Kapitel 13 - Objektorientierte Programmierung

Peter Ulbrich

AG Systemsoftware

Veranstaltungswebseite

Lehrevaluation

Lehrevaluation WiSe 25/26

oh14.de/eva ⇒ 7ST3Y

Einleitung

- Bisher: prozedurale bzw. imperative Programmierung
  - Geprägt durch hierarchisches Aufrufen von Funktionen
  - Variablen (z.B. `int`, `char`), Datenstrukturen (`struct`) und Funktionen sind getrennt
- Jetzt: Neue Sichtw
  → Objektorientier
- OOP hat einige int
  - Bündelt logisch zusammengehörige Funktionen und Daten in einer Datenstruktur → **Objekt**
  - Sauber definierte Schnittstellen für Interaktion mit Daten
  - Verstecken von internen Funktionen bzw. Variablen → *information hiding*

Keine Sorge! OOP ist „nur“ eine Erweiterung des Bekannten.

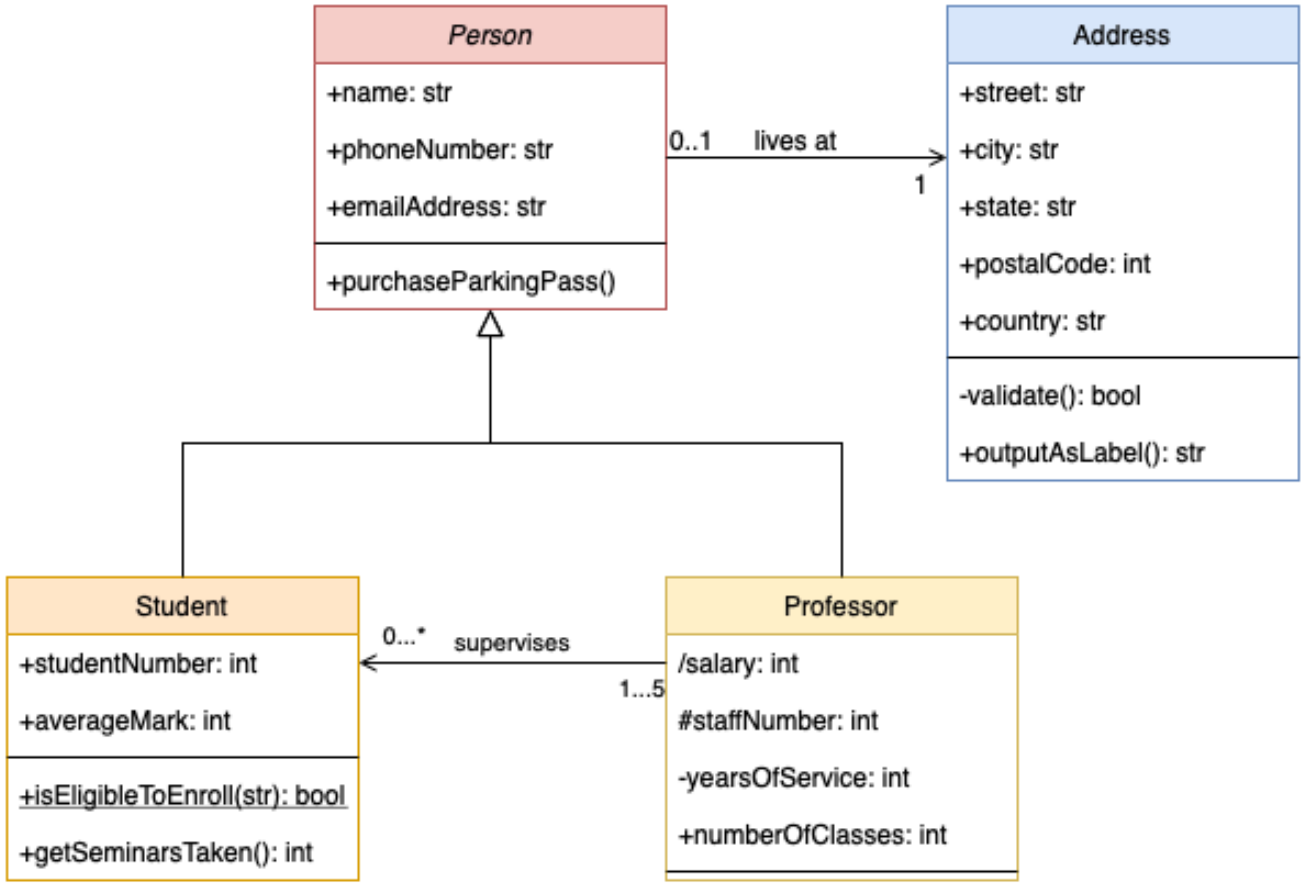
Einleitung

- Modellierung
  - **Anwendungsproblem ⇒ Modellierung ⇒ Reduzieren auf das „Wesentliche“**
  - „Wesentlich“ im Sinne unserer Sicht auf die Dinge bei diesem Problem

