Aufgaben

1. **Solution und Projekt unter Visual Studio anlegen (2 Pkt):**

   a. Erstellen Sie unter Visual Studio eine leere Solution mit dem Namen **OOP_LB_KL1_Nachname.Vorname**.

   b. Erstellen Sie in der oben erstellten Solution ein neues C++ Windows Console Projekt mit dem Namen **FHHumans**.

   c. Downloaden Sie aus Moodle main_mit_testcode.cpp und ersetzen Sie den vom Visual Studio generierten Code mit dem aus Moodle geladenen.

2. **Klassen / Vererbung / Konstruktoren / Setter und Getter (8+3+5+4+3+3 Pkt)**

   Ihre **Klassen** der Applikation FHHumans sollten Personen, Studenten und Lehrer verwalten. Folgende Attribute sollten den jeweiligen Individuen zugeordnet sein:

Person:	**Name** (Zeichenkette, var. Länge, Format *„Nachname, Vornamen“*) **Geburtsdatum** (=>numerisch, im Format *YYYYMMDD*) **Geschlecht** (nur *männlich*, *weiblich* und *divers* möglich)
Student:	alle Daten einer **Person** **Matrikelnummer** (=> ganzzahlig, exakt 10-stellig)
Teacher:	alle Daten einer **Person** „kurze“ **Sozialversicherungsnummer** (=> ganzzahlig, exakt 4-stellig)

   a. Erstellen Sie **Klassen** für die Verwaltung der Entitäten ***Person***, ***Student*** und ***Teacher***. Getter und Setter müssen für den Zugriff auf die Attribute NICHT implementiert werden. Der Zugriff auf die **SVNr** sollte aber **nur abgeleiteten Klassen oder Freunden** der Klasse *Teacher* möglich sein.

   b. Erweitern Sie **Teacher** nun so, dass eine **erweiterte Sozialversicherungsnummer** über einen eigenen Getter **getExtSVNr()** abgefragt werden kann. Die erweiterte SVNr setzt sich aus der kurzen SVNr (SSSS) und dem Geburtsdatum (YYYYMMDD) im **Format „SSSS-DDMMYY“** zusammen.

   c. Erstellen Sie eine **Klasse** ***Email*** zur Speicherung einer **Emailadresse**. Die Emailadresse muss vorerst nicht validiert werden, es sollte aber eine **getter** und ein **setter** implementiert werden.

d. Verwenden Sie Ihre Klasse **Email** nun so, dass **durch Vererbung** (!!) sowohl einem **Student** als auch einem **Teacher** eine **Emailadresse** zugeteilt werden kann.

e. Verwenden Sie wahlweise den im main verfügbaren Testcode, zum Instanziieren der folgenden Entitäten:

Person:	Name: „Mustermann, Max“ Geb.Dat: 19770707 Geschlecht: männllich
Student:	Name: „Example, Eva Maria“ Geb.Dat: 19970217 Geschlecht: weiblich MatNr: 1910581005 Email: evae@fhhmail.com *
Teacher:	Name: „Tempo, Theox“ Geb.Dat.: 20010605 Geschlecht: divers SVNr: 8976 Email: ttempo@fhhmail.com *
Student:	**Fügen Sie auch Ihre eigenen Studentendaten in den Testcode ein!**

3. Namespace / Function Template / Strings (2+12 Pkt)

Sie benötigen später zur Ausgabe in eine Textdatei eine oder mehrere Function Templates bzw. überladene Funktionen.

a. Erstellen Sie eine Include-Datei mit dem Namen **PrintExt.h** und legen darin einen neuen **Namespace** mit dem Namen **PrintExt** an.

Sie benötigen Funktionen, die eine Zeichenkette p1, als auch einen Parameter p2 beliebigen Typs entgegennehmen. Die Funktionen sollte als Rückgabewert einen einzelnen String in der Form *“p1“=p2* oder *“p1“=“p2“* liefern. Strings werden dabei grundsätzlich unter doppelte Anführungszeichen gestellt, alle numerischen Werte aber nicht. Zwei Beispiele dazu:

Für p1 mit „Name“ und p2 „Mustermann, Max“ würde folgender String erzeugt:
„Name“=„Mustermann, Max“
Für p1 mit „SVNr“ und p2 mit einem 1234 würde folgender String erzeugt:
„SVNr“=1234

b. Schreiben Sie im Namensraums PrintExt eine typunabhängige Template Function **`to_PrintItemStr`**, die obige Anforderungen erfüllt und einen String sowie einen beliebigen Typ entgegennimmt und einen formatierten String wie oben beschrieben zurückgibt.

   Schreiben Sie einige Zeilen zum Testen in Ihre main(), indem Sie einmal einen Integer und ein andermal einen String als für den Parameter p2 übergeben.

