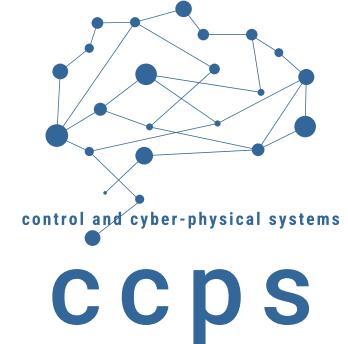


Programmierpraktikum C und C++



```
#include <iostream>

int main() {
    std::cout
        << "Welcome!"
        << std::endl;
}
```



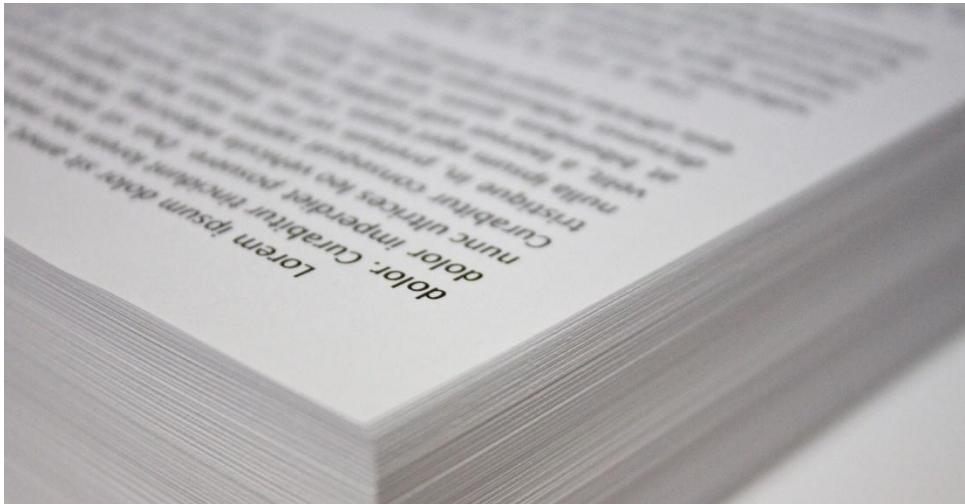
Dr.-Ing. Eric Lenz
elenz@iat.tu-darmstadt.de

Fachgebiet Control and Cyber-Physical Systems (CCPS)

Prof. Dr.-Ing. Rolf Findeisen
Dept. of Electrical Engineering and Information Technology
<https://www.ccps.tu-darmstadt.de/>

Programmierpraktikum C und C++

Organisatorisches



Dr.-Ing. Eric Lenz
elenz@iat.tu-darmstadt.de

Fachgebiet Control and Cyber-Physical Systems (CCPS)
Prof. Dr.-Ing. Rolf Findeisen
Dept. of Electrical Engineering and Information Technology
<https://www.ccps.tu-darmstadt.de/>

Zielsetzung



TECHNISCHE
UNIVERSITÄT
DARMSTADT

In diesem Praktikum wollen wir einige **Besonderheiten der Sprachen C++ und C (für Microcontroller)** kennenlernen.

Idee des Praktikums

- Vortragsteil vermittelt Konzepte
- Übung vermittelt praktische Kenntnisse

Basisvoraussetzungen

- Allgemeine Programmiererfahrung
- Kenntnisse in Java

Zusammenhang zwischen C, C++ und Java



C "1.0"
(1972)



C++ "1.0"
(1980~85)

ANSI C/C89 (1989)
"Programming
Language C"

C95 (1995)

C99 (1999)

C11 (2011)



Java 1.0 (1996)

C++98 (1998)

C++03 (2003)

C++11 (2011)

C++14 (2014)

Java 1.5 (2004)

Java SE 6 (2006)

Java SE 7 (2011)

Java SE 8 (2014)

[G] Grundlagen

- Projektstruktur, Kompiliervorgang, allgemeine Konzepte

[S] Speicherverwaltung

- Speicherbereiche in C++, Vergleich zu Java
- Typische Fallstricke – davon gibt es reichlich!

[O] Objektorientierung

- Besonderheiten von C++

[F] Fortgeschrittene Themen

- Templates: vergleichbar mit Generics in Java
- Funktionszeiger: in C von Anfang an, in Java erst seit 1.8!

[C] Einführung in C

- Besonderheiten von C im Vergleich zu C++
- Arbeiten auf und mit einem eingebetteten System in der Praxis
 - Hands-on Programmieren eines Microcontrollers



Veranstaltung im hybriden Modus

- Keine verpflichtende Präsenzveranstaltung
- Theoretischer (C++) Teil findet Online statt
 - Vorlesungsaufzeichnungen mit Folien
 - Online-Abgabe von (Bonus-)Aufgaben
 - Sprechstunden werden in Präsenz und Online angeboten
 - **Hinweis:** Sie sind für ihren Lernerfolg selbst verantwortlich!
- Hardware-Teil findet in Präsenz statt

Offizieller Moodle-Kurs als zentrale Anlaufstelle

<https://moodle.tu-darmstadt.de/course/view.php?id=43866>

- Ankündigungen und allgemeine Organisation
- Vorlesungs- und Übungsmaterialien (nach Themen aufgeteilt)
- Gesamter Foliensatz auf Moodle **ab jetzt** verfügbar
- Aufzeichnungen schrittweise ab dem 22.04.2025 verfügbar
- Moodle-Quizzes zu einzelnen Themengebieten zur Selbstkontrolle

Ablauf der Veranstaltung

Keine Vorlesungseinheiten oder Übungen als Live-Stream

- Stattdessen: Vorlesungsaufzeichnungen

Vorlesung und Übungen aufgeteilt in Themenblöcke

- Vorlesungen und Übungen zu den Themenblöcken werden nacheinander veröffentlicht
- [G] 22.04, [S] 05.05, [O] 23.05., [F] 16.06, [C] 07.07, [C embedded] 21.07
- [Z] gemischte Zusatzaufgaben (= Bonusaufgaben) ab dem 21.07

Sprechstunden

- Geplant sind bis zu drei Sprechstundetermine je Themenblock
 - Eine „Diskussionssprechstunde“ bei denen der Inhalt des Blocks an verschiedenen Code-Beispiele gezeigt und diskutiert wird
 - Zwei Sprechstunden die nur für Ihre Fragen zur Vorlesung, zu Übungsaufgaben, Bonusaufgaben (nur Verständnis) etc. gedacht sind. Diese können Sie auch gerne zum Programmieren vor Ort nutzen, um bei Bedarf Fragen stellen zu können
- Termine und Ort bzw. Zoom-Link werden über Moodle bekanntgegeben.

Bonusaufgaben

- Bonusaufgaben werden mit den dazugehörigen Themenblöcken veröffentlicht
- [G] 15.04, [S] 29.04, [O] 20.05, [F] 10.06, [C] 01.07, [C embedded] 22.07
- Der letzte Teil [C] ist in zwei Aufgaben aufgeteilt
- Letzte (Bonus-) Aufgabe: Präsenzteil der Veranstaltung

Präsenzteil ab dem 21.07.2025 mit der ersten **Präsenzübung** ab dem 28.07.2025

Allgemeine Hinweise zur Übung



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Übungen sind (überwiegend) Programmieraufgaben

- Empfohlen: Verwendung einer Entwicklungsumgebung (IDE)
➤ Wir verwenden\empfehlen die CLion IDE

Installationsanleitungen auf Moodle

- CLion IDE (Windows / Linux / MacOS)

Bonusübungsabgaben

- Je Gruppe nur eine Abgabe
- Jede Programmieraufgabe als eigenes CMake Projekt
- Abgabe der Projekte (Aufgaben) als Zip-Archiv

Hinweise zur regulären Übung

Aufgabenblatt – Je Thema ein Dokument mit allen Aufgaben

- Thema des Aufgabenblatts markiert durch:
 - [G] C++-Grundlagen
 - [S] Speichermanagement in C++
 - [O] Objektorientierung
 - [F] Fortgeschrittene Themen
 - [C] (Embedded) C-Programmierung
 - [Z] Zusatzaufgaben: Aufzugsimulator aus der Vorlesung selber implementieren und weitere optionale Aufgabe.
➤ **[Z]** Nicht klausurrelevant aber eine gute Vorbereitung für die Klausur!

Musterlösungen

- Kommentierter C/C++ Quell-Code
- Jeweils verfügbar als Zip-Archiv in Moodle

Hinweis: Die regulären Übungen tragen nicht zum Bonus bei und müssen deshalb auch nicht abgegeben werden.

➤ **Inhalt ist dennoch klausurrelevant!**

Bonusaufgaben



Zusätzliche Bonusaufgaben

- Zu jedem Themengebiet gibt es eine Bonusaufgabe (Ausnahme [C])
- **Veröffentlichungstermine:** [G] 22.04, [S] 05.05, [O] 26.05, [F] 16.06, [C] 07.07, [C embedded] 21.07
- Bonusaufgaben ermöglichen einen Bonus von bis zu 1,0 auf das Klausurergebnis
 - Helfen nicht beim Bestehen!
- $$\text{Bonus} = 1.0 \times \frac{\# \text{ Bestandene Bonusaufgaben}}{\# \text{ Bonusaufgaben}}$$

▪ Abgabe und Korrektur

- Aufgaben werden in Gruppen bearbeitet und abgegeben (→ **Gruppe in Moodle auswählen!**)
 - Sie können sich in 1er-, 2er- und 3er-Gruppen anmelden.
- **Abgabefristen beachten!**
- Bonusaufgabe gilt als bestanden wenn mindesten 75% der Teilpunkte erreicht sind
 - Falls die Bestehensgrenze keine glatte Zahl ergibt, runden wir ab.
- Alle Teilnehmer einer Gruppe bekommen die gleiche Punktzahl
- Abgabe der Aufgaben als gezipptes CMake Projekt
- Antworten auf Rechenaufgaben oder Wissensfragen als PDF dem Archiv beilegen
- **Wichtig: Ihr Code muss kompilieren!**
Bei Problemen damit: Wir helfen Ihnen gerne!



Organisation

- **Praktikumstermine:** 28.07.2025 – 12.08.2025 (jeweils von 09:00 - 16:00 Uhr in S3|21-1)
- Aufgeteilt in 6 Blöcke mit jeweils 2 Tagen und bis zu 50 Studierenden
- **Ziel:** Bearbeitung der letzten (Bonus-) Übungsaufgabe mit Hilfe des Microcontroller-Boards

Vorbereitung

- **Auswahl einer Bonusübungsgruppe** (sonst keine Abgabe der Bonusübung möglich)
- Auswahl eines Termins (Blocks) für die Durchführung des Präsenzteils (-> Moodle)
- Ggf. Erstellen eines Kontos für den Rechner-Pool: Hierzu erfolgen noch Informationen

Durchführung

- An zwei zusammenhängenden Tagen Programmieren eines Microcontrollers
- Fokus liegt auf dem Bearbeiten der Bonusübung
- Falls Zeit übrig ist: Bearbeiten der regulären Übung
- Testat durch Tutor am jeweiligen letzten Präsenztag
- Code-Abgabe spätestens am 15.08.2025 auf Moodle.

Klausur



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Termin

Datum: Freitag, den 26. September 2025

Uhrzeit: 09:00 - 11:30

Räume: werden in der Woche vor der Klausur bekanntgegeben

Ablauf

- Bearbeitungszeit: 90 Minuten
- Bekanntgabe von weiteren Details zur Klausurdurchführung rechtzeitig via Moodle

Zur Teilnahme erforderlich

1. Amtlicher Lichtbildausweis
2. Klausuranmeldung (TUCaN!)

Klausur



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Inhalt

- Alle Inhalte der **Vorlesungsfolien**, die nicht als **[Exkurs]** gekennzeichnet sind.
- Alle Inhalte der regulären **Übungen**, die nicht als "optional" gekennzeichnet sind.
- Alle Inhalte der **Bonusübungen**.

Vorbereitung

1. Konzepte der Vorlesung verstehen
2. Übungen und Bonusaufgaben selbstständig lösen
3. Altklausuren (auf Moodle) selbstständig lösen

Hilfsmittel

1. Dokumentenechte Stifte (z.B. Kugelschreiber)
2. Wörterbuch (z.B. Deutsch <-> Englisch)
3. Nicht erlaubt:
 - ❖ Bleistifte
 - ❖ Rote oder grüne Stifte
 - ❖ Tintenkiller, TippEx etc.
 - ❖ Taschenrechner



Fragen?



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Grundsätzlich: Am besten stellen Sie ihre Fragen immer in Moodle, sodass andere auch von der Antwort auf ihre Frage profitieren können.

Organisatorisch

- Allgemeines Moodle-Forum

Vorlesungsinhalte \ Übungen

- Moodle-Forum zum entsprechenden Kapitel

In dringenden Fällen: Per Mail oder Moodle-PN an den Veranstalter oder die Tutoren.

Ansprechpartner

Dr.-Ing. Eric Lenz

(Veranstalter – Vorlesung, Übung)





TECHNISCHE
UNIVERSITÄT
DARMSTADT

ERGÄNZENDE RESSOURCEN

Online C++-Referenzen



- Ausführliche Dokumentation von Standardbibliotheken
- Erläuterung von **Best Practices** und **Programmierkonzepten** für C++

<http://www.cplusplus.com/>

The screenshot shows the cplusplus.com website. The navigation bar includes a search field, a 'Go' button, and tabs for 'Reference' and '<iostream>'. The left sidebar has sections for 'Information', 'Tutorials', 'Reference', 'Articles', and 'Forum'. Under 'Reference', there's a tree view with 'C library', 'Containers', 'Input/Output' (which is expanded to show '<fstream>', '<iomanip>', '<ios>', '<iostfwd>', '<iostream>', '<iostream>', '<ostream>', '<sstream>', '<streambuf>'), 'Multi-threading', and 'Other'. The main content area is titled '<iostream>' and describes it as the 'Standard Input / Output Streams Library'. It notes that the header defines standard input/output streams and may include '<iomanip>' and '<iostfwd>'. A note says that the `iostream` class is mainly declared in the `<iostream>` header. There are also sections for 'Objects' and 'Related pages'.

<http://en.cppreference.com/w/>

The screenshot shows the en.cppreference.com website. The top navigation bar includes a search field, a 'Go' button, and tabs for 'C++ reference' and '<iostream>'. The main content area is titled 'C++ reference' and includes links for 'C++98', 'C++03', 'C++11', and 'C++14'. The page is organized into several sections: 'Language' (ASCII chart, Compiler support), 'Containers library' (array, vector, deque, list, forward_list, set, multiset, map, multimap, unordered_set, unordered_multiset, unordered_map, unordered_multimap, stack, queue, priority_queue), 'Algorithms library', 'Iterators library', 'Numerics library' (Common mathematical functions, Complex numbers, Pseudo-random number generation), 'Strings library' (basic_string, Null-terminated byte strings, Null-terminated multibyte strings, Null-terminated wide strings), 'Headers', 'Concepts', 'Utilities library' (Type support, Dynamic memory management, Error handling, Program utilities, Date and time, bitset, Function objects, pair, tuple, integer_sequence), and 'Headers' (including Preprocessor, Keywords, Operator precedence, Escape sequences, Fundamental types).

C++-FAQ (<https://isocpp.org/wiki/faq/>)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

C++ FAQ Sections

Overview Topics

- Big Picture Issues
- Newbie Questions & Answers
- Learning OO/C++
- Coding Standards
- User Groups Worldwide (map 

Starting From Another Language

- Learning C++ if you already know Objective-C
- Learning C++ if you already know C# or Java
- Learning C++ if you already know C
- How to mix C and C++

Learning C++ if you already
know [...] Java

General Topics

- Built-in / Intrinsic / Primitive Data Types
- Input/output via `<iostream>` and `<cstdio>`
- Const Correctness
- References

Const Correctness,
Referenzen,...

Man kann C++-Code auch online testen:

- <https://www.onlinegdb.com/>
- <http://cpp.sh>
- <https://godbolt.org/>



The screenshot shows the OnlineGDB beta interface. The main window has a dark theme with a light gray header bar. The header bar includes a logo, file, run, debug, stop, share, save, beautify, and download buttons, and a language selector set to C++. Below the header is a toolbar with a magnifying glass and gear icons. The left sidebar contains links for OnlineGDB beta, IDE, My Projects, Login, About, FAQ, Blog, Terms of Use, Contact Us, GDB Tutorial, and Online Java/Python Debugger, along with a copyright notice for 2017. The central area is a code editor with the following C++ code:

```
1 // ****
2
3 // Welcome to GDB Online.
4 // GDB online is an online compiler and debugger tool for C, C++, Python.
5 // Code, Compile, Run and Debug online from anywhere in world.
6
7 ****
8 #include <stdio.h>
9
10 int
11 main ()
12 {
13     printf ("Hello World");
14
15     return 0;
16 }
```

Below the code editor is an "input" section with a "Command line arguments:" input field and "Standard Input" options: "Interactive Console" (selected) and "Text".