    → Es existieren verschiedene Sichten auf dasselbe Problem!
- Objektorientierte Programmierung
  - Formulierung eines Modells in Konzepten und Begriffen der realen Welt
  - Nicht in computertechnischen Konstrukten (Hauptprogramm, Unterprogramm, Funktionen, …)
    - Stattdessen: Klassen (Objekte) mit Attributen und Methoden
    - Interaktion untereinander
- Achtung: Programmiersprache ist hier nur ein Implementierungsdetail

UML-Klassendiagramme

- Pläne zur Beschreibung von abstrakten Objekt-Relationen



© Draw-IO

Objektorientierte Programmierung

Programmierung bis jetzt

```
1  #include <iostream>
2
3  struct Punkt {
4      double x;
5      double y;
6  };
7
8  void print_punkt(Punkt p) {
9      std::cout << "X: " << p.x <<
10         " Y: " << p.y << std::endl;
11 }
```

Objektorientierte Programmierung

```
1  #include <iostream>
2
3  class Punkt {
4      double x;
5      double y;
6  public:
7      void print_punkt() {
8          std::cout << "X: " << x <<
9              " Y: " << y << std::endl;
10     }
11 };
```

Auffällige Unterschiede: Neue Schlüsselwörter `class` und `public`, sonst gleich

Nomenklatur

- Als Teil einer Klasse heißen …
  - Funktionen → Methoden
  - Variablen → Attribute
- Achtung: C++ kennt diese Begriffe offiziell nicht
  - Stattdessen: *Member* einer Klasse
  - Attribut bzw. Methode werden aber trotzdem häufig synonym verwendet
- Bauplan eines Objekts → Klasse
- Mit Werten gefüllte Klasse → Instanz

Auch hier: Instanz und Objekt werden oft synonym verwendet. Aufpassen!

```
1  // Definition des Bauplans
2  class Punkt {
3      double x;
4      double y;
5  };
6
7  // Instanz: Initialisierung
8  Punkt p;
```

Sichtbarkeit

- Abkapselung von Informationen mit Schlüsselwörtern `public` und `private`
- Ziele:
  - Sauberer Zugriff auf Variablen
  - Verstecken von Informationen, die nicht öffentlich zugänglich sein sollen
- *Compiler* hilft bei der Durchsetzung

```
1  struct Punkt {
2      double x;
3      double y;
4  };
5  // Bisher erlaubt
6  Punkt p;
7  p.x = 2.0;
```

(Bei `struct` ist alles öffentlich)

```
1  class Punkt {
2      double x;
3      double y;
4  };
5  // Compiler-Fehler
6  Punkt p;
7  p.x = 2.0;
```

(Bei `class` ist alles privat)

Alles unsichtbar

- Grund für Compiler-Fehler: `private` wird bei Klassen automatisch gesetzt

```
1  class Punkt {
2  // private: implizit gesetzt
3      double x;
4      double y;
5  };
```

```
1  class Punkt {
2  private: // Äquivalent
3      double x;
4      double y;
5  };
```

- Für Strukturen ist implizit `public` → Zugriff erlaubt
  - Einzige Unterschied zwischen Klassen und Strukturen
  - Ansonsten sind sie vollständig identisch

```
1  struct Punkt {
2  // public: implizit gesetzt
3      double x;
4      double y;
5  };
```

```
1  struct Punkt {
2  public: // Äquivalent
3      double x;
4      double y;
5  };
```

Zugriff auf Attribute

- **Externer** Zugriff auf private Attribute nur über Umwege
- Grund: Werte der Attribute sollen stets wohldefiniert sein
- Lösung
  - Verwendung öffentlicher Funktionen für Zugriff
  - `get()` → Abrufen
  - `set()` → Setzen
  - Heißen deswegen auch *Getter* bzw. *Setter*
  - Ermöglicht Überprüfung bzw. Konvertierung der Werte

```
 1  class Punkt {
 2      double x;
 3      double y;
 4  public: // Ab hier alles öffentlich
 5      void set_x(double new_x) {
 6          if (new_x != 0.0) {
 7              x = new_x;
 8          }
 9      }
10      void set_y(double new_y) {
11          if (new_y != 0.0) {
12              y = new_y;
13          }
14      }
15      double get_x() { return x; }
16      double get_y() { return y; }
17  };
```

Sichtbarkeitsmodifikation

- Beliebiges Mischen von `private` und `public` ist prinzipiell möglich
- Aber: Schlecht lesbar und unübersichtlich

