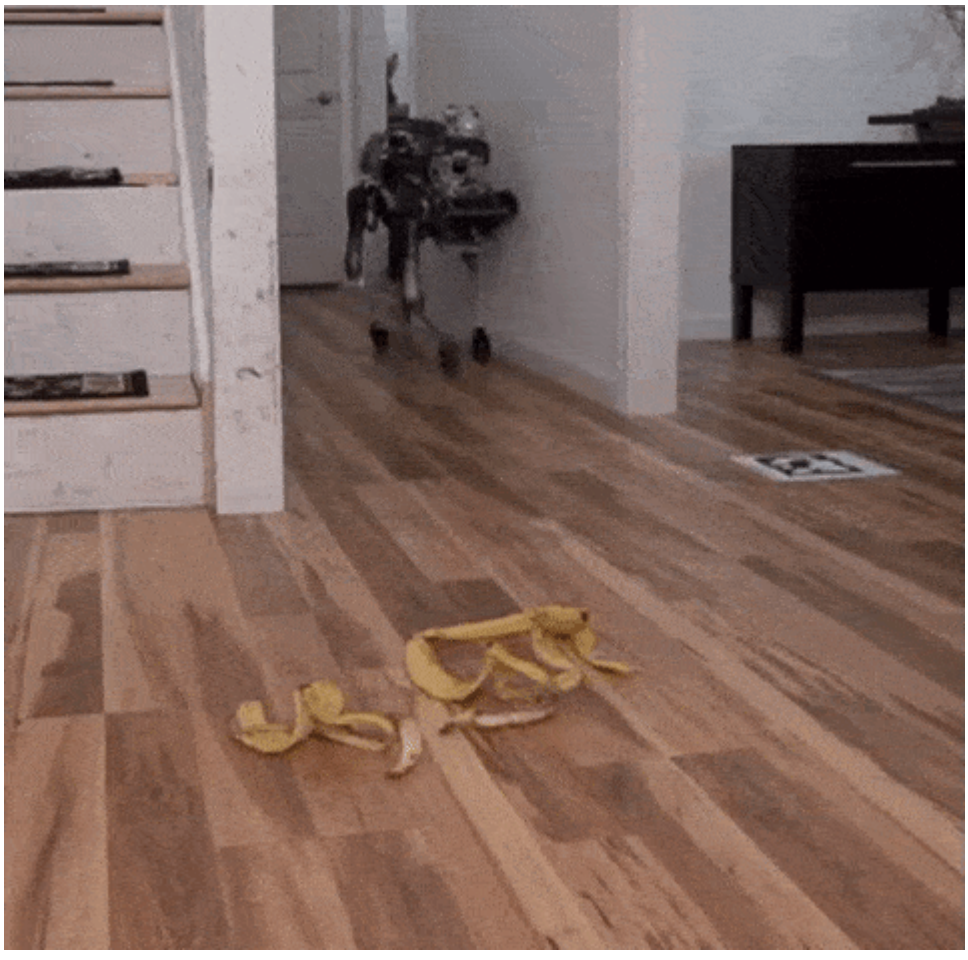
Objektorientierte Programmierung in C++

Parameter	Kursinformationen
Veranstaltung:	`Softwareprojekt Robotik`
Semester	`Wintersemester 2022/23`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Konstruktoren und Operatoren`
Link auf GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_SoftwareprojektRobotik/blob/master/01_OOPinC++.md
Autoren	Sebastian Zug & Georg Jäger



Zielstellung der heutigen Veranstaltung

- Klassen/Strukturen
- Rule of Five
- Operatoren

Von Strukturen und Klassen

Klassen (`class`) und Strukturen (`struct`) unterscheiden sich unter C++ nur in einem Punkt. Während bei erstgenannten immer das Zugriffsattribut private als Default-Wert angenommen wird, ist dies für `struct`s public. Die folgenden Beispiele nutzen structs um C++ spezifische Eigenschaften darzustellen, können aber direkt auf Klassen übertragen werden.

Eine Struktur ist ein Datentyp, der mehrere Variablen gleichen oder verschiedenen Typs zu einem neuen Datentyp zusammenfasst. Die Deklaration erfolgt mit dem Schlüsselwort `struct`.