Literaturvorschläge



- Bruce Eckel: Thinking in C++ (frei verfügbar)
<https://www.micc.unifi.it/bertini/download/programmazione/TICPP-2nd-ed-Vol-two-printed.pdf>)
- Mike Banahan: The C Book (frei verfügbar:
http://publications.gbdirect.co.uk/c_book/)
- Expert C Programming: Deep C Secrets, Peter van der Linden, Prentice Hall 1997
- Scott Meyers: Effective C++ & More Effective C++
- Helmut Schellong: Moderne C Programmierung [Springer]
- Ralf Schneeweiß: Moderne C++ Programmierung [Springer]
- Jürgen Wolf: Grundkurs C [Galileo] & Grundkurs C++ [Galileo]
- Bjarne Stroustrup: Einführung in die Programmierung mit C++
- Heinz Tschabitscher: Einführung in C++
http://ladedu.com/cpp/zum_mitnehmen/cpp_einf.pdf
- LearnCPP.com <http://www.learncpp.com/>
- CProgramming.com <http://www.cprogramming.com/>
- Google C++ Style Guide: <https://google.github.io/styleguide/cppguide.html>

Programmierpraktikum C und C++



Grundlagen

(Übungsblatt: [G])



Dr.-Ing. Eric Lenz
elenz@iat.tu-darmstadt.de

Fachgebiet Control and Cyber-Physical Systems (CCPS)

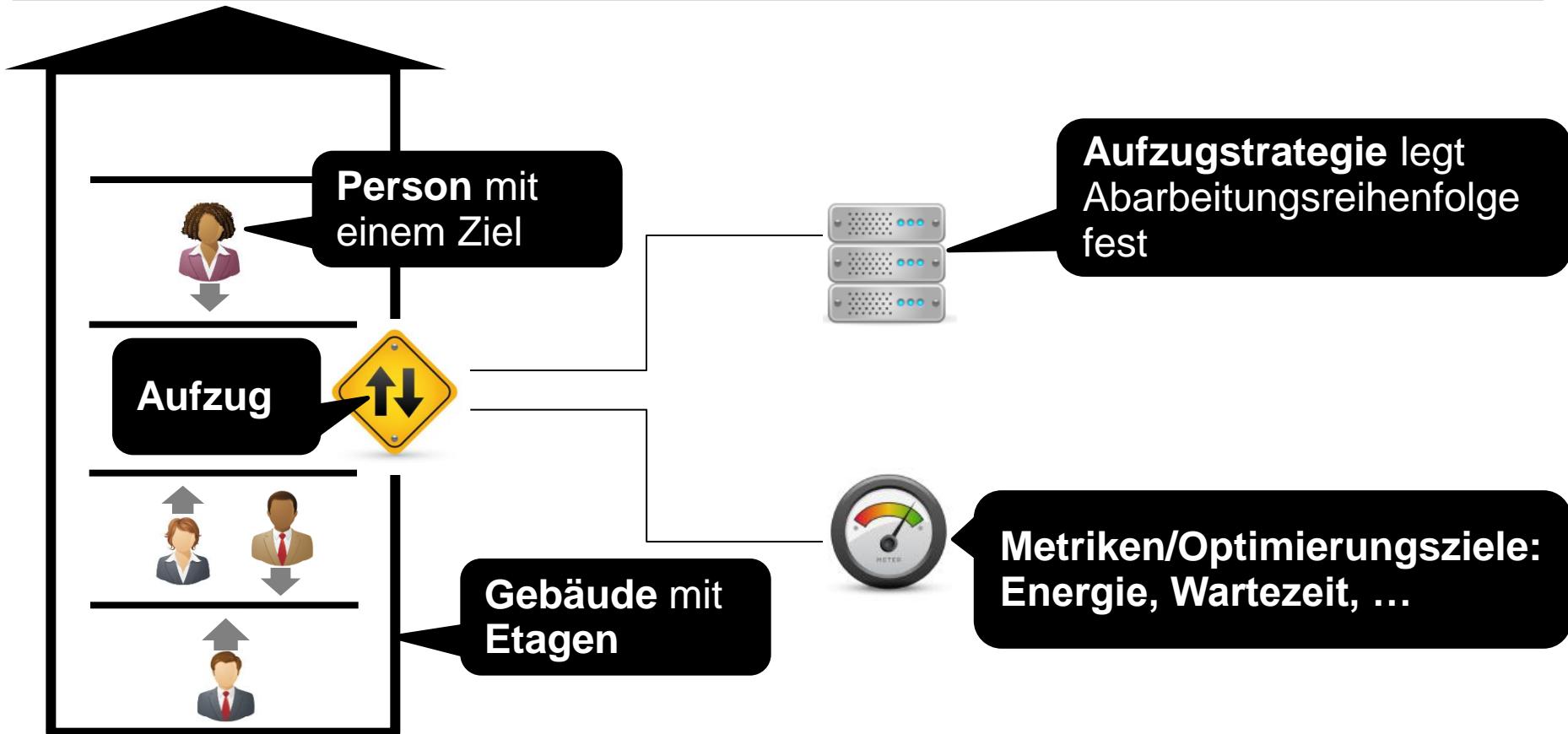
Prof. Dr.-Ing. Rolf Findeisen
Dept. of Electrical Engineering and Information Technology
<https://www.ccps.tu-darmstadt.de/>



TECHNISCHE
UNIVERSITÄT
DARMSTADT

LAUFENDES BEISPIEL

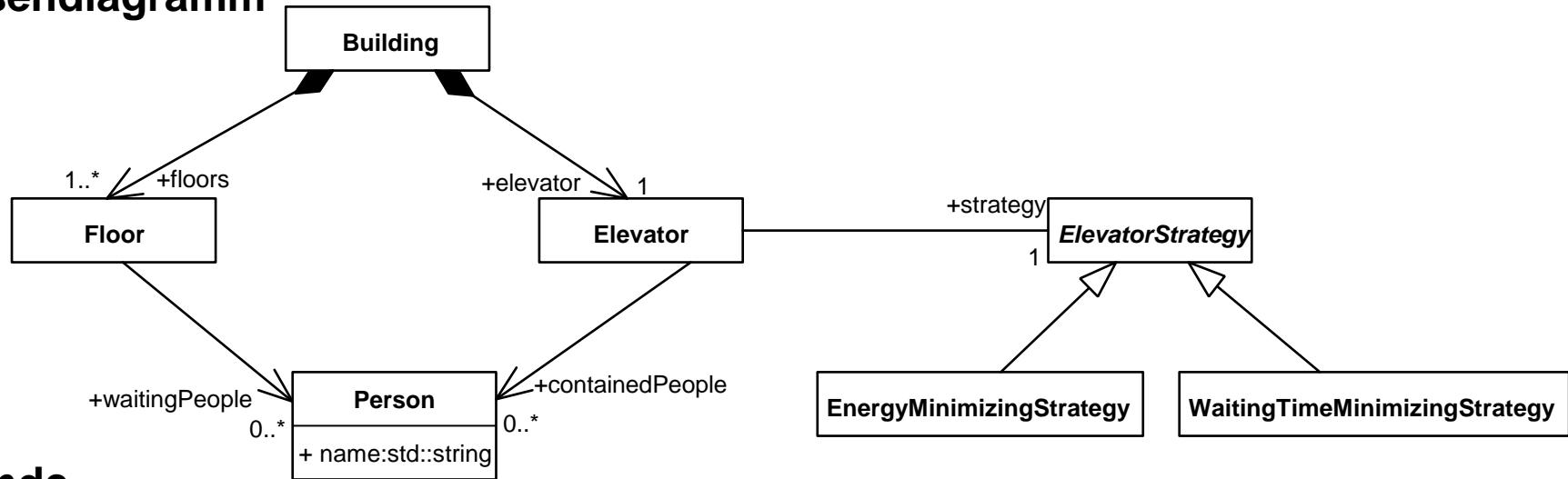
Laufendes Beispiel: Aufzugsimulation



Laufendes Beispiel: Klassendiagramm



Klassendiagramm



Legende

- Building** **ElevatorStrategy** Klasse und abstrakte Klasse
- Floor** **Person** **+ name : String** Assoziation mit Rollenname und Multiplizität und String-Attribut von Person
- ElevatorStrategy** **EnergyMinimizingS.** Vererbung ("EnergyMin.S. ist eine ElevatorS.")
- Building** **Floor** Aggregation ("Ein Floor ist immer Teil eines Buildings")



TECHNISCHE
UNIVERSITÄT
DARMSTADT

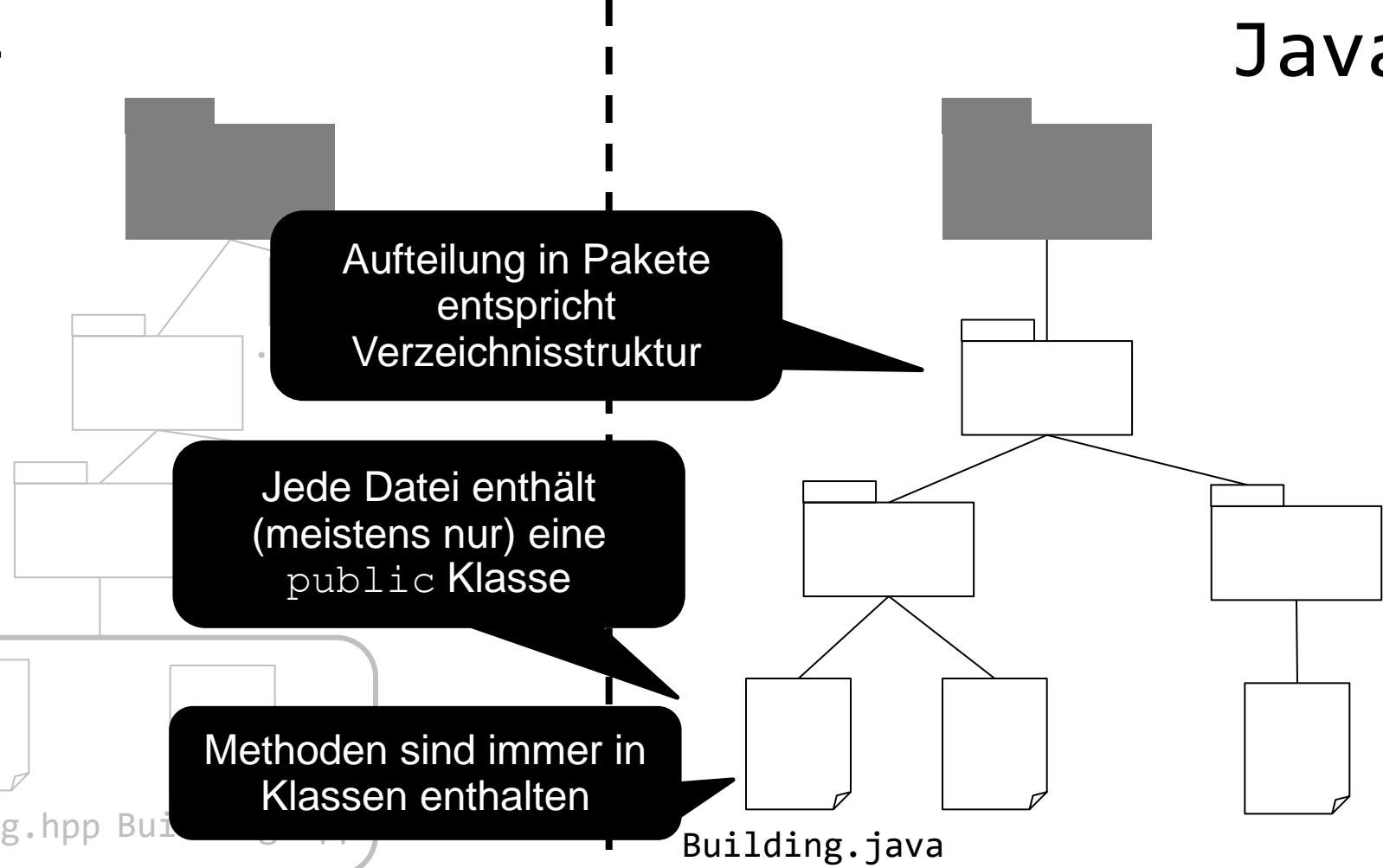
PROJEKTSTRUKTUR

Projektstruktur



C++

Java



Projektstruktur



C++

Java

Implementierungsdateien mit
Funktionen (nicht Methoden!)
sind möglich und üblich

Beliebige Verzeichnisstruktur -
hat nichts mit Sichtbarkeit zu tun

Klassen werden in **Header-** und
Implementierungsdatei getrennt

Mehrere Klassen können flexibel in
Header/Implementierungsdateien
definiert werden

Building.hpp Building.cpp

Header und Implementierungs-Dateien



```
/*
 * Part of the elevator simulation
 * A Building is a container for
 * Floors and the Elevator
 */

#ifndef BUILDING_HPP_
#define BUILDING_HPP_

#include <vector>

#include "Floor.hpp"
#include "Elevator.hpp"

class Building {
public:
    Building(int numberOffFloors);
    ~Building();

    void runSimulation();

private:
    std::vector<Floor> floors;
    Elevator elevator;
};

#endif /* BUILDING_HPP_ */
```

Kommentare wie in Java
/* ... */ mehrzeilig
// einzeilig

Include-Anweisungen wie Import-Befehle in Java:
<...> für Bibliotheken ("System")
"..." für eigenen Code ("Projekt")

Definition der Klasse mit
Deklaration der Methoden

"Members" = Methoden + Attribute

Der Header enthält die
nach "außen" sichtbare
Schnittstelle einer Klasse

Header und Implementierungs-Dateien



```
#include <iostream>
#include "Building.hpp"

using std::cout;
using std::endl;

Building::Building(int number_of_floors) :
    floors(number_of_floors, Floor()) {
    cout << "Creating building with "
        << number_of_floors << " floors."
        << endl;
}

Building::~Building() {
    cout << "Destroying building." << endl;
}

void Building::runSimulation() {
    cout << "Simulation running ..." << endl;
}
```

Header-Datei wird eingebunden

Using-Befehle sind wie statische Imports in Java (*cout* statt *std::cout*)

VORSICHT: using's sollten stets hinter den #includes auftreten.