  → Möglichst vermeiden!
- Bei großen Projekten oft erst öffentlicher Teil

  → Weniger Sucharbeit bei sehr großen Klassen

```
1  class VeryBigClass {
2  public:
3      /* Viele Funktionen */
4  private:
5      /* Noch mehr private Details */
6  };
```

```
1  // Schlecht lesbar
2  class Punkt {
3  private:
4      double x;
5  public:
6      double get_x() {
7          return x;
8      }
9  private:
10     double y;
11 public:
12     double get_y() {
13         return y;
14     }
15 };
16
```

Information Hiding

- Ziel: Trennung von Klassendefinition und Implementierung
  - Umsetzung über Header- und Source-Datei
  - Definition im Header und Implementierung in der Source-Datei
- Häufige Konvention:
  - Eine Klasse pro Header-/Source-Datei
  - Name von Header-/Source-Datei entspricht dem Klassennamen
- Achtung: Die Methoden benötigen hier zusätzlich das Namespace-Präfix der Klasse!

```cpp
// punkt.h
#pragma once

class Punkt {
    double x;
    double y;
public:
    void print_punkt();
};
```

```cpp
// punkt.cpp
#include "punkt.h"
#include <iostream>

using std::cout, std::endl;
void Punkt::print_punkt() {
    cout << "X: " << p.x
         << " Y: " << p.y
         << endl;
}
```

Initialisierung der Member-Variablen

- Wir kennen jetzt den grundlegenden Aufbau von Klassen in C++
- Aber: Bisher keine Initialisierung möglich
  - Struct-Initialisierung nicht ohne Weiteres erlaubt
    - `Punkt p = {2.0, 3.1}; // Error`
  - Nachträgliches Setzen mithilfe einer `set()`-Methode → sehr umständlich

```
1  class Punkt {
2      double x;
3      double y;
4  public:
5      void set_x(double new_x) {
6          x = new_x;
7      }
8      double get_x() { return x; }
9      /* Restliche Implementierung */
10 };
11
12 Punkt p;
13 p.set_x(2.0); // Umständlich
```

- Lösung: Konstruktor
  - Besondere Methode, die **automatisch** bei der Initialisierung aufgerufen wird
  - Stellt sicher, dass alle Member-Variablen korrekt und konsistent initialisiert sind

Konstruktoren: Initialisierung der Member-Variablen

- Konstruktoren haben eine Reihe von besonderen Eigenschaften:
- Sind an die jeweilige Klasse fest gebunden
- Haben den gleichen Namen wie die Klasse
- Können **nicht** explizit aufgerufen werden
- Automatischer Aufruf bei Instanziierung

```
1  class Punkt {
2      double x;
3      double y;
4  public:
5      Punkt() {} // Default
6      // Parametrisiert
7      Punkt(double x, double y)
8          : x(x), y(y) {}
9  };
```

- Es gibt vier Hauptarten von Konstruktoren
  - Default-Konstruktor → `Punkt() {}`
  - Parametrisierte Konstruktor → `Punkt(double nx, double ny) : x(nx), y(ny) {}`
  - Copy-Konstruktor → `Punkt(const Punkt& p) {}`
  - Move-Konstruktor → `Punkt(Punkt&& p) {}`

Default-Konstruktor

- Wird bei jeder Klassendefinition **automatisch** eingefügt,

  falls kein anderer Konstruktor definiert wird
- Ist ein parameterloser Konstruktor
- Die folgenden beiden Definitionen sind identisch:

```
1  class Punkt {
2      double x;
3      double y;
4  public:
5      Punkt() {} // Explizit genannt
6  };
```

```
1  class Punkt {
2      double x;
3      double y;
4  // public: // Automatisch eingefügt
5  //     Punkt() {}
6  };
```

Eigenschaften des *Default*-Konstruktor

- Initialisiert standardmäßig keine Member-Variablen
- Konsistente Initialisierung auf verschiedene Weisen möglich
- Hier: Durch Überschreiben des Konstruktors bzw. durch Inline-Zuweisungen

```cpp
#include <iostream>
using std::cout;
class Punkt {
    double x = 1.0; // Inline
    double y = 1.0;
public:
    Punkt() {}
    double get_x() { return x; }
    double get_y() { return y; }
};
int main() {
    Punkt p;
    cout << p.get_x() << " " << p.get_y();
}
```

```cpp
#include <iostream>
using std::cout;
class Punkt {
    double x;
    double y;
public:
    Punkt() { x = 2.0; y = 2.0; }
    double get_x() { return x; }
    double get_y() { return y; }
};
int main() {
    Punkt p;
    cout << p.get_x() << " " << p.get_y();
}
```

Member Initializer-Liste

- Initialisierung der Member über eine separate Stelle im Code
- Eigenschaft: wird vor dem Body der Funktion ausgeführt
- Aufbau:

```
<class name>(<function parameters>):<initializer list>{ <function body> }
```