ApplicationOfStructs.cpp

```
1  #include <iostream>
2
3  struct Student{
4    std::string name;
5    int alter;
6    std::string ort;
7  };   // <- Dieses Semikolon wird gern vergessen :-)
8
9  int main()
10 {
11   Student bernhard = {"Cotta", 25, "Zillbach"};
12   std::cout << bernhard.ort << " " << bernhard.alter  << std::endl;
13   //Student alexander = { .name = "Humboldt" , .alter = 22 , .ort =
       "Berlin"  };
14   //std::cout << alexander.ort << " " << alexander.alter  << std::end
15   return EXIT_SUCCESS;
16 }
```

```
Zillbach 25
```

Elementinitialisierung:

Umsetzung	Beispiel
vollständige Liste in absteigender Folge (uniforme Initialisierung)	`Student Bernhard {"Cotta", 25, "Zillbach"};`
unvollständige Liste (die fehlenden Werte werden durch Standard Defaultwerte ersetzt)	`Student Bernhard {"Cotta", 25};`
vollständig leere Liste, die zum Setzen von Defaultwerten führt	`Student Bernhard {};`
Aggregierende Initialisierung (C++20)	`Student alexander = { .ort = "unknown"};`

C++11 führte die uniformen Initialisierungssyntax ein. In C++20 wird dieser Mechanismus um die die *Designated Initialisers*, die Sie aus C kennen erweitert.

Wo ist bei den *Designated Initialisers* der Unterschied zu C? Die C++ Implementierung integriert nicht:

- eine variable Reihenfolge der Member zu initialisieren.
- die Member eines verschachtelten Aggregates zu initialisieren.
- Designated Initializers und reguläre Initialisierer zu vermischen.

Konstruktoren

Im Grunde können wir unsere drei Datenfelder im vorangegangen Beispiel mit der uniformen Initialisierungssyntax oder dem Designated Initializer in vier Kombinationen initialisieren:

```
{name, alter, ort}
{name, alter}
{name}
{}
```

Eine differenziertere Zuordnung der Reihenfolge `{name = "Zeuner", ort = "Chemnitz"}` unter Auslassung von Student.alter ist nur möglich, wenn hierfür ein default Initializer vorgesehen ist.

Was passiert aber bei dem Aufruf `Student alexander {"Humboldt", 23};`? Der Kompiler generiert uns implizit/automatisch passende Konstruktor(en), wenn Sie gar keinen eigenen Konstruktor generiert haben. Diese werden daher auch *Implicit Constructors* genannt.

Welche Varianten sind für die Erzeugung einer Instanz denkbar, sprich wie kann ich individuelle Mechanismen für die Intialisierung eines Objektes definieren?

1. Erzeugung auf der Basis eines Parametersets mit individuellem Konstruktor

```
int alter = 21;
Student erstsemester(alter);
```

2. Erzeugung auf der Basis einer existierenden Instanz, die als Parameter übergeben wird (*Copy Constructor*)

```
Student erstsemester_template();
Student erstsemester(erstsemester_template);
```

3. Erzeugung auf der Basis einer existierenden Instanz, per Verschiebung (*Move Constructor*)

```
Student erstsemester_template();
...
Student erstsemester (std::move(erstsemester_template));
```

Nicht-Default Basiskonstruktoren

Constructor.cpp

```
1  #include <iostream>
2
3  struct Student{
4    std::string name;
5    int alter;
6    std::string ort;
7
8    Student(); // Default Constructor
9    void printCertificate();
10 };
11
12 void Student::printCertificate(){
13     std::cout << "Student " << this->name << " " << this->alter << "
         the exam!\n";
14 }
15
16 Student::Student() : name {"Cotta"}, alter {18}, ort {"Freiberg"}
17 { }  // keine Funktionalität
18
19 int main()
20 {
21   Student erstsemester {};
22   erstsemester.printCertificate();
23 }
```

```
Student Cotta 18 passed the exam!
```

Delegierende Konstruktoren rufen einen weiteren Konstruktor für die teilweise Initialisierung auf. Damit lassen sich Codeduplikationen, die sich aus der Kombination aller Paramter ergeben, minimieren.

```
Student(std::string n, int a, std::string o): name{n}, alter{a}, ort{o} { }
Student(std::string n) : Student (n, 18, "Freiberg") {};
Student(int a, std::string o): Student ("unknown", a, o) {};
```

Einer Ihrer Kommilitonen kommt auf die Idee einer `init()` Methode, die die Initialsierung übernehmen soll. Was halten Sie von dieser Idee?