Methoden werden implementiert
(Details später)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

KOMPILEIERUNG



Compile, Link, Load Time



- Drei aufeinander aufbauende Phasen von Quellcode zur Ausführung
- **Compile Time**
 - Übersetzung einzelner Einheiten (Dateien) in Objektcode (.java → .class, .c/.cpp → .o)
 - Alle verwendeten Namen müssen in einer Einheit deklariert, aber nicht definiert sein.
- **Link Time**
 - Auflösung von Abhängigkeiten ("externe Symbole") zwischen den Object Files
 - C++: Zu jedem verwendeten Namen muss es (genau) eine Definition geben.
- **Load Time**
 - Vorbereitung und Start der Programmausführung durch das Betriebssystem
 - Speicherbereich zuordnen, dyn. Abhängigkeiten laden, Ausführung von main beginnen



- | | | |
|--|--|--|
| • Initiator: Entwickler | • Initiator: Entwickler | • Initiator: Benutzer |
| • 1x je Übersetzungseinheit
(c.-Datei) | • 1x je Bauvorgang | • Durchgeführt mithilfe
Betriebssystem |
| • Durchgeführt mithilfe
Compiler | • Durchgeführt mithilfe
Compiler | |

Statisches und dynamisches Linken



Statisches Linken

(Static Libraries und Shared Archives)

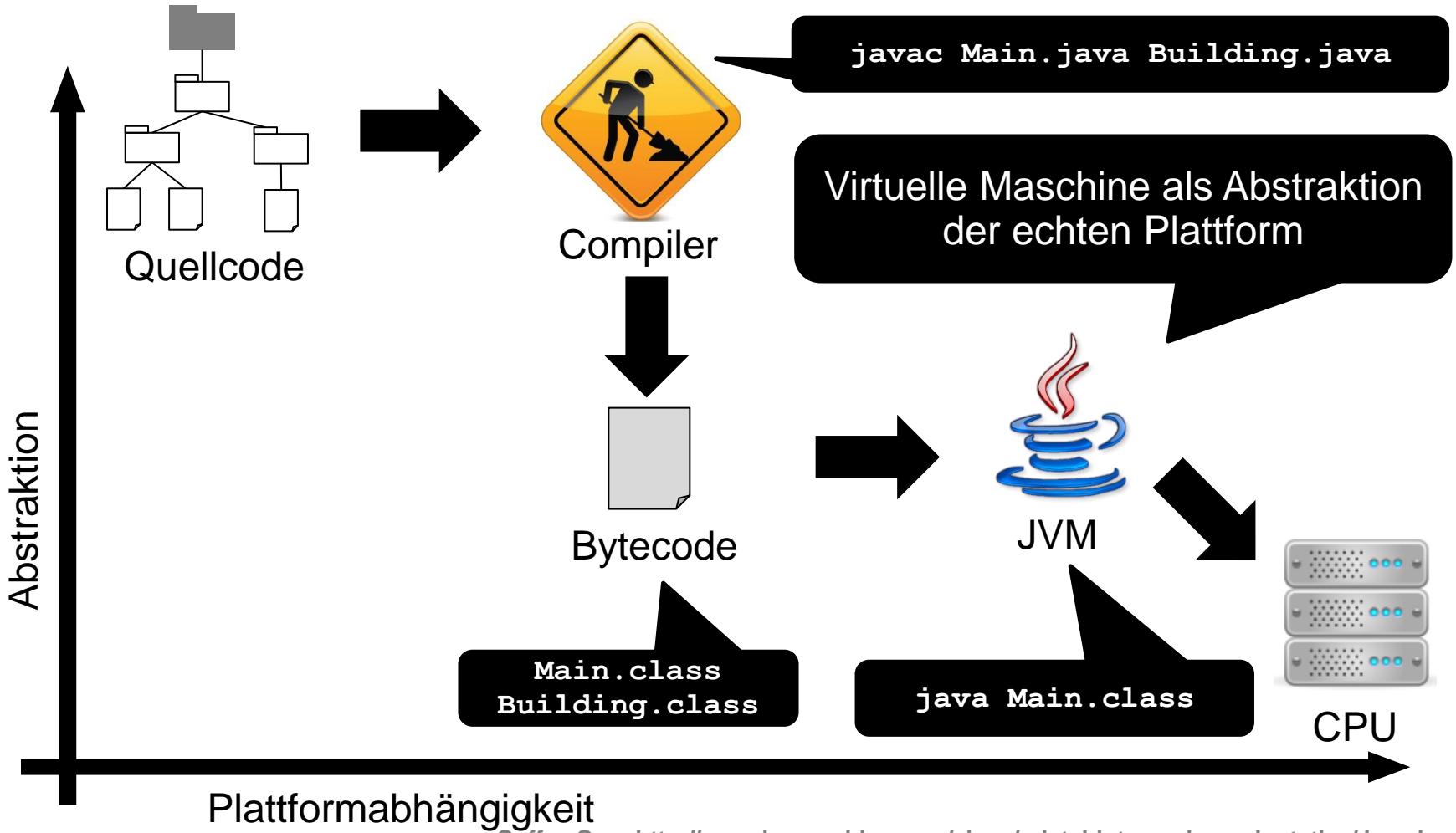
- Bibliothek muss zur **Linkzeit** vorhanden sein.
 - "Kopie" der Bibliothek wird im Compilat (*main.exe*) abgelegt.
 - Unterschied zwischen SL und SA eher klein
- Compilat ist "**standalone**", aber (oft wesentlich) **größer** als beim dynamischen Linken

Dynamisches Linken

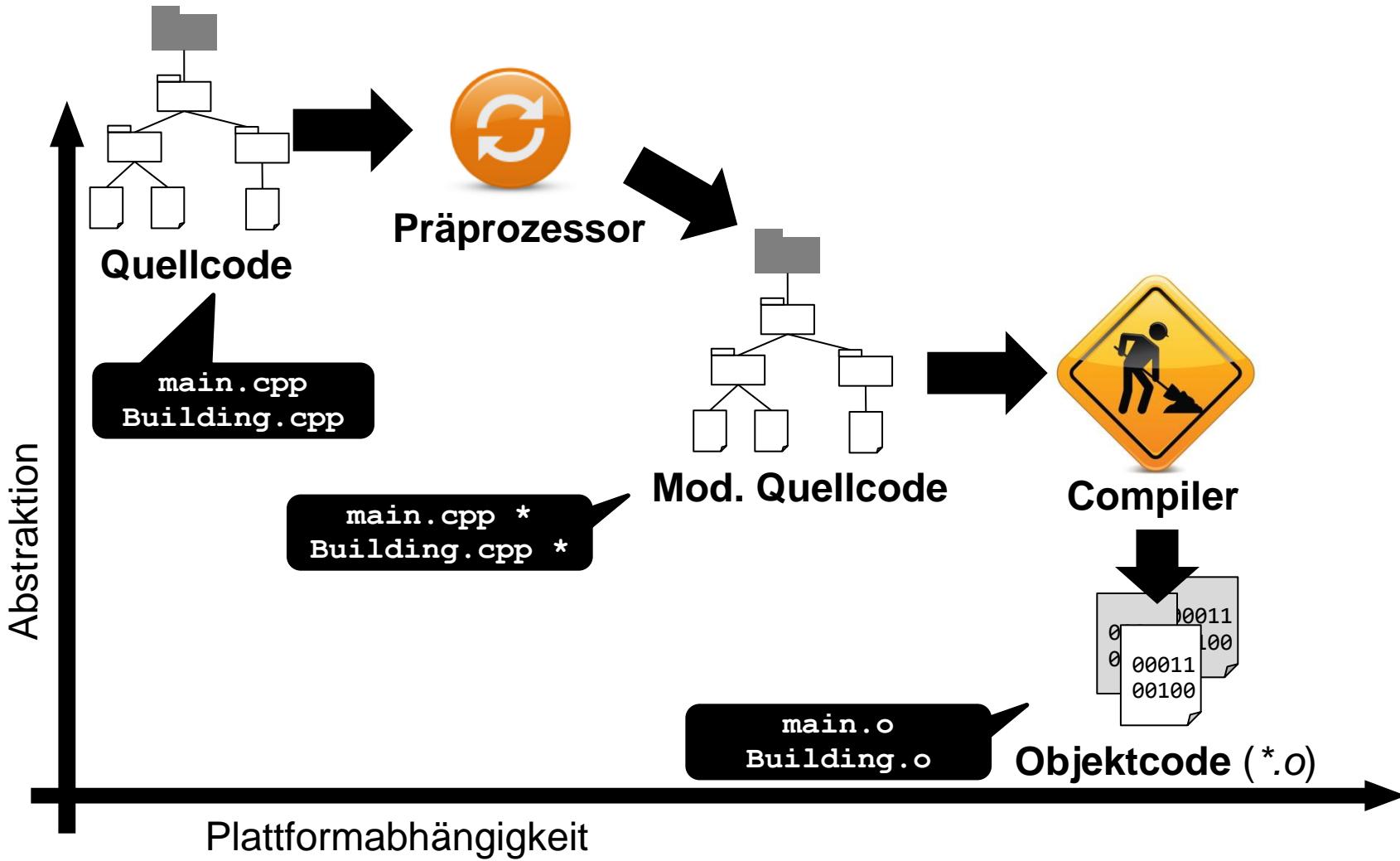
(Shared Objects und DLLs)

- *Shared Objects* müssen zur **Linkzeit** und zur **Ladezeit** vorhanden sein.
- *DLLs* müssen **nicht** zur **Linkzeit** und **nur beim konkreten Aufruf** zur **Laufzeit** verfügbar sein (werden aber zur Linkzeit von einer LIB-Datei begleitet).
→ Compilat ist "**minimal**", braucht aber zur Laufzeit **zusätzliche Abhängigkeiten**

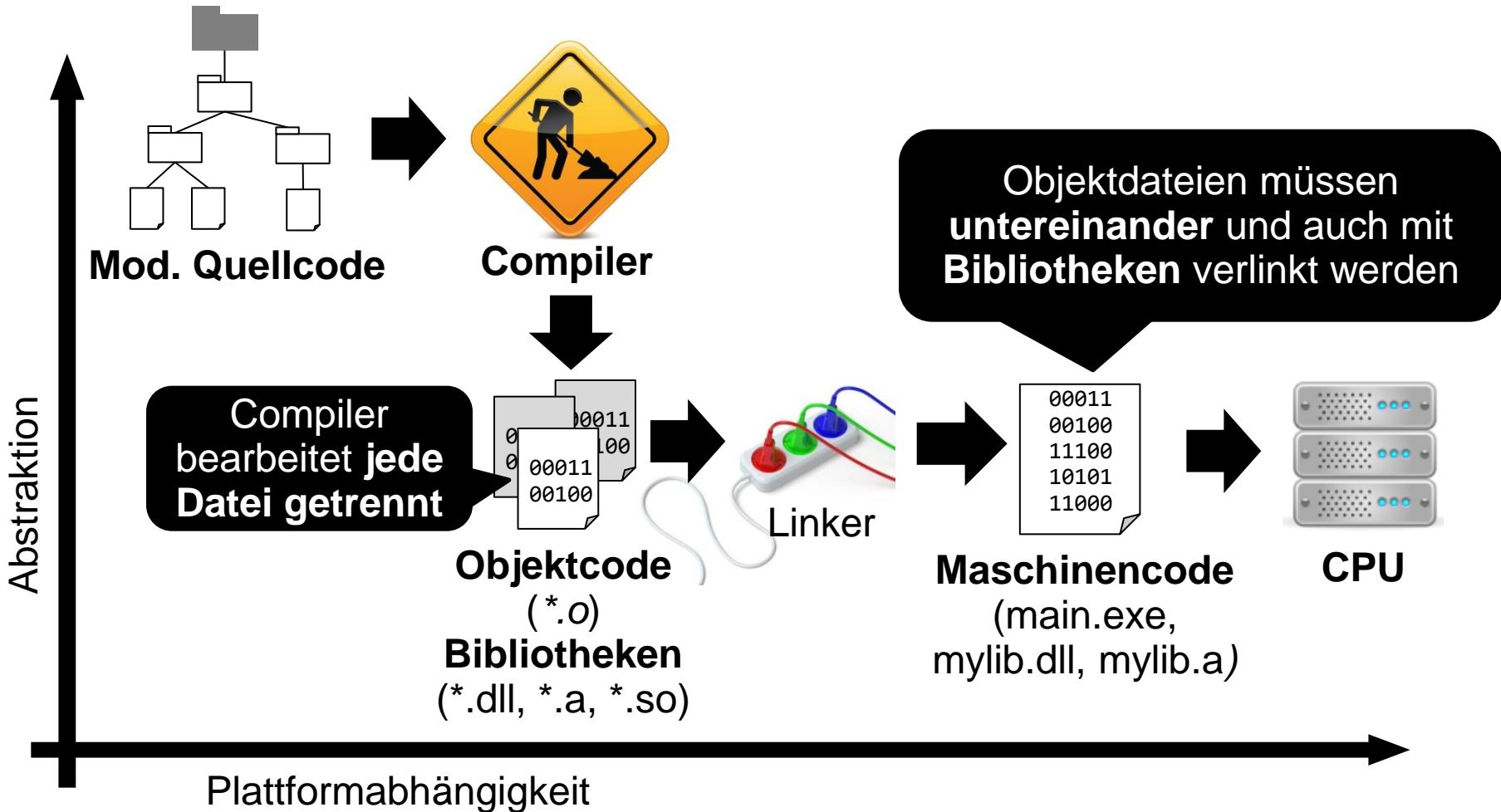
Kompilierung in Java



Kompilierung für C/C++ I



Kompilierung für C/C++ II



Unterschiede zwischen Java- und C/C++-Compiler



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Java

- **Java-Code → Java-Bytecode** (relativ ähnliche Struktur)
- 1:1-Beziehung zwischen .java- und .class-Dateien → inkrementelle Compilierung
- Sprachumfang **von Java deutlich kleiner als C++**
- **Optimierungen:** fast ausschließlich zur **Laufzeit** durch die JVM
- Auflösung **externer Abhängigkeiten** über Java **Classpath** (~ dyn. Linken)

C/C++

- **C/C++-Code → Assembler-Code**
 - Komplexere Transformation, (oft) inklusive Linken
 - Im einfachsten Fall: eine "fette" Datei als Ergebnis → keine Inkrementalität
- Auflösung von **externen Abhängigkeiten** über **#include** (Compile-Schritt) und statisches/dynamisches Linken (Link-Schritt)
- **Optimierungen zur Compile-Zeit**

Java vs. C++: (Vermeintliche) Stärken und Schwächen



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Plattformunabhängigkeit?

- **Java**: "Binary" kann "überall" verwendet werden ("Write once, run anywhere"). Achtung bei Pfad-/Dateinamen
- **C++**: Muss neu kompiliert werden, Standardbibliothek/STL/Boost stellen sicher, dass nur minimale Plattformabhängigkeiten bestehen ("Write once, compile anywhere")

Geschwindigkeit?

- (Ausführungsgeschwindigkeit vs. Entwicklungsgeschwindigkeit (inkl. Testen/Debugging)?)
- **Java**: (sozusagen) interpretiert, aber mit Just-in-Time Compilation
- **C++**: deutlich größerer Sprachumfang, schwieriger zu meistern
- Spannendes Paper von Google: <https://research.google.com/pubs/pub37122.html>

Sicherheit?

- **Java**: Angriffe über Schwächen in der JVM möglich
- **C++**: Zahlreiche Angriffe über Speicherüberläufe (Stichwort: Nullterminierung bei Strings)

Was genau macht der Präprozessor?



```
#ifndef BUILDING_HPP_
#define BUILDING_HPP_

#include <vector>

#include "Floor.hpp"
#include "Elevator.hpp"

class Building {
public:
    Building(int numberOffFloors);
    ~Building();

    void runSimulation();

private:
    std::vector<Floor> floors;
    Elevator elevator;
};

#endif /* BUILDING_HPP_ */
```

Include Guard: schützt vor mehrmaligem Einbinden von *Building.h*

Dadurch können wir alle benötigten Header überall (beliebig oft) einbinden.

Funktionsweise:

- #define-Konstanten auswerten (→ #if(n)def) und ersetzen
- #include-Anweisungen durch Dateiinhalt ersetzen (rekursiv!)

Weitere Anwendungsfälle des Präprozessors:

- DEBUG vs. NDEBUG/RELEASE
- Betriebssystemerkennung (z.B. WIN32, UNIX)
- Konstanten (in C)

https://en.wikipedia.org/wiki/Include_guard

Was passiert ohne Include Guards?



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Vor dem Präprozessor

```
/* Building.hpp */  
#include "Floor.hpp"  
#include "Elevator.hpp"  
  
class Building {};
```

```
/* Elevator.hpp */  
#include "Floor.hpp"  
  
class Elevator {};
```

```
/* Floor.hpp */  
class Floor {};
```

Nach dem Präprozessor

```
/* Building.hpp */  
// #include "Floor.h"  
  
class Floor {};  
  
// #include "Elevator.hpp"  
  
// #include "Floor.hpp" (recursive)  
  
class Floor {};  
  
class Elevator {};  
  
class Building {};
```

One-Definition Rule:
Jede Klasse/Methode/...
durf höchstens einmal
definirt werden

**Die meisten Probleme beim Arbeiten mit C++
gehen auf Regelverletzungen zurück.**

http://en.cppreference.com/w/cpp/language/definition#One_Definition_Rule

Was passiert ohne Include Guards? Lösung.