- Wichtig:

  Reihenfolge der Initialisierung =
  Reihenfolge der Deklaration

- Alternative Schreibweise: {} statt ( )

```
1  class Punkt {
2      double y;
3      double x;
4  public:
5      Punkt() : x(2.0), y(1.0) {}
6      // Alternativ: Braced Init.
7      Punkt() : x{2.0}, y{1.0} {}
8
9      /* Restliche Implementierung */
10 };
11
12
```

Parametrisierter Konstruktor

- Initialisierung der Member mit beliebigen Werten
- Prinzip bereits bekannt, jetzt mit Funktionsparametern
- Wichtig: Beliebig viele, **verschiedene** Konstruktoren erlaubt

  → Nennt sich Überladung des Konstruktors

```
1  class Punkt {
2      double x_;
3      double y_;
4      bool g_; // Geheim
5  public:
6      Punkt() :
7          x_(2.0), y_(1.0), g_(false) {}
8
9      Punkt(double x, double y) :
10         x_(x), y_(y), g_(false) {}
11
12     Punkt(double x, double y, bool g) :
13         x_(x), y_(y), g_(g) {}
14
15     /* Restliche Implementierung */
16 };
```

Parametrisierter Konstruktor

cpp Run ▶

```cpp
#include <iostream>

using std::cout, std:: endl;

class Punkt {
    double x_;
    double y_;
    bool g_; // Geheim
public:
    Punkt() : x_(2.0), y_(1.0), g_(false) {}

    Punkt(double x, double y) : x_(x), y_(y), g_(false) {}

    Punkt(double x, double y, bool g) : x_(x), y_(y), g_(g) {}

    void print_punkt() {
        cout << "X: " << x_ << " Y: " << y_ << " G: " << g_ << endl;
    }
};
```

Konstruktoren

- Bisher: Beliebige Anzahl von **verschiedener** Konstruktoren definieren
- Aber: Können wir Konstruktoren auch verbieten?
- Lösung: Schlüsselwort `delete` löscht automatisch erzeugte Konstruktoren

  → `Punkt P() = delete;`
- Analog dazu: Konstruktoren auch explizit als `default` markierbar

  → `Punkt P() = default;`
- `delete` und `default` sollen die Intention des Programmierers verdeutlichen

  → Macht Außenstehendem sofort deutlich, dass die Konstruktoren korrekt sind

Destruktor

- Das Erzeugen und Initialisieren von Objekten ist jetzt bekannt
- Der durch Objekte belegte Speicher muss aber irgendwann auch wieder freigegeben werden
  - Für primitive Datentypen wie `int`, `double` geschieht dies automatisch
  - Aber: Komplexere Datentypen wie Zeiger müssen explizit wieder freigegeben werden
  - Gleiches gilt im Übrigen auch für von der Klasse gehaltene Ressourcen
    - Beispiele: Geöffnete Datei, aktive Datenbankverbindung
    - … oder was auch immer gerade benötigt wird
- Hier wird der Destruktor relevant!

Destruktor

- Destruktor ist das Komplement zum Konstruktor
  - Benennung: Vorangestellte Tilde (~) vor dem Klassennamen → `~Punkt() {}`
  - Wird ebenfalls automatisch erzeugt und muss ggf. überschrieben werden
  - Allerdings keine Überladung möglich (immer nur *ein* Destruktor pro Klasse)
- Wird bei Verlassen des Scopes aufgerufen, in dem das Objekt instanziiert wurde

```
1  class DataHandle {
2      void * data;
3      int size_;
4  public:
5      DataHandle() = delete;
6      DataHandle(int size): size_(size) {
7          data = malloc(size * sizeof(void*));
8      }
9      ~DataHandle() {
10         free(data);
11     }
12 };
13
14
15 int main() {
16     DataHandle dh{3}; // Instanziierung
17     /* Arbeite mit dh */
18 }   /* Aufruf Destruktor dh */
```

Destruktor

cpp Run ▶

```cpp
#include <iostream>

class DataHandle {
    void * data;
    int size_;
public:
    DataHandle() = delete;
    DataHandle(int size): size_(size) {
        data = malloc(size * sizeof(void*));
        std::cout << "Constructor called" << std::endl;
    }
    ~DataHandle() {
        free(data);
        std::cout << "Destructor called" << std::endl;
    }
};

int main() {
```

Beispiel - *Copy*-Konstruktor

- Angenommen, folgender Code wird ausgeführt. Was wird passieren?

cpp Run ▶

```cpp
#include <iostream>

class DataHandle {
    void * data;
    int size_;
public:
    DataHandle() = delete;
    DataHandle(int size): size_(size) {
        data = malloc(size * sizeof(void*));
    }
    ~DataHandle() {
        std::cout << "Destructor called" << std::endl;
        free(data);
    }
};
```