Copy-Konstruktoren

Copy-Konstruktoren gehen einen anderen Weg und aggregieren die Informationen unmittelbar aus einer bestehenden Instanz.

Welche Schritte sind im folgenden Beispiel notwendig, um einen Studenten aus der Bachelorliste in die Masterliste zu transferieren? Logischerweise sollte der Student dann nur noch in der Masterliste enthalten sein! Intuitiv würde dies bedeuten:

1. Erzeugen einer neuen Instanz von `Student` und initialisieren mit einer Kopie des existierenden Studenten
2. Löschen des Studenten in der Bachelorliste

Auf diesem Weg bestehen zwischen 1 und 2 letztendlich zwei Kopien des Studenten, was für aufwändigere Datentypen (Bilder, Messungen) zu vermeiden ist!

CopyConstructor.cpp

```cpp
#include <iostream>
#include <list>
#include <iterator>

struct Student{
  std::string name;
  int alter;
  std::string ort;

  Student(std::string n, int a, std::string o);

  Student(const Student&);                // Copy Constructor
  //Student& operator=(const Student&);    // Copy Alignment
};

Student::Student(std::string n, int a, std::string o): name{n}, alter
  ort{o} {}

// Copy-Constructor
Student::Student(const Student& other){
  std::cout << "Copy constructor executed!\n";
  this->name = other.name;
  this->alter = other.alter;
  this->ort = other.ort;
}

void showlist(std::list <Student> g)
{
    for(auto it = g.begin(); it != g.end(); ++it)
        std::cout << it->name << "\n";
    std::cout << '\n';
}

int main()
{
  Student max {"Maier", 19, "Dresden"};
  Student gustav {"Zeuner", 27, "Chemnitz"};
  Student x = gustav;                    // initialization by copy
    constructor
  Student y(max);                        // Also initialization by cop
    constructor
  //y = gustav;                            // assignment by copy assig
    operator
  //std::cout  << y.ort;
  //std::list <Student> bachelor, master;
  //bachelor.push_back(max);
  //master.push_back(gustav);
  //showlist(bachelor);
```

```
  //showlist(master);
}
```

```
Copy constructor executed!
Copy constructor executed!
```

Der Verschiebungskonstruktor löst dieses Problem.

```
Student::Student(Student&& other) noexcept {
  this->name = std::move(other.name);
  this->alter = other.alter;
  this->ort = std::move(other.ort);
}
```

Während das '&' eine Variable als Referenz deklariert, legt '&&' eine 'rvalue-Referenz' an. D.h. eine Referenz auf ein Objekt, dessen Lebensdauer *am Ende ist*. (Mehr dazu später...)

Destruktoren

Destructor.cpp

```
#include <iostream>

struct Student{
  std::string name;
  int alter;
  std::string ort;

  Student(std::string n, int a, std::string o);
  ~Student();
};

Student::Student(std::string n, int a, std::string o): name{n}, alter
  ort{o} {}

Student::~Student(){
  std::cout << "Destructing object of type 'Student' with name = '" <
    ->name << "'\n";
}

int main()
{
  Student max {"Maier", 19, "Dresden"};
  std::cout << "End...\n";
  return 0;
}
```

```
End...
Destructing object of type 'Student' with name = 'Maier'
```

Destruktoren werden aufgerufen, wenn eines der folgenden Ereignisse eintritt:

- Das Programm verlässt den Gültigkeitsbereich (*Scope*, d.h. einen Bereich der mit `{...}` umschlossen ist) eines lokalen Objektes.
- Ein Objekt, das `new`-erzeugt wurde, wird mithilfe von `delete` explizit aufgehoben (Speicherung auf dem Heap)
- Ein Programm endet und es sind globale oder statische Objekte vorhanden.
- Der Destruktor wird unter Verwendung des vollqualifizierten Namens der Funktion explizit aufgerufen.

Einen Destruktor explizit aufzurufen, ist selten notwendig (oder gar eine gute Idee!).

> **Merke:** Ein Destruktor darf keine Exception auslösen!