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Vor dem Präprozessor

```
/* Building.hpp */  
#include "Floor.hpp"  
#include "Elevator.hpp"  
  
class Building {};
```

```
/* Elevator.hpp */  
#include "Floor.hpp"  
  
class Elevator {};
```

```
/* Floor.hpp */  
#ifndef FLOOR_HPP_  
#define FLOOR_HPP_  
  
class Floor {};  
  
#endif
```

Nach dem Präprozessor

```
/* Building.hpp */  
// #include "Floor.h"  
// FLOOR_HPP_ undefined -> defined  
  
class Floor {};  
  
// #include "Elevator.hpp"  
  
// #include "Floor.hpp" (recursive)  
  
class Elevator {};  
  
class Building {};
```

Include Guards: #ifndef vs. #pragma once



- Anstelle der Klammer aus #ifndef, #define, #endif kann man auch #pragma once verwenden → **kompakter, weniger fehleranfällig**
- #pragma once ist **(noch) nicht im Standard**, wird aber von den meisten Compilern unterstützt.
- Liste von kompatiblen Compilern:
https://en.wikipedia.org/wiki/Pragma_once#Portability

```
#ifndef BUILDING_HPP_
#define BUILDING_HPP_

#include <vector>
#include "Floor.hpp"
#include "Elevator.hpp"

class Building {
public:
    Building(int number_of_floors);
    ~Building();

    void runSimulation();
};

#endif /* BUILDING_HPP_ */
```

```
#pragma once

#include <vector>
#include "Floor.hpp"
#include "Elevator.hpp"

class Building {
public:
    Building(int number_of_floors);
    ~Building();

    void runSimulation();
};
```

Anwendungsmöglichkeiten von #define

Die Direktive #define kann auf drei Arten eingesetzt werden

- **Symbol:** `#define BUILDING_HPP`
 - Das Symbol BUILDING_HPP existiert (ohne Wert).
- **Konstante:** `#define BUILDING_HPP 1`
 - Alle Auftreten von BUILDING_HPP werden mit 1 ersetzt
 - Heute unüblich, dank "const" und "static" (siehe später)
- **Makro:** `#define MAX(a,b) ((a < b) ? (b) : (a))`
 - Verwendung: `std::cout << "MAX(1,2)" : " " << MAX(1,2) << std::endl;`
 - N.B.: **Ternärer Operator:** If-Condition ? Then-Value : Else-Value



Grundlegendes Konzept in den meisten Programmiersprachen!

- **Deklaration**
 - ... gibt an, dass ein Element (z.B. Variable, Funktion, Klasse) **existiert ohne** ihm dabei einen **konkreten Wert** zuzuweisen oder **Speicher zu reservieren**.
 - **Beispiele:** `extern int x; void myFunction(); class MyClass;`
- **Definition**
 - ...**reserviert Speicher** und belegt ein Element implizit mit einem **Wert**
 - Eine **Redefinition** ist nicht möglich, eine neue **Zuweisung** hingegen schon.
 - **Beispiele:** `int x; int x=3; void myFunction() /* function def. */;`
`class MyClass /* class def. */; MyClass::MyClass() /*...*/`
 - **Achtung:** `int x;` ist immer eine Definition, auch ohne Gleichheitszeichen! (vgl. Initialisierung)
- Deklaration und Definition können **gleichzeitig geschehen**
 - Trennung erlaubt es **zyklische Abhängigkeiten aufzubrechen**.
- **Initialisierung:**
 - Belegt eine Variable explizit mit einem Wert. (z.B. `int x = 3;`)
 - Explizite Zuweisung ist vorzuziehen!

Praktisches Beispiel: <https://www.geeksforgeeks.org/difference-between-definition-and-declaration/>

Definitionen: <https://en.cppreference.com/w/cpp/language/definition> <https://en.cppreference.com/w/cpp/language/declarations>

Konzepte und Konventionen sind in C++ wesentlich

Konzept



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- C++ vertraut dem Programmierer – **alles** ist möglich.

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off" - B. Stroustrup, 1986

- **Konventionen** sind in C++ wesentlich, werden tw. mittels Schlüsselwörtern spezifiziert und vom Compiler überprüft:

- `void f() noexcept` garantiert, das f keine Exceptions wirft.

▪ Konzepte:

▪ One-Definition Rule

- Variablen/Methoden/Klassen dürfen nur einmal definiert werden.

▪ Undefined Behavior (UB)

- UB tritt ein, wenn Code auf eine nicht-spezifizierte Weise aufgerufen wird

▪ Const Correctness

- Schutz vor ungewollten Zustandsänderungen, vgl. `final` Variablen neu zuweisen in Java

https://en.wikipedia.org/wiki/One_Definition_Rule

<https://isocpp.org/wiki/faq/const-correctness>



- **Definition:** Konstrukte mit UB lassen ein Programm bedeutungslos werden. Ein Compiler kann im Falle von UB mit Fehlermeldung abbrechen oder Code mit beliebigem Verhalten generieren.
- **Beispiele:**
 - Dereferenzieren von null: `int *nP = null; int x = *nP;`
 - Division durch 0: `int y = x/0;`
 - Konstanten nach `const_cast` manipulieren:
 - Fehlendes return-Statement: `int f {/*no return statement*/}`
 - Zugriff auf uninitialisierte Variablen: `int b; int a = b + 1;`
- **Warum wird UB überhaupt vom Compiler zugelassen?**
 - Der Hauptgrund dürfte Performance-Steigerung und Ressourcen-Minimierung sein (z.B. kein 0-Check beim Dividieren).

<http://en.cppreference.com/w/cpp/language/ub>

<http://blog.regehr.org/archives/213>



TECHNISCHE
UNIVERSITÄT
DARMSTADT

PROGRAMMSTART

Systemstart



- **main-Funktion in C++ entspricht main-Methode in Java.**
- **Zwei Formen** werden vom **Linker** erkannt:
 1. parameterlos (`int main()`)
 2. mit Kommandozeilenparametern (`int main(int argc, char **argv)`, `argv[0]` enthält Pfad zum Programm)

```
#include "Building.hpp"

int main() {
    Building building(3);
    building.runSimulation();
}
```

Kein Rückgabewert
(= `return 0;` = alles OK)

```
#include <iostream>
#include <cstdlib>
#include "Building.hpp"
int main(int argc, char **argv){
    if (argc >= 2) {
        unsigned int levels = std::atoi(argv[1]);
        Building hbi(levels);
        hbi.runSimulation();
    }
}
```

Arrays

C++-Datentypen können
vorzeichenlos (unsigned)
sein.

Die Deklarationsreihenfolge ist wichtig!



- Der C++-Compiler analysiert jede Datei von **vorne nach hinten** – einfach, aber effizient.

```
int main() {  
    myFunction();  
}  
  
void myFunction() {}
```

- Abhilfe:** Aufrufende Funktion ans Ende stellen (geht nicht immer) oder Funktion deklarieren.

```
// Declaration of myFunction  
void myFunction();  
  
int main() {  
    myFunction();  
}  
  
// Definition of myFunction  
void myFunction() {}
```

Funktionsprototyp



Eine lose Übersicht über

WEITERE KONZEPTE IN C++

Enumerationen



- Enumerationen sind **Klassen mit einer beschränkten Anzahl von Instanzen**
- Enum-Konstanten können **ganzzahlige Werte zugewiesen werden** (DOWN=-1)
- Es existieren auch **anonyme Enumerationen**
 - z.B. `enum { UP, DOWN } elevatorButton;` (führt Variable `elevatorButton` ein)
- **Java:** Enumerationen können zusätzlich Konstruktoren, Methoden, Felder haben

```
#include <iostream>
enum Button {
    UP, DOWN = -1
};
int main() {
    Button d = UP;
    switch(d) {
        // Traditional style
        case UP:           std::cout << "Up!" << std::endl; break;
        // Since C++11
        case Button::DOWN: std::cout << "Down!" << std::endl; break;
        default:           std::cout << "Cannot handle this!" << std::endl;
    }
}
```

<http://en.cppreference.com/w/cpp/language/enum>

Switch-Case



- Als Bedingung von switch-case sind in C++ nur ganzzahlige Typen (**int**, **long**, ...) und Enumerationen möglich (+ Referenzen darauf).
- **Falldefinition** mittels case-Label (z.B. **case UP:**)
- Jeder **Fall** sollte beendet werden mittels **break;** (sonst "fall through")
- Falls kein Fall zutrifft, kann man **default:** als Standardfall nutzen.
- **Java:** Seit 1.7 auch Strings möglich.

```
#include <iostream>
enum Button {
    UP, DOWN = -1
};
int main() {
    Button d = UP;
    switch(d) {
        // Traditional style, may conflict with preprocessor
        case UP:           std::cout << "Up!" << std::endl; break;
        // Since C++11
        case Button::DOWN: std::cout << "Down!" << std::endl; break;
        default:           std::cout << "Cannot handle this!" << std::endl;
    }
}
```

Namenskonflikte vermeiden mit Namespaces



- **Java:** package x.y.z; class X;
 - **Zugriff:** import und import static in Java
 - **Standardnamensraum:** [leer]
- **C++:** namespace x{};, class X{};, struct X{};
 - **Zugriff:** using-Direktive zum Importieren
 - **Standardnamensraum:** namespace{} → ::sum(1,2);

```
int sum(int a, int b)
{ return a+b; }

namespace my_utils {
    int sum(int a, int b)
    { return a+b; }
}

class MyUtils {
public:
    int sum(int a, int b);
};

MyUtils::sum(int a, int b)
{ return a+b; }
```

```
int main1() {
    sum(1,2);
    my_utils::sum(1,2);

    using my_utils::sum; // ERROR<-conflict
}

int main2() {
    using my_utils::sum;
    sum(1,2);
}
```

<http://en.cppreference.com/w/cpp/language/namespace>



Java

- Quote-Literale werden zu **java.lang.Strings**
- Beispiel:
 - `System.out.println("Hello World".getClass()); // java.lang.String`

C++

- Quote-Literale werden zu **C-Strings = char-Arrays**
- Beispiele:
 - `const char *myString = "Hello World.";` // A C-style string
 - `std::string myString2("Hello World.");` // explicit constructor invoc.
 - `std::string myString3 = "Hello World";` // implicit constructor invoc.

class vs. struct vs. union



In C++ gibt es (mind.) drei Wege zur Implementierung "komplexer Datentypen".

- **class**
 - Standardmittel in C++
- **struct**
 - **Geerbt von C** (→ µC-Teil), u.a. für Binärkompatibilität (z.B. Datentypen in `ctime`)
 - In C++: **Konstruktor, Methoden, Vererbung** möglich
 - Unterschied zu class: standardmäßig sind alle Member **public**
- **union** [eher exotisch]
 - **Spezialdatentyp**, zur Speicherung "alternativer" Member
 - Alle Felder belegen den selben Speicher
 - Höhere **Effizienz** durch gemeinsame Speichernutzung

```
struct RawVector3D {  
    int x;  
    int y;  
    int z;  
};  
  
int main() {  
    RawVector3D myVec;  
    myVec.x = 5;  
}
```

```
union ResultValue {  
    int exitCode;  
    bool flag;  
};  
  
int main() {  
    ResultValue result1;  
    result1.exitCode = 3;  
    result1.flag = false;  
}
```

<http://en.cppreference.com/w/cpp/language/union>



Neben **rein dezimalen Ganzzahlliteralen** (z.B. in `int x = 125`) gibt es weitere Möglichkeiten, Ganzzahlliterale anzugeben

Suffixe

- 'u' oder 'U' stellt sicher, dass das Literal als **vorzeichenlos** interpretiert wird (z.B. 255u)
- 'l', 'L', 'll', 'LL' stellt sicher, dass das Literal als '**long long int**' interpretiert wird (z.B. 2345678901234567890LL). Seit C++11 kein Unterschied mehr zwischen L und LL.

Infixe

- (**Seit C++14**) **Hochkommata** können an beliebigen Stellen eingesetzt werden (z.B. 18'446'744'073'709'550'59211u)
- Kombinationen beider Suffixe sind möglich.

Präfixe

- **Oktaldarstellung:** führende 0 bewirkt Interpretation als Oktalliteral (z.B. 0753 entspricht $7*64+5*8+3=491$). Kombination mit U/L möglich.
- (**Seit C++14**) **Binärdarstellung:** führende 0b bewirkt Interpretation als Binärliteral (z.B. 0b1010'1101 entspricht 0xAD=10*16+13=173) . Kombination mit U/L möglich.

Seit C++11 kann man übrigens **eigene Literaltypen** definieren ("user literals"). Die Suffixe u, U, l, L, ll, LL und die Präfixe 0x und 0 funktionieren auch in C.

https://en.cppreference.com/w/cpp/language/integer_literal
https://en.cppreference.com/w/cpp/language/user_literal



Java

- Operatoren in **Sonderrolle, fest belegt** ("Lehre aus Erfahrung mit C++)
- Fixe **Präzedenz** (= Abarbeitungsreihenfolge bei mehreren Operatoren)
 - Im Ausdruck `int x = a + 1;` wird zuerst `'+'` und dann `'='` ausgewertet.
 - **Beispiel:** `++, --` (Postfix) vor `++,--,+, -,~, !` vor `*, /, %` vor ...

C++

- Operatoren als **Syntactic Sugar** und beliebig überschreibbar
 - `a + b` gleichwertig zu `operator+(a,b)` oder `a.operator+(b)`
 - Extrem wichtig: Zuweisungsoperator `operator=` (siehe später)
- Fixe **Präzedenz**
 - `::` vor `++, --` (Postfix), `(), [], ., ->` vor `++, --, ...`

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html>

<http://en.cppreference.com/w/cpp/language/operators>

http://en.cppreference.com/w/cpp/language/operator_precedence



- **Member:** Oberbegriff für Attribute und Methoden
- Gültigkeit: je Bereich, nicht je Member
- Nur innerhalb einer class-Definition für Members möglich
 - **public:** alle folgenden Members sind unbegrenzt nutzbar
 - **protected:** alle folgende Members sind nur in dieser und Unterklassen nutzbar
 - **private:** alle folgenden Members sind nur in dieser Klasse nutzbar
 - **friend f()** erlaubt Funktion/Methode f() Zugriff auf **private** Members dieser Klasse
- **Anders als in Java:** keine package-/default-Sichtbarkeit
 - via ::-Operator oder **using** können alle Funktionen und public-Methoden genutzt werden
 - Beispiel: `using std::string;`

Das Schlüsselwort static



Java

- Markiert **Zugehörigkeit zur Klasse**, nicht zu einer Instanz
 - z.B.: `static int instanceCount = 0; // incremented in constructor`
- Definition von **Konstanten**
 - z.B. `private static final int MAX_FLOOR_COUNT = 100;`

C/C++:

- **Als Zugehörigkeitsmodifikator und für Konstanten:**
 - Verwendung wie bei Java
 - Kombination mit `const`-Modifier (s. später)
 - z.B. `static const int MAX_FLOOR_COUNT = 100;` (in class-Definition)
 - z.B. `const int Building::MAX_FLOOR_COUNT = 100;` (außerhalb, **ohne** `static!`; One Definition Rule beachten)
- **Als Sichtbarkeitsmodifikator:**
 - Vor allem in C (s.a. später)
 - `static` Funktion/Variable ist nur innerhalb der Implementierungsdatei sichtbar; ansonsten: globale Funktion/Variable

<http://en.cppreference.com/w/cpp/keyword/static>

Inlining und Code-Optimierung



```
class Floor {
public:
    Floor(int number);
    Floor(const Floor& floor);
    ~Floor();

    inline int getNumber() const {
        return number;
    }

    inline void setNumber(int n) {
        number = n;
    }

private:
    int number;
};

inline int max(int a, int b) const {
    return a >= b ? a : b;
}
```

Floor.hpp

- **inline** zeigt **Wunsch** an, dass statt eines Methoden-/Funktionsaufrufs direkt der Code an jeder Aufrufstelle eingefügt werden soll.
- **inline** ist innerhalb einer class-Definition redundant.
- Nur ein **Hinweis** an den Compiler – nicht "verpflichtend" und oft **nicht notwendig**, da der Compiler automatisch über Optimierungen entscheidet (Flags -O1, -O2, -O3, ...)

https://en.wikipedia.org/wiki/Inline_function



Java

- **For:** `for(int i = ...; i < ...; ++i){/*body*/}`
- **While-Do:** `while(/*condition*/*){/*body*/}`
- **Do-While:** `do {/*body*/} while(/*condition*/)`
- **Foreach:** `for(final String s : new String[]{"a", "b", "c"}{/*body*/})` (seit Java 1.7)
- **Iterator:** `Iterator<Object> iter = list.iterator(); while(iter.hasNext())
{Object o = iter.next();}`
 - z.B. um Elemente leicht überspringen zu können

C++

- **For:** `for(int i = ...; i < ...; ++i){/*loop body*/}` (wie in Java),
- **While-Do:** `while(/*condition*/*){/*loop body*/}` (wie in Java),
- **Do-While:** `do {/*loop body*/} while(/*condition*/)` (wie in Java),
- **STL:** `std::foreach(v.begin(), v.end(), /*function to execute*/)`
- **Iterator:** `for(std::vector<int>::iterator iter=v.begin();iter!=v.end();++iter)
{int x = *iter;}` (traditionell, STL-Stil)
- **Foreach:** `for (int i : {1,2,3,4,5}) {/*...*/}` (seit C++11, wie in Java)

STL: http://www.cplusplus.com/reference/algorithm/for_each/
C++11: <http://en.cppreference.com/w/cpp/language/range-for>

Standard-Bibliotheken in C++

- ISO-genormte, stetig wachsende Standardbibliothek
- Alle Komponenten liegen in `namespace std`
- Komponenten:
 - I/O (z.B. `<iostream>`, `<sstream>`)
 - Strings (z.B. `<string>`, `<regex>`)
 - Standard Template Library (STL)
 - Generische Datenstrukturen
(z.B. `<vector>`, `<array>`, `<list>`, `<prioriy_queue>`)
 - Generische Algorithmen
(z.B. `<algorithm>`, `<iterator>`, `<functional>`)

Boost: "Brutschränk" für C++-Standardkomponenten



TECHNISCHE
UNIVERSITÄT
DARMSTADT



<http://www.boost.org/>

"...one of the most highly regarded
and expertly designed C++ library
projects in the world."

