die erste Zeile kann überlesen werden, da sie keine eigentlichen Daten enthält. Berechnen Sie für die anderen Zeilen mit Hilfe der Klasse `WetterMittel` die Mittelwerte der Daten pro Monat. Die anderen Zeilen, also die Zeilen mit den eigentlichen Messwerten, sind aufsteigend nach dem Datum (`Messdatum`) geordnet. Der Einfachheit halber gehören alle Daten zur gleichen Messstation, die `StationsID` ist also für alle Daten gleich. Berechnen Sie mit Hilfe der Klasse `WetterMittel` die Mittelwerte der Daten pro Monat. Wenn sich der Monat in einem eingelesenen Datensatz ändert, ist ein Monat vorbei und der zugehörige Mittelwert muss in die Datei `monatswetter-<StationsID>.csv` geschrieben werden. Die Daten der Klasse `WetterMittel` müssen neu initialisiert werden, um den nächsten Monat zu bearbeiten. Bearbeiten Sie auf diese Weise alle Datensätze der Datei `wetterdaten.csv`.

a) Öffnen Sie die Datei `wetterdaten.csv` nur zum Lesen. Prüfen Sie, ob die Datei korrekt geöffnet werden konnte. Wenn nicht, geben Sie eine Fehlermeldung aus und beenden Sie das Programm.

b) Überlesen Sie die erste Zeile und lesen Sie dann den ersten Datensatz in der zweiten Zeile. Sollten beim Lesen Fehler auftreten, beenden Sie das Programm mit einer Fehlermeldung.

c) Berechnen Sie den Dateinamen `monatswetter-<StationsID>.csv` durch Einfügen der ID der Messstation aus dem ersten Datensatz. Öffnen Sie die Datei zum Schreiben. Prüfen Sie auch hier, ob die Datei geöffnet werden konnte.

d) Verarbeiten Sie nun alle Wetterdaten wie bereits oben skizziert und schreiben Sie die berechneten Monatsmittelwerte in die Ausgabedatei.

e) Am Ende der Verarbeitung sind beide Dateien auch wieder zu schließen.

f) Zählen Sie die gelesenen Wetterdatensätze und geben Sie am Ende des Programms auf die Standardausgabe eine Meldung aus, die den korrekten Ablauf des Programms verkündet und die Anzahl der verarbeiteten Datensätze enthält.