Operatoren

Mit dem Überladen von Operatoren +, -, *, \/, = kann deren Bedeutung für selbstdefinierter Klassen (fast) mit einer neuen Bedeutung versehen werden. Diese Bedeutung wird durch spezielle Funktionen bzw. Methoden festgelegt.

Im folgenden Beispiel wird der Vergleichsoperator `==` überladen. Dabei sehen wir den Abgleich des Namens und des Alters als ausreichend an.

Allgemeine Operatoren

Comparison.cpp

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

struct Student{
  std::string name;
  int alter;
  std::string ort;

  Student(const Student&);
  Student(std::string n);
  Student(std::string n, int a, std::string o);

  bool operator==(const Student&);
};

Student::Student(std::string n): name{n}, alter{18}, ort{"Freiberg"}{

Student::Student(std::string n, int a, std::string o): name{n}, alter
  ort{o} {
}

Student::Student(const Student& other){
  this->name = other.name;
  this->alter = other.alter;
  this->ort = other.ort;
}

bool Student::operator==(const Student& other){
  if ((this->name == other.name) && (this->alter == other.alter)){
    return true;
  }else{
    return false;
  }
}

int main()
{
  Student gustav {"Zeuner", 27, "Chemnitz"};
  Student alexander {"Humboldt", 22, "Berlin"};
  Student bernhard {"Cotta", 18, "Zillbach"};
  Student gustav2(gustav);
  if (gustav == gustav2){
    std::cout << "Identische Studenten \n";
  }else{
    std::cout << "Ungleiche Identitäten \n";
  }
  std::vector<Student> studentList {gustav, alexander, bernhard, gust
```

```
48    for (auto &i: studentList)
49        std::cout << i.name << ", ";
50    std::cout << std::endl;
51
52    //std::sort(studentList.begin(), studentList.end());
53    //for (auto &i: studentList)
54    //    std::cout << i.name << ", ";
55    //std::cout << std::endl;
56
57  }
```

```
Identische Studenten
Zeuner, Humboldt, Cotta, Zeuner,
```

Analysieren Sie die Hinweise zur Sortiermethode in der Standard-Bibliothek

https://en.cppreference.com/w/cpp/algorithm/sort

Wie kann die entsprechende Sortierfunktion übergeben werden? Wann kann darauf verzichtet werden?

Mit der Operatorüberladung von `<` haben wir ein Sortierkriterium abgebildet. Wie würden Sie vorgehen, wenn sich Ihr Auftraggeber hier eine größere Flexibilität wünscht und ein Set von Metriken bereit gehalten werden soll?

Zuweisungsoperatoren

Zuweisungsoperatoren können in zwei Konfigurationen realisiert werden.

- Kopierend ... die auf der rechten Seite stehende Instanz bleibt erhalten, so dass nach der Operation zwei Objekte bestehen (*Copy Assignment*)
- Verschiebend ... die auf der rechten Seite stehend Instanz wird kopiert, so dass nur die linke Instanz weiter besteht (*Move Assignment*)

Die Unterscheidung der Mechanismen erfolgt anhand der Signaturen der Operatorüberladungen:

```
// copy assignment
myClass& operator=(const myClass& other)
{
  if(this != &other) {
    return *this = myClass(other);      // Aufruf des Copy Constructors
  }
}

// move assignment
myClass& operator=(myClass&& other) noexcept
{
    if(this != &other) { //1. Überprüfung der Zuweisung auf sich selbst
      // ...               2. Freigeben von bisherigen Ressourcen
      // ...               3. elementweises Übertragen
    }
```

```
    return *this;       // 4. Rückgabe des neuen Objektes
}
```

Im Beispiel hat einer Ihrer Kommilitonen das Copy-Assignment implementiert. Die Lösung generiert aber eine unerwartete Ausgabe. Welchem Irrtum ist der Kandidat erlegen?

Assignment.cpp

```cpp
#include <iostream>

struct Student{
  std::string name;
  int alter;
  std::string ort;

  Student(const Student&);
  Student(std::string n);
  Student(std::string n, int a, std::string o);
  Student& operator=(const Student&);
};

Student::Student(std::string n): name{n}, alter{18}, ort{"Freiberg"}{

Student::Student(std::string n, int a, std::string o): name{n}, alter
  ort{o} {
  std::cout << "Non-default constructor executed!\n";
}