Herb Sutter, Andrei Alexandrescu

Array

Filesystem

Lambda

Odeint

Chrono

Function(al)

Math
(advanced)

Smart Ptr

Date Time

Graph

MPI

System

Programmierpraktikum C und C++



Speicherverwaltung und Lebenszyklus – Teil 1

(Übungsblatt: [S])



Dr.-Ing. Eric Lenz
elenz@iat.tu-darmstadt.de

Fachgebiet Control and Cyber-Physical Systems (CCPS)

Prof. Dr.-Ing. Rolf Findeisen
Dept. of Electrical Engineering and Information Technology
<https://www.ccps.tu-darmstadt.de/>



Stack und Heap

WO LEBEN MEINE DATEN? ... UND WIE LANGE?

Speicherbereiche in C++



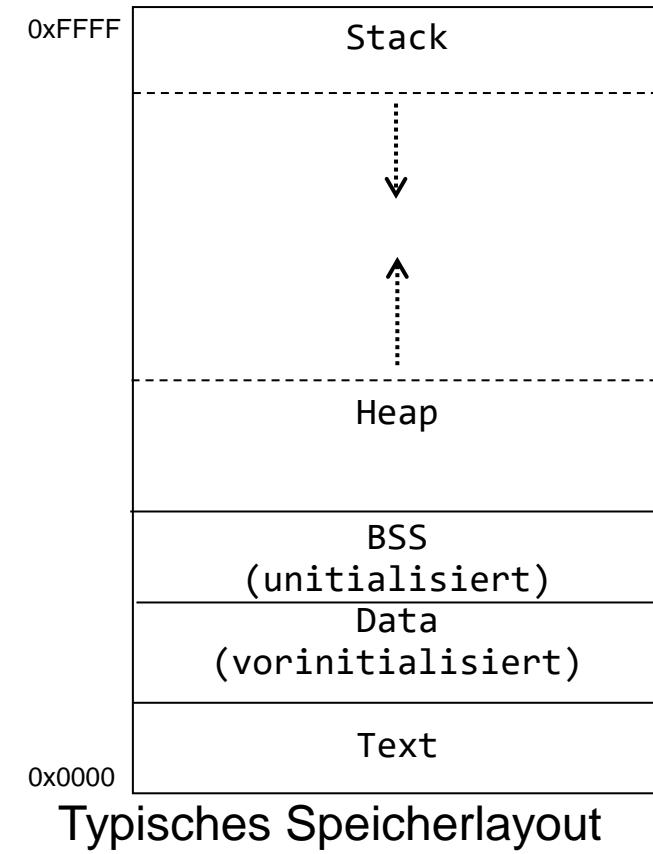
In C++ spielt die **Speicherverwaltung** eine **wesentlich größere Rolle** als in Java

Vier wesentliche Speicherbereiche ("Segmente")

- **Programmspeicher ("Text")**
Binären Programmcode, read-only.
- **Globaler Speicher ("BSS", "Data")**
Globalen Variablen und Konstanten

Für uns hier relevant.

- **Dynamischer Speicher ("Heap")**
Frei verwendbar; verwaltet durch Entwickler
- **Statischer Speicher ("Stack")**
Verwendung für lokale Variablen; verwaltet durch Compiler.



Stack vs. Heap



Stack

- Begrenzte Größe (lokale Variablen, Rücksprungadresse)
- **Speicherverwaltung** durch den Compiler
- Speicherverwaltung: *last-in first-out*

→ sehr effizient, statisch

Heap

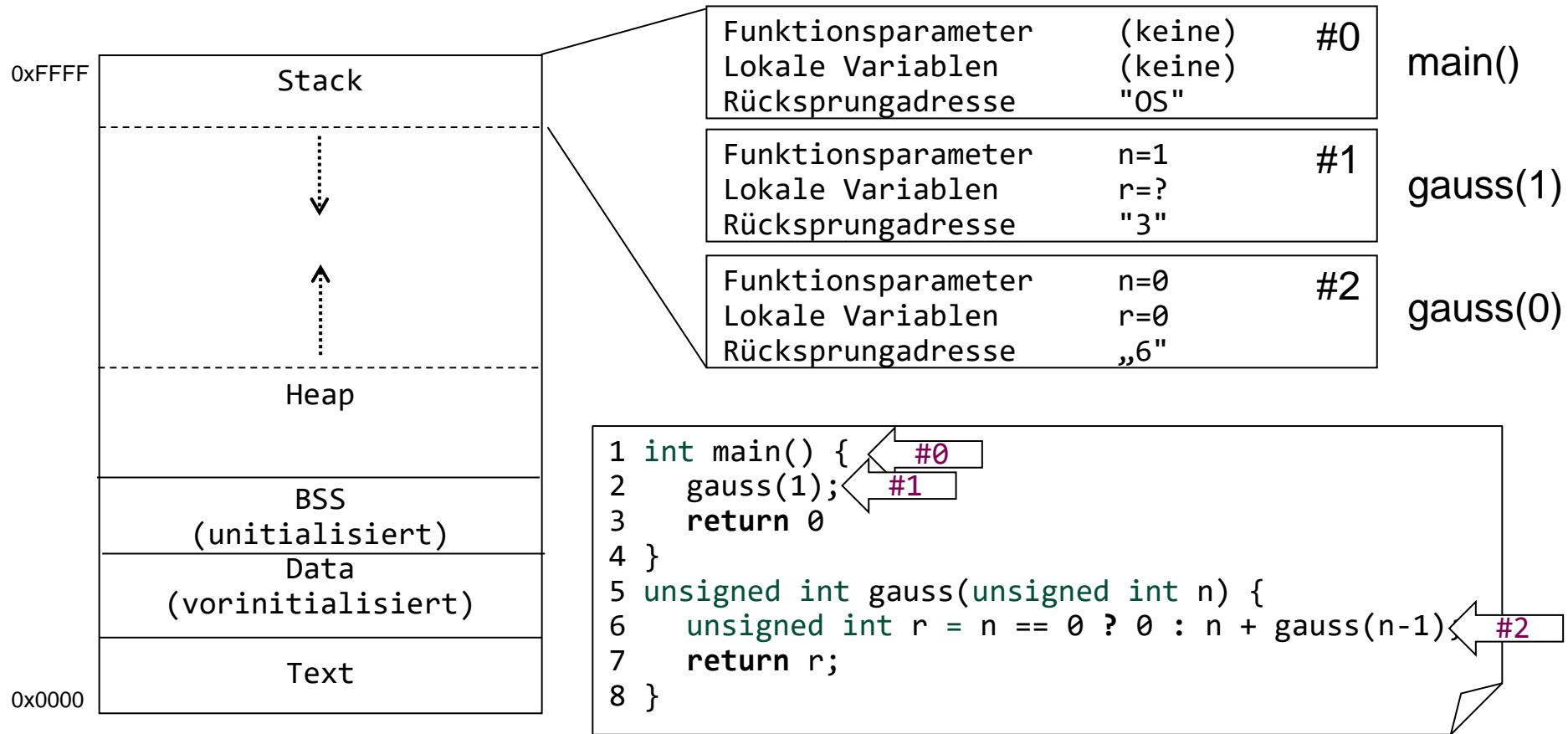
- Typischerweise wesentlich größer als Stack
- **Speicherverwaltung:** manuell, durch Entwickler mithilfe der Operatoren new, delete, delete[]

→ groß aber teuer (Laufzeit)

Stackframes



Ein **Stackframe** speichert Ausführungszustand einer Funktion

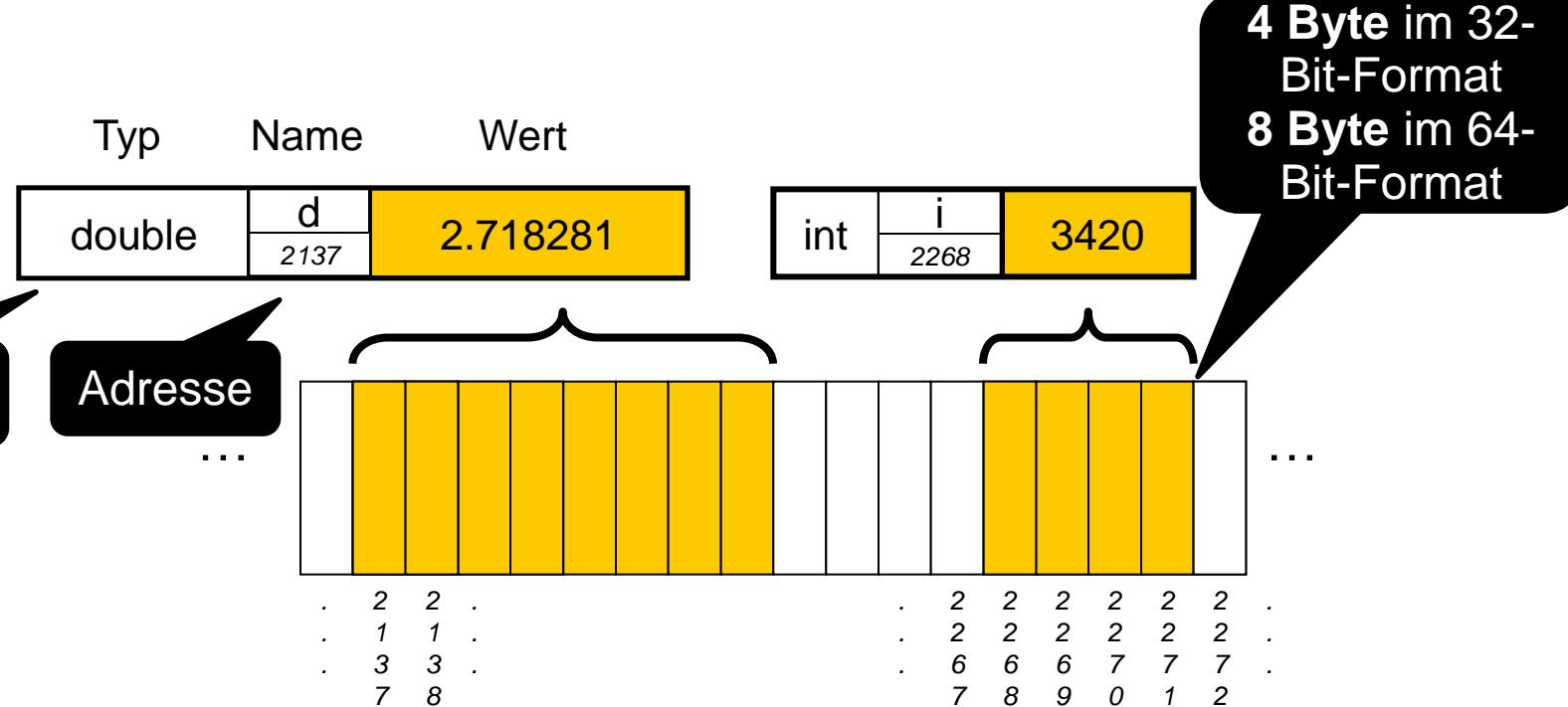


Variablen und Zeiger: Was ist eine Variable?



Eine **Variable** entspricht intern einer **Speicheradresse** mit einer **Menge von Speicherstellen**

Der **Typ einer Variable** bestimmt die **Größe** des reservierten Speicherplatzes und die **Interpretation** der enthaltenen Daten

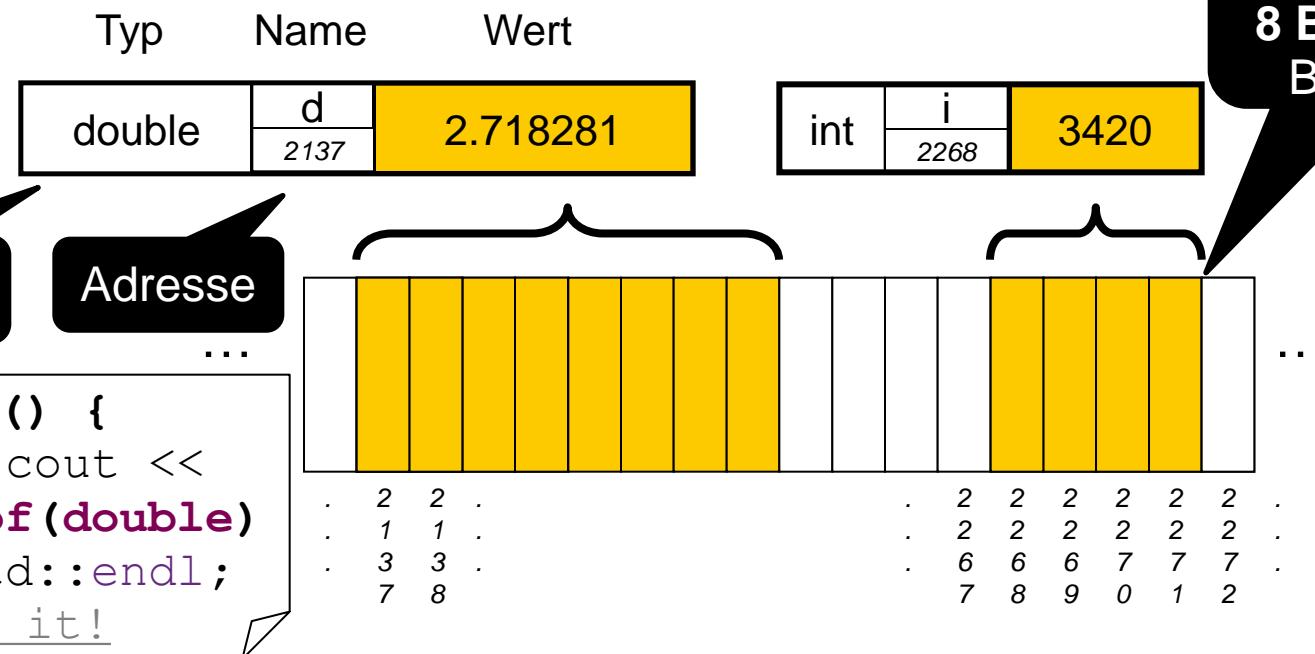


Variablen und Zeiger: Was ist eine Variable?



Eine **Variable** entspricht intern einer **Speicheradresse** mit einer **Menge von Speicherstellen**

Der **Typ einer Variable** bestimmt die **Größe** des reservierten Speicherplatzes und die **Interpretation** der enthaltenen Daten

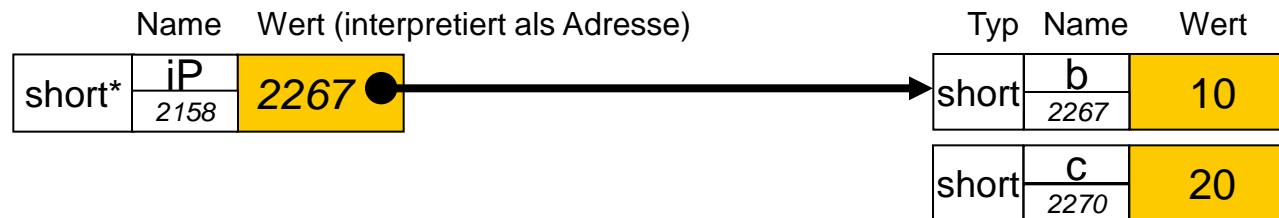
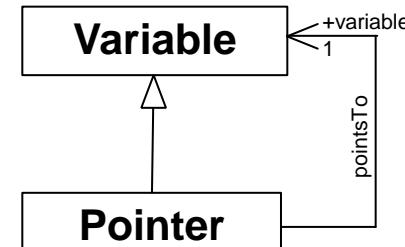


Variablen und Zeiger: Was ist ein Zeiger?