Copy-Konstruktor

- `dh2` ist eine bitweise Kopie von `dh1` (*Shallow Copy*)
- Beinhaltet auch den Zeiger `data`
- `dh1.data` und `d`
  gleichen Speicher
- Doppelter Aufruf von Destruktor
- Ergebnis: Zweifaches `free()` führt zu Absturz des Programms 🤯

Deshalb gibt es den Copy-Konstruktor

```
1  class DataHandle {
2      void * data;
3      int size_;
4  public:
5      ~DataHandle() {
6          std::cout
                << "Destructor called"
                << std::endl;
            e(data);
```

```
1  int main() {
2      DataHandle dh1{4};
3      {
4          DataHandle dh2 = dh1;
5      } // Destruktor dh2
6  }     // Destruktor dh1. BOOM
7
```

Copy-Konstruktor

- Copy-Konstruktor hat einen Parameter → Referenz auf Objekt
  - Beispiel: `DataHandle(const DataHandle& d) {}`
  - `const` verbietet Modifikation des Originals
  - Referenz → **kein** *Copy-by-Value* des Originals (aufwendig)
- Korrektes Initialisieren von Member (*Deep Copy*)

```
1  class DataHandle {
2      void * data;
3      int size_;
4  public:
5      DataHandle(const DataHandle& d) : size_(d.size_) {
6          data = malloc(d.size_);
7          std::memcpy(data, d.data, d.size_);
8      }
9  };
```

Copy-Konstruktor

cpp Run ▶

```cpp
#include <iostream>
#include <cstring>

class DataHandle {
    void * data;
    int size_;
public:
    DataHandle() = delete;
    DataHandle(int size): size_(size) {
        data = malloc(size * sizeof(void*));
        std::cout << "Constructor called" << std::endl;
    }
    DataHandle(const DataHandle & d) : size_(d.size_) {
        data = malloc(d.size_);
        std::memcpy(data, d.data, d.size_);
        std::cout << "Copy Constructor called" << std::endl;
    }
    ~DataHandle() {
```

Move-Konstruktor

- Ist ein etwas fortgeschrittener Aspekt von C++
- Hier: Konzept verstehen, statt jedes technische Detail zu ergründen
- Zunächst ein abstraktes Beispiel: Staffellauf
- Dabei übergibt ein
  Objekt) einem neu
  Ressource (→ Stab,

An dieser Stelle kommt der Move-Konstruktor ins Spiel!

© Microsoft Imagine - mit KI generiert

- Mit unserem bisherigen Wissen wäre das nicht gut umsetzbar 😕

Move-Konstruktor

- Ziel: effizientes Verschieben von Ressourcen von einer Instanz zu einer anderen
  - Beispiel Zeiger: Verschieben des Zeigerwerts zum neuen Objekt
  - Wichtig: Originales Objekt sollte danach keinen Zugriff mehr auf die Ressource haben

    → Bester Weg: Alter Zeiger wird `nullptr`

- Spezielle Signatur:

  `<Type>(<Type>&& var_name);`

  - Beispiel: `DataHandle(DataHandle&& dh);`
  - Formal ist `&& dh` eine *Referenz* auf einen *Rvalue*
  - Der Vollständigkeit halber: Rvalues sind temporär existierende Werte/Ausdrücke (z.B. `3 + 3;`)

```
1  3 + 3; // Temp. Rvalue
```

Move-Konstruktor

- C++ bietet für die Konvertierung eine Hilfsfunktion: `std::move`

```cpp
#include <utility> // für std::move
DataHandle dh1("foo");
// Type-Casting auf DataHandle &&
// gefolgt von Konstruktor-Aufruf
DataHandle dh2(std::move(dh1));
```

- Irreführender Name, `std::move` bewegt eigentlich nichts
- Stattdessen nur Type-Casting zu Referenz auf Rvalue

  → `DataHandle` wird zu `DataHandle &&`
- Gecasteter Wert wird anschließend als Parameter im Konstruktor-Aufruf verwendet

Move-Konstruktor

- Move-Konstruktor hat grundsätzlich Ähnlichkeit zum Copy-Konstruktor
- Wichtiger Unterschied: Ressourcen beim alten Objekt unzugänglich machen
- Grund: Alte Objekte existieren nach Move weiterhin

  (und könnten prinzipiell auch Ressourcen manipulieren)

```
1  class DataHandle {
2    void * data;
3    int size_;
4  public:
5    DataHandle(DataHandle && dh) : data(dh.data), size_(dh.size_) {
6        dh.data = nullptr; // Zugangssperre
7    }
8    /* Restliche Implementierung */
9  };
```