// Copy-Constructor
Student::Student(const Student& other){
  std::cout << "Copy constructor executed!\n";
  this->name = other.name;
  this->alter = other.alter;
  this->ort = other.ort;
}

// Copy Assignment
Student& Student::operator=(const Student& other){
  std::cout << "Copy assignment executed!\n";
  if(&other != this){
    this->name = other.name;
    this->alter = other.alter;
    this->ort = other.ort;
  }
  return *this;
}

int main()
{
  Student gustav {"Zeuner", 27, "Chemnitz"};
  Student gustav2(gustav);
  Student gustav3 = gustav;
  gustav.alter = 29;            // Hiermit wird geprüft ob eine unabhä
    Kopien
                                // von gustav entstanden sind.
  std::cout << gustav2.name << " " << gustav2.alter << "\n";
```

```
47    std::cout << gustav3.name << " " << gustav3.alter << "\n";
48 }
```

```
Non-default constructor executed!
Copy constructor executed!
Copy constructor executed!
Zeuner 27
Zeuner 27
```

> **Merke**: Ein Gleichheitszeichen in einer Variablendeklaration ist niemals eine Zuweisung, sondern nur eine andere Schreibweise der Initialisierung! (vgl. Zeile 41 im vorangegangen Codebeispiel)

Rule of Five

Der Kompiler generiert automatisch für Sie:

- einen Standardkonstruktor (wenn Sie gar keinen Konstruktor selbst angelegt haben)
- einen Destruktor
- einen Kopierkonstruktor
- einen Kopierzuweisungsoperator
- den Verschiebekonstruktor (seit C++11)
- den Verschiebeoperator (seit C++11)

Die generierten Versionen haben dabei eine in der Sprachnorm festgelegte Bedeutung: Es werden alle nicht-statischen Datenelemente in der Reihenfolge ihrer Deklaration kopiert (Konstruktoren bzw. Zuweisungen) bzw. in umgekehrter Reihenfolge freigegeben (Destruktor).

Super! Alles gelöst! Warum müssen wir also darüber nachdenken?

Falls eine Klasse jedoch eine andere Semantik hat, z. B. weil sie eine Ressource als Datenelement enthält, die nicht auf diese Weise kopiert oder abgeräumt werden kann, kann jede der genannten Konstruktoren/Destruktoren/Operatoren durch eine eigene Definition ersetzt werden. In den meisten Fällen erfordern solche Klassen dann, dass alle Konstruktoren/Destruktoren/Operatoren eigene, benutzerdefinierte Implementierungen haben.

Beispiel:

```
class Datei
{
public:
    Datei(const char* dateiname)
    : file(fopen(dateiname, "rb"))
    { /* Fehlerbehandlung usw. */ }
```

```
    // Rule of Three (bis C++11):
    Datei(const Datei&) = delete; // Kein Kopieren!
    ~Datei() { fclose(file); }
    void operator=(const Datei&) = delete; // Kein Kopieren!

    // weitere Elementfunktionen
    // ...

private:
    FILE* file;
};
```

Seit C++11 ist es zudem möglich, das Erzeugen der compilergenerierten Version nicht nur explizit zu unterdrücken, sondern auch explizit zu erzwingen (=default). Damit wird dem Compiler (und auch dem menschlichen Leser) mitgeteilt, das in diesem Fall die compilergenerierte Version genau das gewünschte Verhalten bietet, so dass man es nicht manuell implementieren muss:

```
class Example
{
    Example(const Example&) = default;  // erzwinge compilergenerierte Vers
    void operator=(const Example&) = delete; // verhindere compilergenerier
      Version
};
```

Wiederholung - Gültigkeitsbereich von Variablen

Wie lange ist meine Variable, die ich deklariert und initialisiert habe, verfügbar?

C++ definiert dafür 5 Gültigkeitsbereiche einer Variable, die einem jederzeit bewusst sein sollten, um "Irritationen" zu vermeiden:

- Globalen Gültigkeitsbereich - eine Variable oder ein Objekt wird außerhalb von jeder Klasse, Funktion oder Namespace deklariert. C++ ordnet diese automatisch einem globalen Namespace zu.

globalVariables.cpp