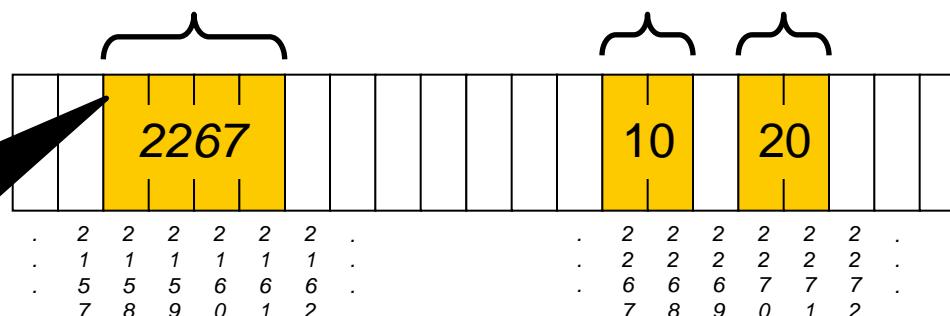
Ein **Zeiger (Pointer)** ist eine Variable, deren Inhalt als die Speicheradresse einer anderen Variable **interpretiert** wird

Der **Typ eines Zeigers** legt fest, auf welchen Typ von Variable "gezeigt" wird



short *iP = &b

short b short c



4 Byte im 32-Bit-Format

8 Byte im 64-Bit-Format

Variablen und Zeiger: Syntax



```
int i = 42;
```

Definition eines Zeigers vom Typ int* (Zeiger auf int; hat strenggenommen keinen Wert)

```
int *iP;
```

Zuweisung eines Zeigers vom Typ int* durch Zuweisung einer Adresse (Referenzierung)

```
iP = &i;
```

Dereferenzierung eines Zeigers, um den Inhalt zu erhalten

```
int j = *iP;
```

Ohne Dereferenzierung bekommt man den Wert des Zeigers (= die gespeicherte Adresse).

```
int *jP = iP;
```

Der Null-Pointer



Der Null-Pointer wird verwendet, um anzudeuten,
dass ein Pointer noch **keinen definierten Wert** hat.

0x0

- C:

```
int *i = 0; int *j = 0x0;
```

- C90

```
#include <stddef.h>
int *k = NULL;
```

- C++

```
#include <cstddef>
int *k = NULL;
```

- C++11

```
int *m = nullptr;
```

NULL

nullptr

Wie <stddef.h>, aber
mit Namespaces



Java

- **Eingebautes Sprachfeature** mit speziellem operator[] / length-Attribut
- Enthält konkrete Werte (int,...) oder Referenzen auf Objekte im Speicher
- **Beispiel:** `int[] x = {1, 1, 2, 3, 5, 8}; int x2 = x[2]; int len = x.length; // 6`

C++

- **Syntactic Sugar:** Array = Pointer auf zusammenhängenden Speicherbereich
- **Problem:** Längeninformation werden nicht explizit gespeichert
- **Gefahr:** Keine Bereichsprüfung
- **Beispiel:**

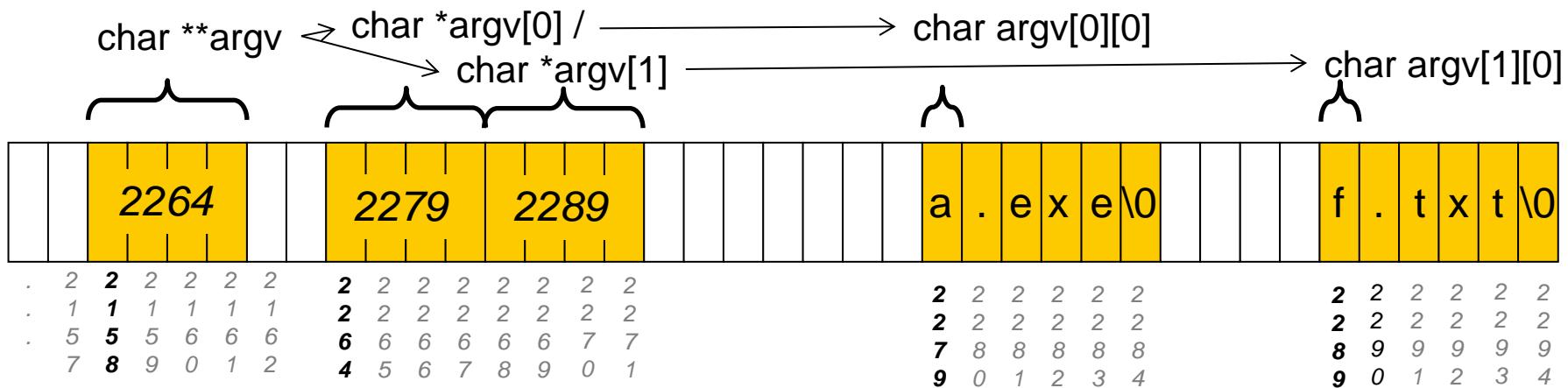
```
int myArray[] = {1, 1, 2, 3, 5, 8};  
int *myArray2 = myArray;  
int x2 = myArray[2];  
int x77 = *(myArray + 77); // Danger!
```

<http://www.cplusplus.com/doc/tutorial/arrays/>

int main(int argc, char** argv)



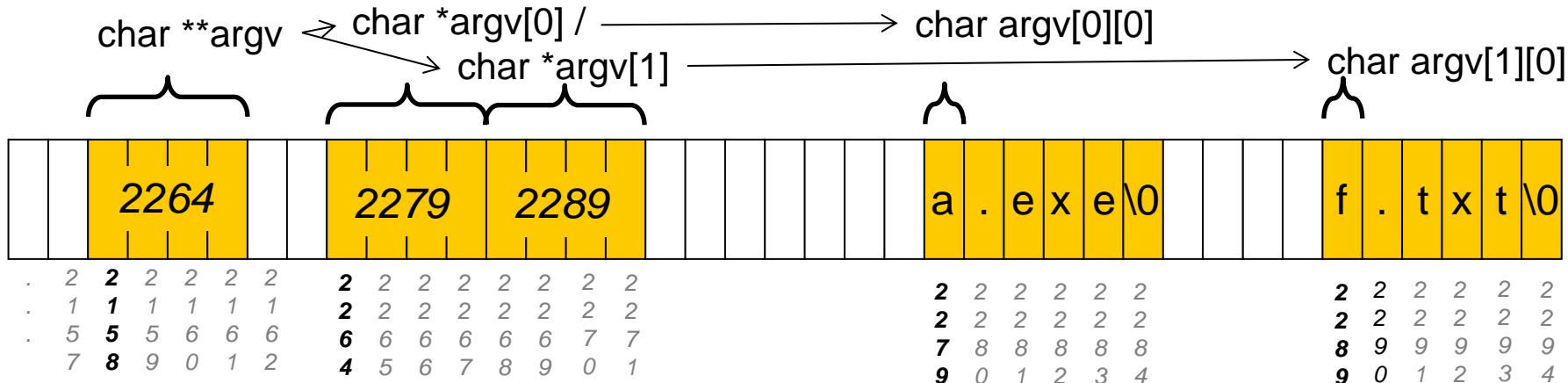
- Was passiert beim Aufruf `a.exe f.txt`?
- Traditionelle Strings: Folgen von char (mit '\0' abgeschlossen)



int main(int argc, char** argv)



- Was passiert beim Aufruf `a.exe f.txt`?
- Traditionelle Strings: Folgen von char (mit '\0' abgeschlossen)



```
cout << argv      << endl;
cout << argv[0]    << endl;
cout << argv[1]    << endl;
```

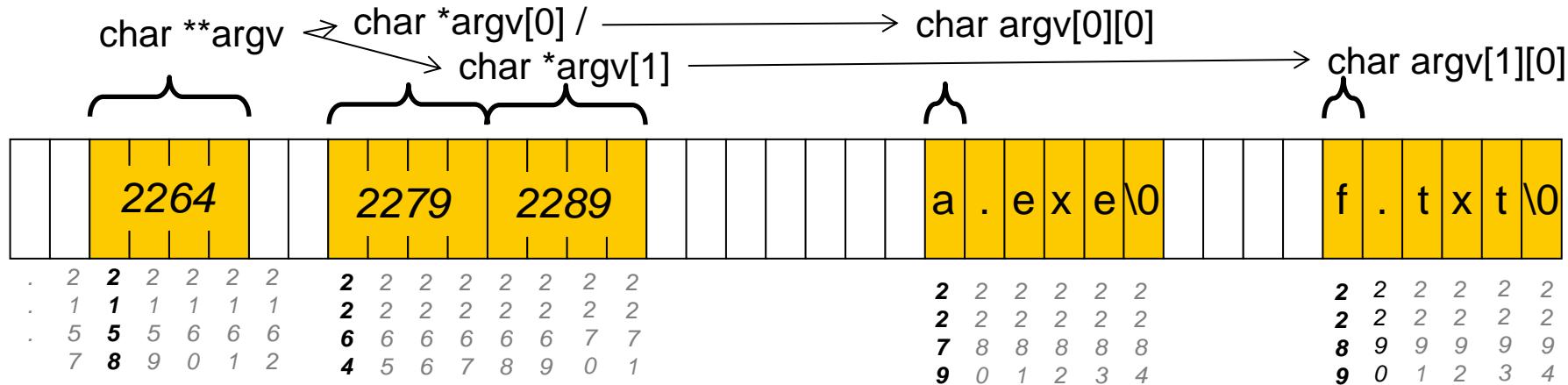


2264
a.exe 2279
f.txt 2289

int main(int argc, char** argv)



- Was passiert beim Aufruf `a.exe f.txt`?
- Traditionelle Strings: Folgen von char (mit '\0' abgeschlossen)



```
cout << argv      << endl;
cout << argv[0]    << endl;
cout << argv[1]    << endl;
```



```
2264
a.exe 2279
f.txt 2289
```

Spezieller operator<<
für char*

```
cout << (void *)argv[0] << endl
```

2279

void* =
"Generischer" Pointer

sizeof-Operator und std::size_t



▪ sizeof-Operator

- ... liefert die **Größe (in Byte) einer Variable** eines bestimmten Typs.
- Aufruf über **Typ** oder **konkrete Variable** möglich

▪ Datentyp std::size_t

- Standard-STL-Datentyp, um **Objektgrößen** in Byte zu speichern
- Ist immer groß genug, um das größtmögliche Objekt auf der jeweiligen Plattform zu speichern.

```
#include <iostream>
// #include <cstddef> // contains std::size_t

int main() {
    // Via type
    std::cout << sizeof(double) << std::endl;
    // Via variable
    double x = 1.0;
    std::size_t y = sizeof(x);
    std::cout << y << std::endl;
}
```

<http://en.cppreference.com/w/cpp/language/sizeof>
http://en.cppreference.com/w/cpp/types/size_t

Unveränderlichkeit - **const**

- Das Schlüsselwort **const** deklariert eine **Variable** als unveränderlich.
- Das bedeutet, dass die zur Variablen gehörige Speicherzelle über die Variable nicht verändert werden kann.

const int i = 42;

i = 7; X

Unveränderlichkeit - *const*

Zeiger auf Konstante

vs.

Unveränderlicher Zeiger

|
|
|
|
|
|
|

`int i = 42;`

`const int *iP;`

`iP = &i; ✓`

`(*iP)++; X`

Unveränderlichkeit - *const*



Zeiger auf Konstante

vs.

Unveränderlicher Zeiger

```
int i = 42;
```

```
const int *iP;
```

```
iP = &i; ✓
```

```
(*iP)++; ✗
```

```
|  
| int i;  
| int j = 7;  
|  
| int *const jP = &j;  
|  
| (*jP)++; ✓  
|  
| jP = &i; ✗
```

Einmalige,
sofortige
Definition

Unveränderlichkeit - *const*



Zeiger auf Konstante

vs.

Unveränderlicher Zeiger

```
int i = 42;
```

```
const int *iP;
```

```
iP = &i; ✓
```

```
(*iP)++; ✗
```

```
|  
| int i;  
| int j = 7;  
|  
| int *const jP = &j;  
|  
| (*jP)++; ✓  
|  
| jP = &i; ✗
```

Einmalige,
sofortige
Definition

Unveränderlicher Zeiger auf Konstante:

```
int i = 42;  
const int *const iP = &i;
```

Eselnbrücke:

- *const* bezieht sich immer auf das "Nächstliegende".
- Lese von rechts nach links.



Ein Programm ist "**const correct**", wenn als unverändlich gekennzeichnete Objekte durch das Programm nicht verändert werden.

- Wird in C++ durch das **Schlüsselwort const** (für Typen und Funktionen) sichergestellt.
- `const int` und `int` entsprechen **zur Compile-Zeit** verschiedenen Typen, **zur Laufzeit** jedoch wird kein Unterschied gemacht

<https://isocpp.org/wiki/faq/const-correctness>

Was ist eine C++-Referenz?



Eine Referenz ist ein Alias auf eine Variable

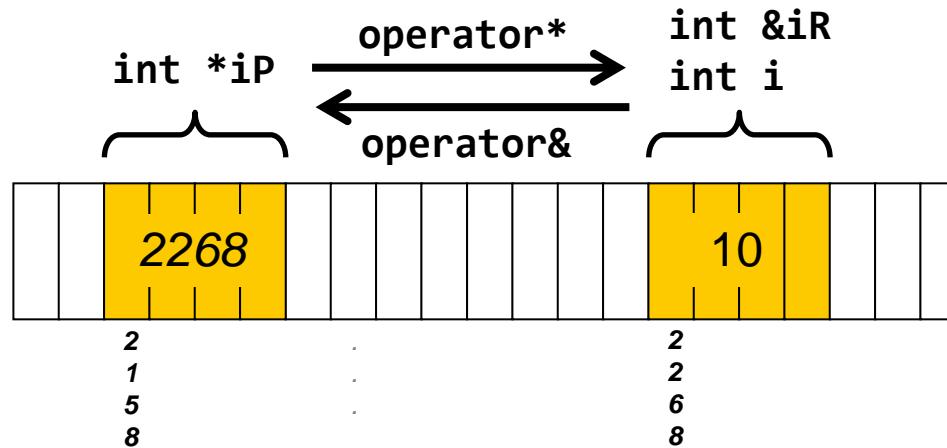
- Sie braucht **nicht zwangsweise eigenen Speicher** (bspw. innerhalb einer Funktion)
- Sie verhält sich **wie ein const-Pointer**.

```
int i = 42;  
  
int *const iP = &i;  
  
(*iP)++;  
  
const int *const iP = &i;  
  
cout << *iP << endl;
```

```
int i = 42;  
  
int &iR = i;  
  
iR++;  
  
const int &iR = i;  
  
cout << iR << endl;
```

Syntax wie
für Variablen

Zusammenfassung: Asterisk und Ampersand



	Asterisk (*)	Ampersand (&)
Typ	<code>int *iP = 2268;</code>	<code>int &iR = i;</code>
Operator	<code>// operator*</code> <code>if(*iP == 10){}</code>	<code>// operator&</code> <code>if(&i == iP){}</code>

Zusammenfassung: Variabtentypen

- **Werttypen** (enden weder auf &, *, [],)

 - Variablen mit **Werttyp X** repräsentieren konkrete Werte/Objekte
 - z.B. int x = 3; Building b = Building(3);

- **Referenztypen** (enden auf &)

 - Variable mit **Referenz-Typ X&** ist ein Alias für ein Objekt vom Typ X.
 - z.B. int &y = x;

- **Pointer-Typen** (enden auf *)

 - Variable mit **Pointer-Typ X*** verweist auf eine Speicherstelle, die einen X-Wert/-Objekt enthält.
 - z.B. int *x = new int(3);

- **Array-Typen** (enden auf [], Syntactic Sugar)

 - Variable mit **Array-Typ X[]** verweist auf ein Array, dessen Elemente den Typ X haben, und ist äquivalent zu Typ X*.
 - z.B. int x[] = {1,1,2,3,5}; int *x2 = x;

