

20. Ich habe Dependency Injection noch nicht ganz verstanden. Was genau ist die Dependency Injection? Was wird dabei gemacht? Ich finde das Beispiel im Lernpfad nicht anschaulich. Was ist ein besseres Beispiel für Dependency Injection?

Das DIP findet Anwendung zwischen Modulen, also zusammengehörigen Klassen. Es ermöglicht es, dem Anwender der externen Module das Interface nach seinen Wünschen zu schreiben, welches dann von den externen Modulen implementiert wird. Dies ermöglicht es beispielsweise, dass die Business-Logik nicht von der Implementierung der Peripherie abhängig ist, sondern die Peripherie von der Business-Logik. Das einfachste Beispiel stellt der Wechsel zwischen Ausgabe per GUI und Ausgabe per CLI dar. Siehe Java-Beispiel.

21. Überladen eines Operators (Kapitel D1):

```
bool operator==(const Tier& a) {  
    return a.name_ == name_;  
}
```

Wird der Operator == hier quasi neu definiert und dient dann innerhalb des struct Blocks ausschließlich dazu die Namen zu vergleichen?

Muss bei Überladungen der zu überladende Operator wie eine Funktion definiert werden und die verwendeten Parameter angegeben werden (const Tier& a)? Ist dabei Tier der Typ und a der Bezeichner? Was genau bedeutet die Zeile (const Tier& a)?

Operatoren in C++ sind letzten Endes auch nur Funktionen mit besonderer Syntax. Als solches lassen sie sich genauso definieren wie eine „normale“ Funktion. Es gelten alle Regeln, was Parameter angeht bei Operatoren wie bei normalen Funktionen. Const Tier& a ist im Prinzip genau dasselbe, wie const int& a. Nur halt, dass wir diesmal keinen Integer sondern ein Tier(-struct) übergeben. Tier ist hier der Typ und a der Bezeichner.

22. Überladen des Linksshift-Operators (Kapitel D1):

```
ostream& operator<<(ostream& out) {  
    return drucken(out);  
}
```

Wird hier << neu definiert und ruft die Funktion drucken auf, wenn man ihn innerhalb des struct Blocks benutzt?

Ja. Der << Operator delegiert hier an die drucken Funktion.

23. Überladen des Konstruktors (Kapitel D1):

Was ist der Sinn des Überladens eines Konstruktors?

```
// Konstruktor für die Struktur Tier  
Tier() : rasse_{UNGEKLAERT}, name_{UNGEKLAERT},  
        gewicht_{UNGUELTIG}, alter_{UNGUELTIG}, istSaeugetier_{true} {  
    // Weitere Anweisungen wenn das Tier Objekt erstellt wird  
}  
// Überladung des Konstruktor für die Struktur Tier  
Tier(string rasse, bool istSaeugetier) : rasse_{rasse},  
        name_{UNGEKLAERT}, gewicht_{UNGUELTIG},  
        alter_{UNGUELTIG}, istSaeugetier_{istSaeugetier} { }  
// Überladung mit Vertauschung der Stellen  
Tier(string name, string rasse, bool istSaeugetier)  
    : rasse_{rasse}, name_{name}, gewicht_{UNGUELTIG},  
    alter_{UNGUELTIG}, istSaeugetier_{istSaeugetier} { }
```

Ein Konstruktor wird überladen, um verschiedene Möglichkeiten der Erzeugung des Objekts zu ermöglichen. Wenn ich nichts über das Tier, welches ich haben möchte weiß, kann ich den ersten Konstruktor also Tier() verwenden. Wenn ich den Namen und den Säugetierstatus weiß, kann ich den zweiten Konstruktor verwenden Tier(„Mausi“, true). Wenn ich zudem auch noch die Rasse weiß, kann ich den dritten Konstruktor verwenden Tier(„Katze“, „Mausi“, true).

24. Musterlösung Übungaufgabe D1:

Diesen Code verstehe ich nicht ganz:

```
std::ostream& operator<<(std::ostream& out, Mensch& m) {  
    // Der Parameter m vom Typ Mensch soll gedruckt werden
```

```
    return m.drucke(out);
}
```

Was genau bedeutet er?

Das sind meine Überlegungen dazu:

Der Linkshift-Operator << wird überladen.

<< soll vom Typ ostream sein und hat die Parameter out vom Typ ostream und m vom Typ Mensch.

drucke ist eine Methode von m.