Aufrufreihenfolge Destruktor

- Wichtig: Aufrufreihenfolge von Konstruktoren und Destruktoren
- Zerstörung immer in **umgekehrter** Reihenfolge zur Deklarierung

cpp Run

```cpp
#include <iostream>
using std::cout, std::endl;

class A {
public:
    A() { cout << "A: Constructor" << endl; }
    ~A() { cout << "A: Destructor" << endl; }
};

class B {
public:
    B() { cout << "B: Constructor" << endl; }
    ~B() { cout << "B: Destructor" << endl; }
};
```

Resource Acquisition Is Initialization (RAII)

- Ist eine interessante Programmiertechnik: *Resource Acquisition Is Initialization (RAII)*
  - Möglich durch automatische Destruktoraufrufe bei Verlassen des Scopes
  - Ressourcen in diesem Kontext müssen vorher akquiriert werden
  - Beispiel: Öffnen einer Datei bzw. Datenbankverbindung
- Grundgedanke: Bei **Konstruktor**-Aufruf wird Ressource akquiriert, bei **Destruktor**-Aufruf automatisch wieder freigegeben
- **Sehr praktisch**: Speicherleck/Ressourcenleck wird dadurch unmöglich

Beispiel - *Resource Acquisition Is Initialization*

- Situation: Automatisches Öffnen und Schließen einer Datei bei Initialisierung und Destruktor-Aufruf

cpp Run ▶

```cpp
#include <iostream>
#include <fstream>

using std::cout, std::e
using std::ios;

class FileHandler {
    std::fstream file; // File Handle
public:
    FileHandler(const std::string& filename) {
        file.open(filename, ios::in | ios::app);
        if (file.is_open()) {
            cout << "File opened successfully" << endl;
        }
    }
```

RAII ist eine extrem nützliche Programmiertechnik. Nutzt sie!

Zwischenstand

- Zeit für einen kurzen Zwischenstand:
  - Grundlegende Eigenschaften von Objekten wurden vorgestellt
  - Die grundlegende Reihenfolge der Initialisierung/Destruktion ebenso
  - Konstruktoren/Destruktoren sind jetzt bekannt
- Es fehlt: *Smart Poi* …ern)
  - Teil einer Reihe vor…

Zunächst einmal kommen aber noch einige allgemeine Konzepte!

Operator-Überladung

- Zur Erinnerung: C++ erlaubt das Überladen von Funktionen (und Konstruktoren)
- Außerdem: Operatoren könnnen auch überladen werden (Klassen/Strukturen)
  - Bereits bekannt: `<<`-Operator bei `std::cout`
  - `std::cout << "Echo";`
- Überladung mithilfe des Schlüsselworts `operator`
  - `operator+=(int i); // Überlade += für int-Parameter`
- Ist für die meisten Operatoren möglich, mit ein paar Ausnahmen:
  - `.` (bzw. `.*`) → Member-Zugriff (bzw. Member-Zugriff über Zeiger)
  - `::` → Scope Resolution Operator
  - `?:` → Ternärer Operator

Beispiel - Operator-Überladung

- Beispiel: Wrapper-Klasse für Integer

cpp Run ▶

```cpp
#include <iostream>

class Integer {
    int val;
public:
    Integer() = delete;
    Integer(int i) : va

    int operator+(int s
    void operator+=(int

    int value() { return val; }
};

int main() {
```

Operator-Überladung ist sehr nützlich, wenn es an den richtigen Stellen eingesetzt wird!

Einschub: Rule of Three/Five

- Betrachtung einer wichtigen Regel in C++: *Rule of Three/Five*
- Rule of Three
  - **Wenn** es notwendig ist, einen eigenen Destruktor, Copy-Konstruktor, oder Copy-Assignment-Operator (`operator=(const &)`) zu erstellen, sollen* **alle drei** *erstellt werden*
- Rule of Five
  - Erweiterung der *Rule of Three* um Move-Konstruktor und Move-Assignment-Operator (`operator=(&&)`)
- Grund ist simpel: Ressourcen sollen **immer** konsistent initialisiert werden bzw. verfügbar sein

Schlüsselwort `friend`

- Externer Zugriff auf Member-Variablen nur via Methoden
- Ausgangslage: Funktionen oder fremde Klasse soll direkten Zugriff haben
- Problem: C++ verbietet das! 😱

```
1  class Punkt {
2      double x;
3      double y;
4  public:
5      void print_punkt() { // Ok
6          cout << "X=" << x <<
7              " Y=" << y << endl;
8      }
9  };
10 // Überladung von Operator << für cout
11 // Compiler-Error: Zugriff auf private Member
12 ostream& operator<<(ostream& os, Punkt& p) {
13     return os << "X=" << p.x
14         << "Y=" << p.y << endl;
15 }
```

- Mit den aktuellen Möglichkeiten geht das nicht!