Zusammenfassung: Zuweisung

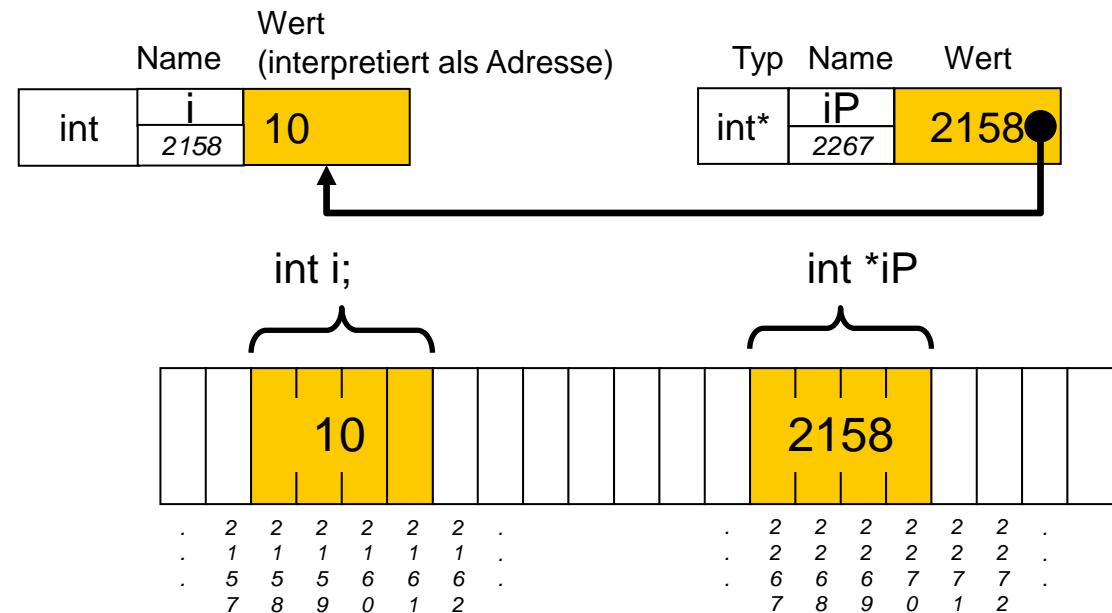


- Was passiert bei der Zuweisung zwischen verschiedenen Variabtentypen?
- [LHS-Typ] $x = [\text{Operator}] [\text{RHS-Typ}] y;$

LHS	RHS	Wert-Typ $Y\ y$	Referenz-Typ $Y\ \&y$	Pointer-Typ $Y\ *y$
Wert-Typ $X\ x =$	$x = y$ (Kopie)	$x = y$ (Kopie)	$x = *y$ (Kopie)	
Referenz-Typ $X\ \&x =$	$x = y$ (Alias)	$x = y$ (Alias)	$x = *y$ (Alias)	
Pointer-Typ $X\ *x =$	$x = \&(1)y$ (Alias)	$x = \&y$ (Alias)	$X = y$ (Alias)	

(1) **Adressoperator** kann nur auf "benannte" Objekte angewandt werden (z.B. Variablen), nicht aber auf anonyme Objekte und Literale (z.B. `&42`, `&Building()`)

Zusammenfassung: Rolle von const



const int	*	iP = &i;	CONST
int const	*	iP = &i;	CONST

int	* const iP = &i;	CONST
-----	------------------	-------

const int	* const iP = &i;	CONST
int const	* const iP = &i;	CONST

const bei Objekten



```
class Building {  
public:  
    Building(int number_of_floors);  
    ~Building();  
  
    void printFloorPlan() const;  
  
private:  
    std::vector<Floor> floors;  
    Elevator elevator;  
};  
  
void iDoNotChangeAnything(const Building &building) {  
    building.printFloorPlan();  
}
```

Verändert den Zustand des
Objekts nicht
(Read-only-Zugriff)

building darf
nicht verändert
werden

Es dürfen **nur const Methoden** auf
building aufgerufen werden

const Overloading



- Überladung von Methoden anhand von **const** ist möglich
- Typischerweise ähnliche oder identische Implementierung

```
class Building {  
public:  
    std::vector<Floor>& getFloors() { return floors; };  
    const std::vector<Floor>& getFloors() const { return floors; };  
private:  
    std::vector<Floor> floors;  
};  
  
int main() {  
    const Building b{};  
    const std::vector<Floor> &fs = b.getFloors();  
    const Floor &f = b.getFloors().at(1);  
}
```

Auch die Elemente des Vektors sind const!

Der this-Zeiger



- **this** ist in **jeder Methode** implizit verfügbar – wie in Java.
- Für eine Klasse C ist der **Typ** von **this**
 - **C*** innerhalb von **nicht-const** Methoden
 - **const C*** innerhalb von **const** Methoden
- **this** kann genutzt werden, um **Code "sprechender"** zu machen.

```
class Building {           Building.hpp
public:
    Building(int number_of_floors);
    ~Building();

    void printFloorPlan() const;

private:
    std::vector<Floor> floors;
    Elevator elevator;
};
```

```
void
Building::printFloorPlan()
const {
    /*...*/
    floors.at(0);
    this->floors.at(0);

    // Same for methods:
    printFloorPlan();
    this->printFloorPlan();
}
```

→ entspricht
(*this).floors.at(0)

Zusammenfassung: Vorteile von const?

- **Compiler** kann automatisch die **Absichten des Programmierers** statisch durchsetzen (es gibt einen guten Grund wieso etwas const sein soll!)
- Compiler kann **ggf. Optimierungen durchführen** mit dem Wissen darüber, was const ist und was nicht (diskutabel...)
- **Leser des Programmcodes** kann **Absichten des Programmierers** besser erkennen.

Programmierpraktikum C und C++



Speicherverwaltung und Lebenszyklus – Teil 2

(Übungsblatt: [S])



Dr.-Ing. Eric Lenz
elenz@iat.tu-darmstadt.de

Fachgebiet Control and Cyber-Physical Systems (CCPS)

Prof. Dr.-Ing. Rolf Findeisen
Dept. of Electrical Engineering and Information Technology
<https://www.ccps.tu-darmstadt.de/>



(Kopier-)Konstruktor, Zuweisung und Destruktor

AUF- UND ABBAUEN VON OBJEKTEN

Konstruktor, Destruktor und Copy-Konstruktor



```
class Floor {  
public:  
    Floor(std::string label,  
          int number);  
    ~Floor();  
    Floor(const Floor &floor);  
  
private:  
    const std::string label;  
    const int number;  
};
```

Konstruktor mit Initialisierungsliste
(Reihenfolge beachten!)

Copy-Konstruktor

Destruktor

```
Floor::Floor(  
    std::string label, int number):  
    label(label),  
    number(number) {  
    cout << "Creating floor"  
        << number << "]" << endl;  
}  
  
Floor::Floor(const Floor &floor):  
    label(floor.label),  
    number(floor.number+1) {  
    cout << "Copying floor"  
        << floor.number << "]" << endl;  
}  
  
Floor::~Floor() {  
    cout << "Destroying floor ["  
        << number << "]" << endl;  
}
```



Initialisierungslisten

- Initialisierungslisten haben mit C++11 eine **zweite Bedeutung** erhalten: Mittels Array-ähnlicher Syntax können jetzt **Datenstrukturen leichter initialisiert werden.**
- **Klassisch:** Pflicht bei const-Attributen und Referenzen im Konstruktor
 - `Floor::Floor(std::string label, int number)
: label(label), number(number) {}`
- **In C++11: "..." als Syntactic Sugar für std::initializer_list** zur vereinfachten Initialisierung von Vektoren etc. Beispiele:
 - `std::vector<int> v = std::vector<int>({7, 5, 16, 8});`
 - `std::vector<int> v({7, 5, 16, 8});`
 - `std::vector<int> v = {7, 5, 16, 8};`

impliziter Konstruktoraufruf

Klassisch: http://en.cppreference.com/w/cpp/language/initializer_list
std::initializer_list: http://en.cppreference.com/w/cpp/utility/initializer_list

Implizite Typ-Konvertierung und Anonyme Objekte



```
#include <string>

class Student {
public:
    Student(const std::string &name)
        : name(name) {}

private:
    std::string name;
};

int main() {
    Student mike("Mike");
}
```

Konstruktor erwartet std::string

Aber: Aufrufer verwendet const char*

Implizite Typkonvertierung, da std::string einen Konstruktor besitzt, der const char* als Parameter hat.

Das generierte Objekt ist "anonym", d.h. es kann nach dieser Zeile nicht mehr verwendet werden.

Implizite Typkonvertierung unterbinden mit `explicit`



```
#include <string>

class Student {
public:
    explicit Student(const std::string &name)
        : name(name) {}

private:
    std::string name;
};

int main() {
    Student mike("Mike");
    Student anna = std::string("Anna");
}
```

Schlüsselwort `explicit` unterbindet Verwendung des Konstr. für implizite Typkonvertierung

Dieser Aufruf wäre so nicht möglich, da die implizite Typkonvertierung ausgeschlossen wurde!



Java

- Man kann innerhalb eines Konstruktors an einen anderen Konstruktor delegieren (bspw. Default-Werte übergeben)
- **Beispiel:** `public Floor() { this("default", 1); }`

C++

- **Vor C++11:** man kann/muss nur Basisklassen initialisieren
 - `Class(): Base("default") {}`
 - Kann aber nicht an Konstruktoren der eigenen Klasse delegieren.
- **Seit C++11:** Konstruktoraufruf auf eigene Klasse möglich
 - `Floor() : Floor("default", 1) {}`

http://en.cppreference.com/w/cpp/language/initializer_list#Delegating_constructor

Parameterübergabe bei Methodenaufrufen



Parameter werden in C++ **immer** per Wert übergeben (**Call by Value**)

```
void iUseACopy(Floor floor){  
    cout << "This is floor ["  
        << floor.getNumber()  
        << "]" << endl;  
}  
  
int main() {  
    Floor floor("f", 0);  
    iUseACopy(floor);  
}
```

Copy-Konstruktor wird bei der Übergabe aufgerufen, um das Objekt zu kopieren!



Creating floor [0]

Copying floor [0]

This is floor [1]
Destroying floor [1]

Destroying floor [0]

Objekt wird automatisch zerstört wenn *iUseACopy* zu *main* zurückkehrt...

Parameterübergabe bei Methodenaufrufen (I)



Kopieren bei der Übergabe ist oft nicht gewollt. Lösungsmöglichkeiten:

(1) Übergabe "per Referenz" (**Call by Reference**)

```
void iUseAReference(  
    Floor &floor) {  
    cout << "This is floor ["  
        << floor.getNumber()  
        << "]"  
        << endl;  
}  
  
int main() {  
    Floor floor("f", 0);  
    iUseAReference(floor);  
}
```

Es wird **keine Kopie** des Objekts angelegt

Creating floor [0]
This is floor [0]
Destroying floor [0]

! *iUseAReference kann aber das Objekt beliebig verändern!*

Parameterübergabe bei Methodenaufrufen (II)



Kopieren bei der Übergabe ist oft nicht gewollt. Lösungsmöglichkeiten:
(2) Übergabe per **const Referenz**

```
void iUseAConstReference(  
    const Floor &floor){  
    cout << "This is floor ["  
        << floor.getNumber()  
        << "]"  
        << endl;  
}  
  
int main() {  
    Floor floor("f", 0);  
    iUseAConstReference(floor);  
}
```

Creating floor [0]
This is floor [0]
Destroying floor [0]

! Dies sollte grundsätzlich die Default-Übergabestrategie sein.

Parameterübergabe bei Methodenaufrufen (III)



Kopieren bei der Übergabe ist oft nicht gewollt. Lösungsmöglichkeiten:
(3) Übergabe per **Zeiger**

```
void iUseAPointer(  
    Floor *floor){  
    cout << "This is floor ["  
        << floor->getNumber()  
        << "]"  
        << endl;  
}  
  
int main() {  
    Floor floor("f", 0);  
    iUseAPointer(&floor);  
}
```

Äquivalent zu
(*floor).getNumber()

Creating floor [0]
This is floor [0]
Destroying floor [0]



Assignment-Operator (I)

- Neben dem Kopierkonstruktor gibt es auch noch eine andere Art, den **Zustand eines Objektes zu übertragen**: den **Assignment-Operator**

```
class EnergyMinimizingStrategy {
public:
    EnergyMinimizingStrategy() {
        std::cout << "Constructor called" << std::endl;
    }

    EnergyMinimizingStrategy(const EnergyMinimizingStrategy &a) {
        std::cout << "Copy constructor called" << std::endl;
    }
}
```

- ! **Copy-Konstruktor** überträgt Zustand beim Initialisieren
- ! **Assignment-Operator** überträgt Zustand nach dem Initialisieren

```
EnergyMinimizingStrategy &operator=(const EnergyMinimizingStrategy &a) {
    std::cout << "operator= called" << std::endl;
    return *this;
};
```

Rückgabe von *this erlaubt Verkettung ("Operator chaining"):
EMS e1,e2,e3; e1 = e2 = e3; // same as: e1 = (e2 = e3);

Assignment-Operator: Vergleich zu Java



- Assignment-Operator kann in Java **nicht überschrieben/angepasst werden.**

- **Java-Primitive** (int, double,...): **Wertzuweisung**

```
int x = 1;  
int y = x; // -> copy  
++y // Only y is modified
```

- **Java-Objekte:** **Referenzzuweisung/Aliasing**

```
Floor x = new Floor();  
Floor y = x; // -> alias  
y.setLevel(3); // x and y are modified
```

Compiler-generierte Methoden: C++



- Der C++-Compiler ("automagically") generiert automatisch eine Reihe von Methoden, falls sie **nicht vorhanden (=deklariert)** sind, z.B.:
 - Default-Konstruktor `MyClass ()` (←wie in Java!)
 - Copy-Konstruktor `MyClass (const MyClass &a)`
 - Assignment-Operator `MyClass &operator=(const MyClass &a)`
 - Destruktor `~MyClass ()`
 - Initialisierungsliste → Default-Konstruktoren für Felder
- Man kann auch die **Generierung unterbinden**
 - vor C++11: `MyClass &operator=(MyClass &); // WITHOUT implementation`
 - seit C++11: `MyClass &operator=(MyClass &) = delete;`
- **Gegenstück: "= default"**, falls man trotz (bspw.) manuell implementiertem Kopierkonstruktor die Standardimplementierung (bspw.) des parameterlosen Konstruktors haben will.
 - `MyClass() = default;`

**Wichtig als Zeichen
an andere Entwickler!**

https://en.wikipedia.org/wiki/C%2B%2B11#Explicitly_defaulted_special_member_functions



Java

- **Struktur:** try + mehrere catch-Blöcke; **Catch by reference**
- Nur **Untertypen** von `java.lang.Exception` können geworfen/gefangen werden.
- **Default:** `catch(Exception e)`

C++

- **Struktur:** try + mehrere catch-Blöcke; **Throw by-value + catch by-reference**
- **Jeglicher Datentyp** kann geworfen werden, z.B. `throw 10`, `throw "Hello!"`
- **Default:** `catch(...)`

```
// #include <stdexcept>
try {
    throw 10;
    throw std::runtime_error("Some problem occurred");
} catch (std::exception &e) {
    std::cout << "Standard exception: " << e.what() << std::endl;
    throw; // Rethrow e
} catch (...) { std::cout << "Object/value caught" << std::endl; }
```

<http://www.cplusplus.com/doc/tutorial/exceptions/>



Hängende Zeiger und Speicherlecks

STOLPERFALLEN BEI DER SPEICHERVERWALTUNG

http://static.tvtropes.org/pmwiki/pub/images/Bear_Trap_7423.jpg

Hängende Zeiger

Referenzen auf gelöschte Objekte zurückgeben



```
Floor &makeNextFloor(const Floor &floor){  
    Floor next = Floor(floor);  
    cout << "Making next floor [ "  
        << next.getNumber()  
        << "]" << endl;  
    return next;  
}
```

```
int main() {  
    Floor floor(0);  
    Floor &next = makeNextFloor(floor);  
    cout << "Next floor is floor [ "  
        << next.getNumber()  
        << "]" << endl;  
}
```

Hier wird eine Referenz
auf eine lokale Variable
zurückgegeben!

g++ ist gnädig und lässt das mit einer
Warnung durchgehen. Ist trotzdem
sehr schlechter Programmierstil!