```
 1  #include <iostream>
 2
 3  int i = 8;   // i has global scope, outside all blocks
 4  int j = 5;
 5
 6  int main( int argc, char *argv[] ) {
 7     int i = 4;   // das lokale i verdeckt das globale
 8     std::cout << "Block scope der Variable    : " << i << "\n";
 9     std::cout << "Global scope der Variable i : " << ::i << "\n";
10     std::cout << "Global scope der Variable j : " << j << "\n";
11  }
```

```
Block scope der Variable    : 4
Global scope der Variable i : 8
Global scope der Variable j : 5
```

- Namespace-Gültigkeitsbereiche - moduluarisiert Projekte, in die einzelenen Bestandteile individuell gekapselt werden. Variabeln, die innerhalb eines Namespaces "global" angelegt wurden sind nur in diesem sichtbar. Ein Namespace kann in mehreren Blöcken in verschiedene Dateien definiert werden.

namespaces.cpp

```cpp
#include <iostream>

int global = 10;

namespace myFunction{
  int global = 5;

  void doubleGlobal(){
    global += global;
  }
}

int main( int argc, char *argv[] ) {
   std::cout << "Block scope der Variable global         : " << globa
     "\n";
   std::cout << "Variable global im namespace myFunction : " << myFun
     ::global << "\n";
}
```

```
Block scope der Variable global         : 10
Variable global im namespace myFunction : 5
```

- Lokaler Gültigkeitsbereich - Variablen oder Objekte, die innerhalb eines Anweisungsblock oder einem Lambda-Ausdrucks deklariert werden, haben lokale Gültigkeit. Alle Formen von `{ // Anweisungen}` definieren dabei einen eigenen Block (oder *scope*).

localVariables.cpp

```cpp
#include <iostream>

void myFunction(){
  int i = 10;
  std::cout << "Variable i im eingebetteten scope   : " << i << "\n";
}

int main( int argc, char *argv[] ) {
   int i = 0;
   std::cout << "Variable i im aktuellen scope      : " << i << "\n";
   {
    int i = 5;
    std::cout << "Variable i im Scope der Funktion : " << i << "\n";
   }
   std::cout << "Variable i im aktuellen scope      : " << i << "\n";
   myFunction();
}
```

```
Variable i im aktuellen scope      : 0
Variable i im Scope der Funktion : 5
Variable i im aktuellen scope      : 0
Variable i im eingebetteten scope   : 10
```

- Anweisungsbereichs - Anweisungsblöcke erweitern das Konzept des Anweisungsbereiches um die Parameter der Anweisung.

localVariables.cpp

```cpp
#include <iostream>

int main( int argc, char *argv[] ) {
  for (auto i = 0; i<10; i++){
    int j = 5;
    result += i;
  }
  std::cout << "Das Ergebnis für " << i << " Schleifen lautet " << res
    "\n";
}
```

```
main.c: In function ‘int main(int, char**)’:
main.c:6:5: error: ‘result’ was not declared in this scope
    6 |     result += i;
      |     ^~~~~~
main.c:5:9: warning: unused variable ‘j’ [-Wunused-variable]
    5 |     int j = 5;
      |         ^
main.c:8:39: error: ‘i’ was not declared in this scope
    8 |   std::cout << "Das Ergebnis für " << i << " Schleifen lautet "
<< result << "\n";
      |                                       ^
main.c:8:68: error: ‘result’ was not declared in this scope
    8 |   std::cout << "Das Ergebnis für " << i << " Schleifen lautet "
<< result << "\n";
      |
^~~~~~
```

- Klassengültigkeitsbereich - Membervariablen oder Funktionen, die im im Definitionsbereich der Klasse liegen, können nur über die entsprechenden Instanzen oder ggf. als statische Klassenelemente über den Klassennamen adressiert werden. Weiter gesteuert wird dieser Zugriff über öffentliche, private, und geschützt Schlüsselwörter.

classMember.cpp

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

struct Student{
  std::string name;
};

int main()
{
  Student erstsemester {"Gustav"};
  std::cout << "Membervariable 'name' of 'erstsemester' has value '"
    erstsemester.name << "'\n";
}
```

```
Membervariable 'name' of 'erstsemester' has value 'Gustav'
```

Aufgabe der Woche

1. Implementieren Sie die Move Assignment Operation in Beispiel Assignment.cpp
2. Implementieren Sie eine Klasse, die Lese-/Schreiboperationen für Sie realisiert. Warum ist es gerade hier notwendig die Rule-of-Five Idee zu berücksichtigen?