  → Lösung: Neues Schlüsselwort `friend` 😀

Schlüsselwort `friend`

- `friend` gestattet selektiven Zugriff für externe Klassen und Funktionen

  → `friend ostream& operator<<(ostream& os, Punkt& p);`

- Randbemerkung: Aufruf von externen Funktionen ohne Namespace-Präfix

cpp Run ▶

```cpp
#include <iostream>
using std::cout, std::endl;
using std::ostream;

class Punkt {
    double x;
    double y;
public:
    Punkt(double x, double y) : x(x), y(y) {}

    friend void print_punkt(Punkt& p);
    friend ostream& operator<<(ostream& os, Punkt& p);
};
```

Schlüsselwort `this`

- Nur innerhalb einer Klasse verfügbar
- Zeiger auf die aktuelle Instanz der Klasse (wird automatisch erzeugt)
- Verwendung wie alle anderen Zeiger
  - `this->print();`
  - `*this; // Objekt selbst`
- Vorteile
  - Vermeidung von Namenskonflikten
  - Manche Operatorüberladungen erfordern Verweis auf sich selbst
  - Explizite Nennung verbessert Lesbarkeit des Quellcodes

```
1  class Integer {
2      int val;
3  public:
4      Integer(int i) : val(i) {}
5
6      int value() {
7          return val; // Implizit
8      }
9      int value_explicit() {
10         return this->val;
11     }
12 };
```

Schlüsselwort `static` in Klassen

- Member dürfen als `static` deklariert werden
- Gehören keinem Objekt an, sondern der Klasse selbst

cpp Run ▶

```cpp
#include <iostream>

using std::cout, std::endl;

class A {
public:
    static int _count;
    A() { _count++; }
    ~A() { _count--; }
    static int count() { return _count; }
};
int A::_count = 0; // Initialisierung *außerhalb* der Klasse in *.cpp-Datei

int main() {
    A a1, a2;
```

Dynamische Allokation in C++ (`new` / `delete`)

- C++ erweitert die aus C bekannten Funktionen `malloc` und `free`
- Operatoren ersetzen die Funktionen
  - `malloc` → `new` (dyn. Allokation)
  - `free` → `delete` (dyn. Deallokation)
- Im globalen Namespace enthalten

  → kein Einbinden von Headern notwendig
- Ansonsten gleiche Eigenschaften wie `malloc` und `free`

cpp Run ▶

```cpp
#include <iostream>
using std::cout, std::endl;

class Foo {
public:
    int bar;
    Foo(int val) : bar(val) {}
};

int main() {
    Foo * foo = new Foo(1);
    int * i = new int(5);
    cout << "Foo: " << foo->bar << endl;
    cout << "Int: " << *i << endl;
    delete i;
    delete foo;
}
```

Dynamische Allokation in C++ (`new` / `delete`)

- `new` und `delete` können auch mit Arrays umgehen
- Das Löschen von Arrays erfolgt mit Operator `delete[]`
- Achtung Stolperfalle: Bei Arrays mit mehreren Dimensionen muss über die Dimensionen iteriert werden

```
1  Foo * foo = new Foo[ROWS][COLS];
2  for (int i = 0; i < ROWS; ++i) {
3      delete[] foo[i]; // Spalte i löschen
4  }
5  delete[] foo; // Alle Zeilen
6  f = nullptr; // Zeigerwert löschen
```

cpp Run ▶

```
#include <iostream>
using std::cout, std::endl;

class Foo {
public:
    int bar;
    Foo() : bar(0) {}
    Foo(int val) : bar(val) {}
};

int main() {
    // 10 Foo-Objekte, default-initialisiert
    Foo * foo = new Foo[10];
    int * i = new int[5] { 1, 2, 4, 8, 16 };

    foo[2].bar = 4;
    cout << "Foo: " << foo[2].bar << endl;
    cout << "Int: " << i[4] << endl;
    delete[] i;
    delete[] foo;
```

Dynamische Allokation in C++ (`new` / `delete`)

- `new` und `delete` haben die gleichen Schwächen wie `free` und `malloc` 😕
- Syntax ist jetzt etwas angenehmer
- Aber: Programmierer:in kann immer noch Aufrufe vergessen
- Ist es also am Ende in C++ genauso schlimm wie in C?