4. Container / Operator Overloading / Polymorphismus (4+6+8+10+4 Pkt)

Wenn dies alles funktioniert verwenden wir diese Funktion nun wie folgt in unserem Programm. Wir benötigen nun einen Container mit mehreren Testdaten.

a. Speichern Sie hierfür in der main() die Zeiger mehrere unterschiedliche Entitäten bzw. -typen in einem beliebigen **Container** mit dem Namen ***`persons`***.
   Wählen Sie einen Containertyp aus, der **Smart Pointer** beliebig vieler Person- bzw. Student- oder Teacher-Instanzen aufnehmen kann.

b. Befüllen Sie den **Container mit „smarten Zeigern“** auf neu instanziierte Testdaten. Sie können beliebige Objekte erzeugen, es sollte jedoch **jeder Personentyp zumindest einmal** vorkommen. Fügen Sie auch wieder Ihre eigenen Daten ein.

Nun kümmern wir uns um die Ausgabe der Person bzw. Student und Teacher auf *`cout`* im Textformat. Die Ausgabe hat als speziell formatierte Textdatei zu erfolgen.

Am Beginn einer Zeile soll immer der Personentyp als String gefolgt von einem „= { “ stehen. Danach folgen die jeweiligen Items in der Form „itemname“=value, die voneinander mit Komma getrennt sind. Itemnamen und Strings sollten immer zwischen doppelten Anführungszeichen stehen, numerische Werte aber nicht. Ein Personentyp wird immer mit einem „}“ abgeschlossen. Personentypen können in der Ausgabe geschachtelt sein (z.B. ist ein Student ja auch eine Person).

Beispiel: Ausgabe einer Person und eines Studenten

```
person = { „name“=„Mustermann, Max“, „birthdate“= 19770707, „Gender“=“m“ } <CR>
student = { person={„name“=„Example, Eva Maria“, „birthdate“= 19970217,
          „Gender“=“f“}, „Matnr“=1910581005, „Email“=“evae@fhhmail.com“}   <CR>
```

c. Überladen Sie nun den **stream operator** so, dass Objekte vom Typ Person bzw. Student oder Teacher wie oben beschrieben **auf `std::cout` ausgegeben** werden.

d. Iterieren Sie nun durch den von Ihnen gewählten **Container *persons*** und geben Sie über den zuvor implementierten **stream operator** jede Person bzw. Teacher oder Student des Containers auf std::*cout* aus.

   Der Code sähe sinngemäß wie folgt aus:

```
Schleife( über alle Items der Collection persons ){
    std::cout << item;
}
```

   Welcher Typ nun jeweils wie ausgegeben wird, sollte natürlich automatisch über den zugrundeliegenden Typ des Zeigers erfolgen (=> **dynamischer Polymorphismus**).

5. Lambda Funktion (10 Pkt)

a. Schreiben Sie nun einen Code, der nun mittels einer **Lambda Funktion** und mit Hilfe von **std::transform** die **Nachnamen** jedes Individuum **in Großbuchstaben** konvertiert.

6. Exception Handling (8+6+2 Pkt)

Führen Sie eine Validierung der Emailadressen ein.

Validieren Sie an passenden Stellen im Code die **Emailadresse** auf das Vorhandensein eines
- lokalen Teils (i.d.R. ist dies das Pseudonym des Emailkontos)
- eines Klammeraffen (@) und
- eines Domänenteils (Hostnamen + Top-Level-Domain)

a. Schreiben Sie in der Klasse Email eine statische Methode **`validate`**. Werfen Sie in dieser Methode **Exceptions**, wenn die der Methode übergebene Emailadresse die verpflichtenden Elemente einer Emailadresse nicht enthält.

b. Verwenden Sie den vorhandenen Testcode am Ende von main() mit **Ihrer eigenen Emailadresse**. Vervollständigen Sie die Funktion check_email_validation, indem Sie aus Email die Methode **validate** aufrufen. Fangen Sie **ALLE** Exceptions ab, um eine passende Fehlermeldung auf std::cerr auszugeben.

c. Stellen Sie mit Hilfe Ihrer neuen Methode **`validate`** sicher, dass in der Klasse Email nur eine valide Emailadresse oder ein Leerstring gespeichert werden kann.

Abgabe der Klausur als Visual Studio Solution

- Klicken Sie unter Visual Studio mit der **rechten Maustaste im Solution Explorer** auf den Solutionname (erste Zeile).
  Wählen Sie den Menüpunkt **Clean Solution** aus. Damit sollen (fast) alle Compilate und temporären Dateien gelöscht sein.
- Löschen Sie das .vs Verzeichnis (Achtung: hidden folder!) im Verzeichnis der Solution, bevor Sie das zip-File der Solution erstellen.
- Die Abgabe erfolgt im Moodle unter Assignment KLAUSUR 1.
  Beachten Sie, dass es für jede Gruppe einen bestimmten Upload Link gibt.

- bis spätesten **5 Minuten nach dem Ende der Klausurzeit**