Wir haben hier eine Delegation von operator<< zu Mensch.drucke(std::ostream&). Die Funktion operator<< empfängt hierfür einen ostream out sowie einen Mensch m und ruft dann die Funktion drucke der Klasse Mensch auf dem Objekt m auf. Der ostream wird zurückgegeben, um Method-Chaining zu ermöglichen.

25. Warum soll man bei der Rückgabe aus Funktionen darauf achten, dass man keine Referenz auf eine Variable zurückgibt, die in derselben Funktion deklariert wurde? (Kapitel D2)

Wenn die Funktion endet, wird ihr Speicher gelöscht (als unbelegt und damit undefiniert gekennzeichnet). Die Referenz zeigt nun also auf eine unbelegte Stelle und erhält dadurch undefiniertes Verhalten, da sich der Speicher, auf welchen die Referenz zeigt beliebig ändern kann.

26. Halbautomatische Typumwandlung (Kapitel D2)

Beispielcode Halbautomatische Typumwandlung:

```
-----
#include <iostream>
#include <string>
class Vorname{
    std::string name;
public:
    explicit Vorname(std::string name) : name{name} {}

    // Was genau passiert hier?
    operator std::string() const { return name; }
};

int main() {
    std::cout << static_cast<std::string>(Vorname{"Mia"}) << "\n";
    return 0;
}
-----
```

Der Konvertierungsoperator für std::string() wird hier definiert. Dieser wird, wie unten zu sehen, bei einem Cast aufgerufen. static_cast<std::string>(Vorname{"Mia"}) kann man also auch verstehen als Vorname{"Mia"}.std::string()

Ein Konstruktor mit nur einem Parameter (wie oben) kann ebenfalls als Konvertierungsoperator (automatisch) verwendet werden. Hier wurde dies durch das Schlüsselwort „explicit“ jedoch verwehrt.

27. Namensräume (Kapitel D2)

modul.hpp:

```
-----
#ifndef MODUL_HPP
#define MODUL_HPP
// Rückgabetyt der beiden Funktionen wird deklariert
int erhoehenInNamespace();
int erhoehenInGlobal();
#endif
-----
```

modul.cpp:

```
-----
#include "modul.hpp"
```

```
// Deklaration und Initialisierung von j
static int j = 0;
// Deklaration und Initialisierung von i
int i = 0;
// Implementation der beiden Funktionen
int erhoehenInNamespace() { return ++i; }
int erhoehenInGlobal() { return ++j; }
-----
main.cpp:
-----
#include<iostream>
#include "modul.hpp"
// Funktioniert, da j nur in modul.cpp deklariert ist
// Es ist ein anderes j als das, was weiter unten in der main Funktion verwendet wird.
int j = 1;
// Durch extern wird das globale i auch in dieser Datei bekannt
extern int i;
int main() {
    std::cout << erhoehenInNamespace();
    // hier wird dasselbe j verwendet, wie in der modul.cpp deklariert wurde
    std::cout << erhoehenInGlobal();
    // Nun kann auch i direkt ausgegeben werden.
    std::cout << i;
}
```

Warum kommt bei Anwendung der Funktion erhoehenInGlobal() 1 raus? Müsste nicht 2 ausgegeben werden, da int j = 1; ?

Das j in modul.cpp ist ein anderes Objekt als das j in main.cpp. Durch das Schlüsselwort static wird wie im LMS beschrieben ein anonymer Namensraum erstellt, welcher außerhalb von modul.cpp nicht sichtbar ist. Wird jedoch i in der main-Funktion geändert, so wird das globale i verändert, da wir durch „extern int i“ festgelegt haben, dass wir hier das globale i verwenden wollen.

28. Vererbung (Kapitel E1):

```
-----
#include <iostream>
using std::ostream;
struct Basis {
    int wert() const { return 0; }
    ostream& drucken(ostream& os) {
        return os << wert() << "\n";
    }
};
// Warum wird diese Überladung des Operators << außerhalb der Klasse implementiert?
// Kindklassen können doch auch Methoden erben, oder?
ostream& operator<<(ostream& left, Basis& right) {
    return right.drucken(left);
}
struct Wert : public Basis {
    // Überschreibt wert() der Oberklasse Basis
    int wert() const { return 10; }
};
int main() {
    Basis basis{};
    Wert kind{};
    std::cout << basis;
    // Druckt 0 aus, obwohl wert überschrieben wurde, denn es wird die Methode benutzt, die in
    // der Elternklasse definiert wurde.
    std::cout << kind;
}
```

Warum wird diese Überladung des Operators << außerhalb der Klasse implementiert? Kindklassen können doch auch Methoden erben, oder?