  → **Nein!**
- C++11 hat die sogenannten *Smart Pointer* eingeführt

Smart Pointer

- Seit C++11 Bestanteil des Standards
- Objekte kapseln Zeigervariable
- Verwenden dazu im Kern das RAII-Idiom
- Zwei wichtige Typen
  - `unique_ptr`
  - `shared_ptr`
- Klassische Zeiger werden oft als *Raw Pointer* bezeichnet

Unique Pointer

- Lebensdauer von `unique_ptr` vollständig an den aktuellen Scope gebunden (→ RAII)
- Bei Verlassen des Scopes wird ein `unique_ptr` zerstört

```
1  // Vereinfachte Darstellung eines
2  // Unique Pointers für int *
3  class UniquePtr {
4  private:
5      int * ptr_;
6  public:
7      UniquePtr(int * ptr) : ptr_(ptr) {}
8      ~UniquePtr() { delete ptr_; }
9  }
```

Unique Pointer

- Benutzung von Unique Pointer ist sehr einfach

  → Deklaration: `std::unique_ptr<int> i_ptr; // Zeiger auf int *`

- Erstellung eines neuen Zeigers mithilfe `std::make_unique`

```cpp
#include <memory> // std::unique_ptr
using std::unique_ptr;
using std::make_unique;

// Erstellt neuen int *
// Wert des derefenzierten int* ist 5
unique_ptr<int> i_ptr = make_unique<int>(5);

int i = *i_ptr + 3; // 8
```

- Bestehenden Zeiger übernehmen → Konstruktur von `unique_ptr`

```cpp
#include <memory> // std::unique_ptr
using std::unique_ptr;

class Foo{ /* */ };

int main() {
    Foo * f = new Foo();
    unique_ptr<Foo> f_ptr(f);
} // f_ptr ruft delete
```

Unique Pointer

- Dank Operator-Überladung Zugriff auf gekapselten Zeiger
- Dereferenzierung mit `*i_ptr`
- Member-Zugriff mit `->`

  (bei Klassen)

Dereferenzierung

```
1  unique_ptr<int> i_ptr = make_unique<int>(5);
2
3  int i = *i_ptr + 3; // 8
```

Member-Zugriff

```
1  class Foo {
2  public:
3      void bar() {}
4  }
5
6  unique_ptr<Foo> f_ptr = make_unique<Foo>();
7
8  f_ptr->bar();
```

Beispiel - Unique Pointer

cpp Run ▶

```cpp
#include <memory>
#include <iostream>

using std::unique_ptr;
using std::cout, std::endl;

class Foo {
public:
    Foo() { cout << "Fo
    ~Foo() { cout << "F

    void print() { cout
};

int main() {
    cout << "main(): Enter" << endl;
    Foo * f = new Foo();
    {
```

Unique Pointer sind deutlich besser als händisches Verwalten! Nutzt sie, wenn ihr könnt.

Shared Pointer

- Analog zu `unique_ptr` gibt es noch `shared_ptr`
  - Hauptzweck: Einsatz in nebenläufiger Programmierung (-
    CPU-Kernen)
  - Unterschied zu un
    neuem Kopieren w
    inkrementiert (Zähl

```
1  class SharedPtr { // Vereinfachte Darstellung
2  private:
3      int * ptr;
4      int counter = 1;
5  public:
                                    &  cpy) { ++counter; }
                                                    0)
```

`shared_ptr` ist hauptsächlich für nebenläufige Programmierung gedacht!

- Bei Aufrufen des Destruktors wird dieser Zähler dekrementiert
  - Erst wenn Zähler Wert `0` annimmt, wird der gehaltene Zeiger zerstört
- Achtung: Zum Teilen von Membern in verschiedenen Klassen gedacht
  - Aber: Widerspricht dem Grundgedanken der sauberen Kapselung durch Klassen

Einschub: Garbage Collection

- Idee hinter *Shared Pointer*: Zählen von Referenzen
- Gibt es auch in anderen Programmiersparchen: z.B. Java, Python, C#, Javascript
- Keine automatischen Destruktor-Aufrufe vorhanden wie in C++
- Stattdessen: Konzept namens *Garbage Collection*
  - Dazu wird für jedes Objekt zusätzlich ein Referenzenzähler angelegt
- Ausführungsumgebung (*Runtime*) pausiert die Ausführung regelmäßig und zählt
  - Falls ein Zähler den Wert `0` hat, wird das zugehörige Objekt entfernt
  - Das Programm steht in der Zwischenzeit!
    - Natürlich deutlich langsamer als in C++ (Je nach Sprache 1-2 **Größenordnungen**)
    - Dafür muss der Programmierer sich aber keine Gedanken über Speichermanagement machen

Zusammenfassung

- Aufbau von Klassen
  - Sichtbarkeitsmodifikatoren
  - Konstruktoren (*Copy*, *Move*, ...)
  - Destruktoren
- Konzept der Operator-Überladung
- *Resource Acquisition Is Initialization*
- Dynamische Speicherallokation(`new` / `delete`)
- *Smart Pointer*

Kapitel 13 - Objektorientierte Programmierung