Ja, Kindklassen können auch Methoden erben. Der Operator << Definition in der Klasse hat jedoch eine ungewohnte Syntax zur Folge. Siehe [KTCPP/E1_2 \(github.com\)](https://github.com/KTCPP/E1_2)

29. Typumwandlung als Kopie (Kapitel E1):

Beispielcode:

```
-----
#include <iostream>
#include <vector>
using std::ostream;
// Oberklasse Basis
struct Basis {
    // virtuelle Methode, die überschrieben werden kann
    virtual int wert() { return 0; }
};
// Kindklasse Kind erbt mindestens public (also alles) von Basis
struct Kind : public Basis {
    // Methode wert() kann dank virtual in er Elternklasse überschrieben werden.
    int wert() override { return 1; }
};
// Funktion außerhalb der Klassen
void drucken(Basis b) {
    // Wendet die Methode (Funktion) wert() auf das Objekt b, welches den Typ Basis hat, an.
    std::cout << b.wert() << "\n";
}
int main() {
    // Erstellt ein Objekt basis der Klasse (vom Typ) Basis
    Basis basis{};
    // Erstellt ein Objekt kind der Klasse (vom Typ) Kind
    Kind kind{};

    // gibt 0 aus
    drucken(basis);

    // gibt auch 0 aus,
    // da Kind in Basis kopiert wird
    drucken(kind);
    // Sie können auch eine Unterklasse in einen
    // Vector der Oberklasse stecken
    std::vector<Basis> vec{};
    vec.push_back(basis);
    vec.push_back(kind);
}
```

Warum wird bei drucken(kind) 0 ausgegeben? Weil drucken nur den Parameter kind (Typ Kind) in den Typ Basis umwandelt und dann die Methode wert() der Elternklasse Basis benutzt wird anstatt der Methode wert der Klasse Kind?

Die Funktion drucken() wird durch folgenden Code ersetzt:

```
void drucken(Basis& b) {
    // Wendet die Methode (Funktion) wert() auf das Objekt b, welches den Typ Basis hat, an.
    std::cout << b.wert() << "\n";
}
```

Warum zeigt die Referenz auf die Methode von Kind? Das & bezieht sich doch noch immer auf den Parameter b, der wiederum vom Typ Basis ist.

Beim pass-by-value wird das Objekt Kind upcasted zum Typ Basis. Ähnlich wie mit static_cast. Als solches existiert in der pass-by-value Methode nur ein Objekt vom Typ Basis und die pass-by-value Methode weiß nichts von der Kind-Klasse.

Wird hingegen der Parameter mit pass-by-reference übergeben, so geschieht kein upcast, wodurch das Kind bleibt wie es ist und die überschriebene Funktion wird aufgerufen.

Man kann dies einfach als Konvention verstehen, dass man pass-by-reference verwenden muss, wenn man überschriebene Funktionen aufrufen möchte.

Die genauen technischen Details, die hier stattfinden gehen zu weit für diesen Kurs.

30. Verschiebungen (Kapitel E2)

Beispielcode Daten aus einer CSV Laden:

```
-----
#include <vector>
#include <string>
class Tabelle {
    // Was ist using für ein seltsamer Typ bzw. Anweisung?
    using zelle = std::vector<std::string>;
    std::vector<zelle> zellen;
public:
    explicit Tabelle(const char* datei) { /* ... */ }
    // Destruktor ist notwendig, da Sie eine Datei im Konstruktor öffnen
    // Denken Sie an RAII
    ~Tabelle() { /* ... */ }
};
std::vector<Tabelle> ladeCSVs() {
    std::vector<Tabelle> ergebnis{};
    // Lade Daten aus einer csv-Datei
    ergebnis.push_back(Tabelle{"daten1.csv"});
    ergebnis.push_back(Tabelle{"daten2.csv"});
    ergebnis.push_back(Tabelle{"daten3.csv"});
    return ergebnis;
}
-----
```

Was ist using für ein seltsamer Typ bzw. Anweisung? Was macht using in diesem Fall?

Die Syntax für using ist hier

using „Name“ = „Typ“

das bedeutet letzten Endes nichts anderes, als das überall wo Name steht der Typ eingesetzt werden soll.

Dies hat zum einen den Vorteil, dass man sich Schreibarbeit spart und zum anderen kann man dadurch den verwendeten Typ ähnlich wie bei einem Template ändern.