Creating floor [0]

Copying floor [0]

Making next floor[1]

Destroying floor [1]

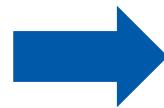
Next floor is floor [1]

Destroying floor [0]

Rückgabe von Objekten durch Kopieren



```
Floor makeNextFloor(const Floor &floor){  
    Floor next = Floor(floor);  
    cout << "Made next floor ["  
        << next.getNumber()  
        << "]"  
        << endl;  
    return next;  
}  
  
int main() {  
    Floor floor(0);  
  
    Floor nextFloor = makeNextFloor(floor);  
  
    cout << "Next floor is floor ["  
        << nextFloor.getNumber()  
        << "]"  
        << endl;  
}
```



Creating floor [0]

Copying floor [0]
Made next floor [1]
Copying floor [1]
Destroying floor [1]

Next floor is floor [2]
Destroying floor [2]
Destroying floor [0]

Erwartet

Rückgabe von Objekten durch Kopieren



```
Floor makeNextFloor(const Floor &floor){  
    Floor next = Floor(floor);  
    cout << "Made next floor ["  
        << next.getNumber()  
        << "]"  
        << endl;  
    return next;  
}  
  
int main() {  
    Floor floor(0);  
  
    Floor nextFloor = makeNextFloor(floor);  
  
    cout << "Next floor is floor ["  
        << nextFloor.getNumber()  
        << "]"  
        << endl;  
}
```

Compiler erkennt, wann Kopien
vermieden werden können



Creating floor [0]

Copying floor [0]
Made next floor [1]
Copying floor [1]
Destroying floor [1]

Next floor is floor [2]
Destroying floor [2]
Destroying floor [0]

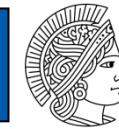
Creating floor [0]

Copying floor [0]
Made next floor [1]

Next floor is floor [1]
Destroying floor [1]
Destroying floor [0]

Erwartet

Tatsächlich

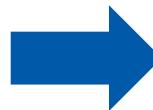


```
class EnergyMinimizingStrategy {
public:
    EnergyMinimizingStrategy() {
        cout << "Constructor called" << endl;
    }

    EnergyMinimizingStrategy(const
        EnergyMinimizingStrategy &a) {
        cout << "Copy constructor called" << endl;
    }

    EnergyMinimizingStrategy &operator=
        (const EnergyMinimizingStrategy &a) {
        cout << "operator= called" << endl;
        return *this;
    }

int main() {
/*1.*/ EnergyMinimizingStrategy a;
/*2.*/ EnergyMinimizingStrategy c = a;
/*3.*/ EnergyMinimizingStrategy b(a);
/*4.*/ b = a;
/*5.*/ EnergyMinimizingStrategy d =
    EnergyMinimizingStrategy();
}
```



Ausgabe:

- 1 Constructor called
- 2 Copy constructor called
- 3 Copy constructor called
- 4 operator= called
- 5 Constructor called

Mit *-fno-elide-constructors* wird tatsächlich kopiert.

- Zu erwarten ist, dass bei (5.) zunächst ein Objekt mittels Default-Konstruktor angelegt und dann mittels *operator=* überschrieben wird – C++ ist da schlauer ☺.

https://en.wikipedia.org/wiki/Copy_elision

Rückgabe von Objekten auf dem Heap



```
Floor* makeNextFloor(const Floor &floor){  
    Floor *next = new Floor(floor);  
    cout << "Made next floor ["  
        << next->getNumber() << "]"  
        << endl;  
    return next;  
}  
  
int main() {  
    Floor floor(0);  
  
    Floor *nextFloor = makeNextFloor(floor);  
  
    cout << "Next floor is floor ["  
        << nextFloor->getNumber()  
        << "]" << endl;  
}
```



Creating floor [0]

Copying floor [0]
Made next floor [1]

Next floor is floor [1]
Destroying floor [0]

Dieses Programm enthält einen
Fehler: Memory Leak!

Rückgabe von Objekten auf dem Heap



```
Floor* makeNextFloor(const Floor &floor){  
    Floor *next = new Floor(floor);  
    cout << "Made next floor ["  
        << next->getNumber() << "]"  
        << endl;  
    return next;  
}  
  
int main() {  
    Floor floor(0);  
  
    Floor *nextFloor = makeNextFloor(floor);  
  
    cout << "Next floor is floor ["  
        << nextFloor->getNumber()  
        << "]" << endl;  
  
    delete nextFloor;  
}
```



Creating floor [0]

Copying floor [0]
Made next floor [1]

Next floor is floor [1]
Destroying floor [1]
Destroying floor [0]

Hängende Zeiger

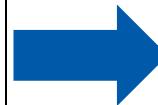
Frühzeitige Zerstörung von Objekten



```
int main() {
    Floor *floor = new Floor(0);
    Floor &refToFloor = *floor;

    delete floor;

    cout << "Dangling reference to floor ["
        << refToFloor.getNumber()
        << "]" << endl;
}
```



Creating floor [0]
Destroying floor [0]

Dangling reference to
floor:
[5444032]



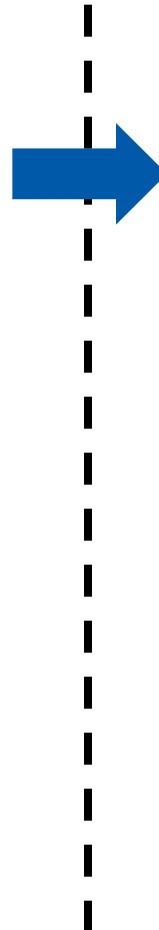
Extrem gefährlich!

Hängende Zeiger

Nochmalige Zerstörung von Objekten



```
int main() {  
    Floor *floor = new Floor(0);  
  
    delete floor;  
    delete floor;  
}
```



Creating floor [0]
Destroying floor [0]
Destroying floor [5903232]

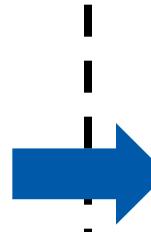
Extrem gefährlich!

Hängende Zeiger

Nochmalige Zerstörung von Objekten



```
int main() {  
    Floor *floor = new Floor(0);  
  
    delete floor;  
    delete floor;  
}
```



Creating floor [0]
Destroying floor [0]
Destroying floor [5903232]

Extrem gefährlich!

```
int main() {  
    Floor *floor = new Floor(0);  
  
    delete floor;  
  
    floor = nullptr;  
  
    delete floor;  
}
```



Creating floor [0]
Destroying floor [0]

Nach dem Löschen
immer auf nullptr
setzen!

Speicherlecks



```
int main() {  
    Floor *floor = new Floor(0);  
    Floor *otherFloor = new Floor(1);  
  
    otherFloor = floor; // -> floor [0]  
  
    delete floor;  
    delete otherFloor;  
}
```



Was wird hier
gelöscht?



Creating floor [0]
Creating floor [1]
Destroying floor [0]
Destroying floor [0]

Es ist nicht mehr möglich, *floor [1]*
freizugeben! Dies wird auch als
ein **Speicherleck** bezeichnet.

Verantwortlichkeitsprobleme bei Zeigern



```
int f(const Floor &floor) {
    // (1) Am I sure that floor is not
    //      already a dangling reference?

    // Use floor in some way

    // (2) Is floor on the heap?
    // (3) Am I supposed to delete it or not?
    // (4) If yes, how about all other references
        to floor from other objects?
        How do these objects know that floor is now destroyed?
}
```

```
int g() {
    Floor *floorOnHeap = new Floor(0);
    Floor floorOnStack(1);

    // How do I signalise that floorOnHeap/floorOnStack should (not)
    // be deleted? Or that I want to give up "ownership" of floorOnHeap
    // (it should be deleted)?
    f(*floorOnHeap);
    f(floorOnStack);

    // I might still want to use floorOnHeap here!
}
```

Saubere Speicherverwaltung im Allgemeinen **nur mit vielen Konventionen** möglich.
Fremdbibliotheken können aber andere Konventionen verlangen.

Wie können wir (1) – (3) klären und vor allem (4) immer garantieren?

Aliasing bei klassischen Zeigern



```
Person *Eve = new Person();  
Person *Alice = Eve;
```

"Rohzeiger"
(raw pointer)

Objekt auf dem Heap

Eve



:Person

Alice

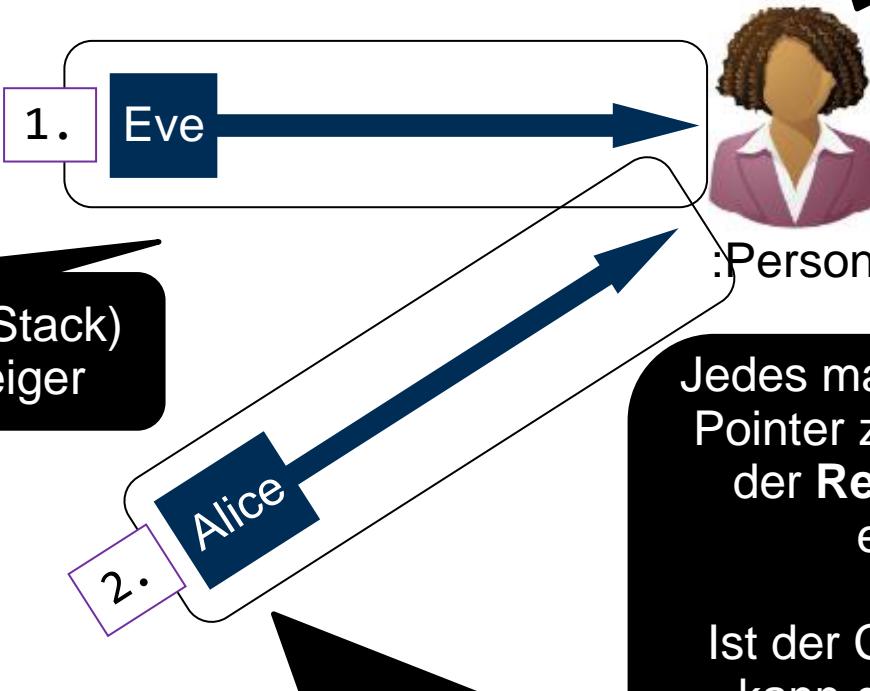
Die Person darf nur zerstört werden, wenn es keine Zeiger mehr gibt!

Mit std::shared_ptr



```
std::shared_ptr<Person> Eve(new Person());  
std::shared_ptr<Person> Alice = Eve;
```

Objekt auf dem Heap



Smart Pointer (auf dem Stack)
als **Wrapper** für Rohzeiger

Jedes mal wenn ein Smart
Pointer zerstört wird, wird
der **Referenzcounter**
erniedrigt.

Smart Pointer wissen, **wie oft**
das Objekt referenziert wird

Ist der Counter bei 0, so
kann das Objekt vom
Smart Pointer zerstört
werden!

Person – ohne std::shared_ptr



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

class Person {
public:
    Person(const string &name);
    Person(const Person &person);
    ~Person();

    const string &getName() const
    {
        return name;
    }

private:
    const string name;
};
```

Person.hpp

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

Person::Person(const string &name):
    name(name) {
    cout << endl << "Created " << name << endl;
}

Person::Person(const Person &person):
    name(person.name){
    cout << "Cloning " << name << endl;
}

Person::~Person() {
    cout << endl << "Good bye " << name << endl;
}
```

Person.cpp

Person – mit std::shared_ptr



```
#include <string>
#include <memory>
```

Person.hpp

```
class Person {

using namespace std;
public:
    Person(const string &name);
    Person(const Person &person);
    ~Person();

    const string &getName() const {
        return name;
    }

private:
    const string name;
};

typedef std::shared_ptr<Person>
PersonPtr;

typedef std::shared_ptr<const Person>
ConstPersonPtr;
```

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

Person.cpp

```
Person::Person(const string &name):
    name(name) {
    cout << "Created " << name << endl;
}

Person::Person(const Person &person):
    name(person.name){
    cout << "Cloning " << name << endl;
}

Person::~Person() {
    cout << "Good bye " << name << endl;
}
```

Beispiel: Weniger Code dank smarter Zeiger



```
#include "Person.hpp"

int main() {
    Person *eve(new Person("Eve"));
    greet(*eve);

    Person *alice = eve;
    greet(*alice);

    delete eve;
    eve = 0;
}
```

main.cpp

```
#include "Person.hpp"

int main() {
    ConstPersonPtr eve(new Person("Eve"));
    greet(eve);

    ConstPersonPtr alice = eve;
    greet(alice);

}
```

main.cpp



std::make_shared

```
#include <string>
#include <memory>

class Person {
public:
    Person(const std::string &name)
        : name(name) {}
private:
    std::string name;
};

int main() {
    std::shared_ptr<Person> nobody;

    Person *leila = new Person("Leila");
    delete leila;

    std::shared_ptr<Person> mike(
        new Person("Mike"));

    std::shared_ptr<Person> susan =
        std::make_shared<Person>("Susan");
}
```

std::shared_ptr()
entspricht nullptr.

Der Raw Pointer sollte **direkt** und
genau einmal in einen
std::shared_ptr eingepackt werden.

Die Utility-Funktion
std::make_shared ist vorteilhaft:
Exceptions führen nicht zu
Speicherfehlern und die
Speicherallokation ist schneller

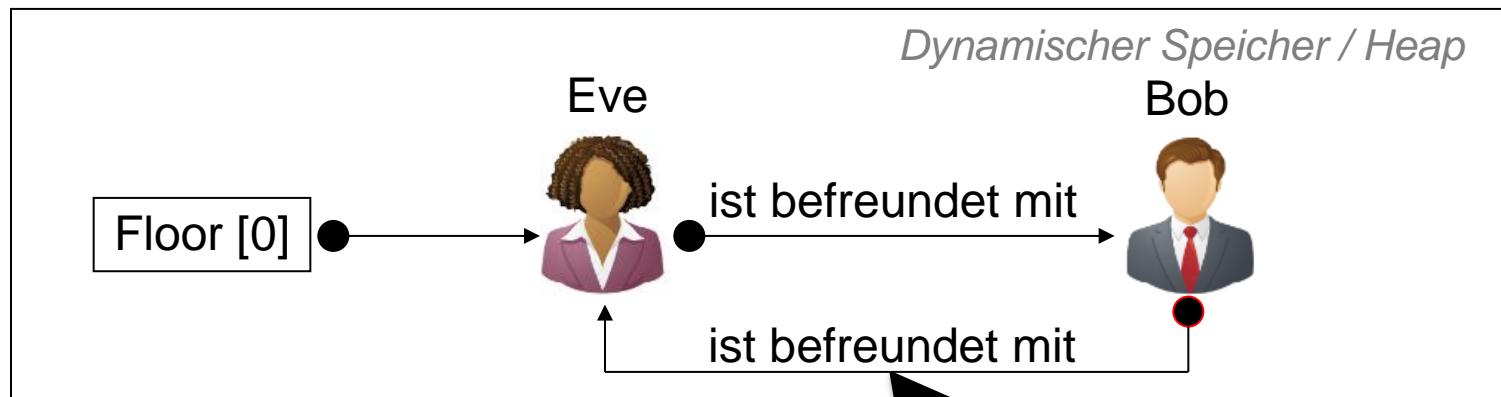
http://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared

Weak SmartPointer: Motivation



`std::shared_ptr<>` ist nicht perfekt:

- Etwas langsamer als Rohzeiger
- Erkennt **zirkuläre Abhängigkeiten** nicht



Weak SmartPointer: Motivation

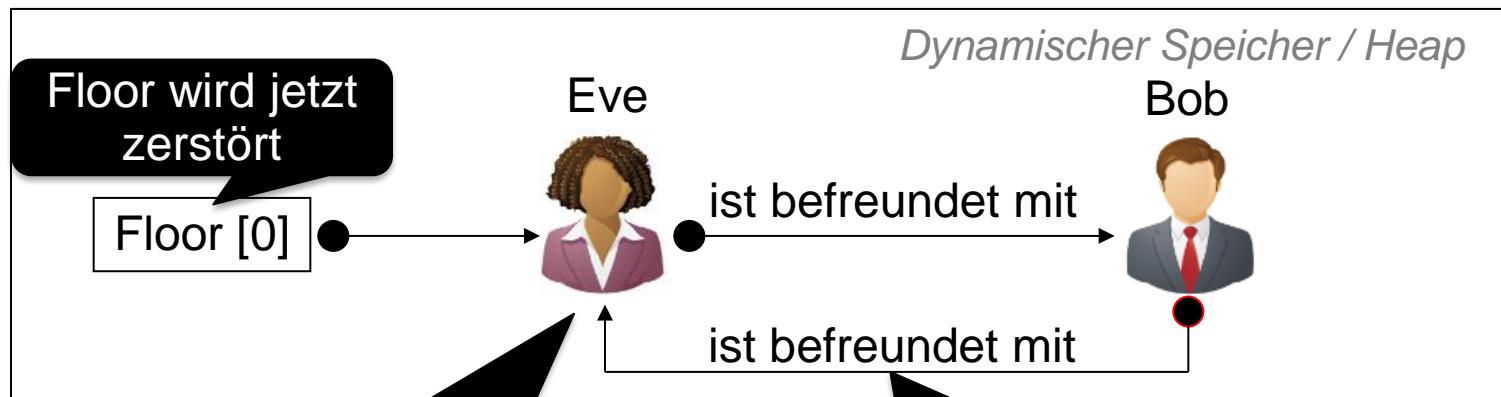


`std::shared_ptr<>` ist nicht perfekt:

- Etwas langsamer als Rohzeiger
- Erkennt **zirkuläre Abhängigkeiten** nicht:

Ablauf:

1. Objekt `Floor[0]` wird zerstört
2. Fertig – Eve und Bob halten sich gegenseitig am Leben.

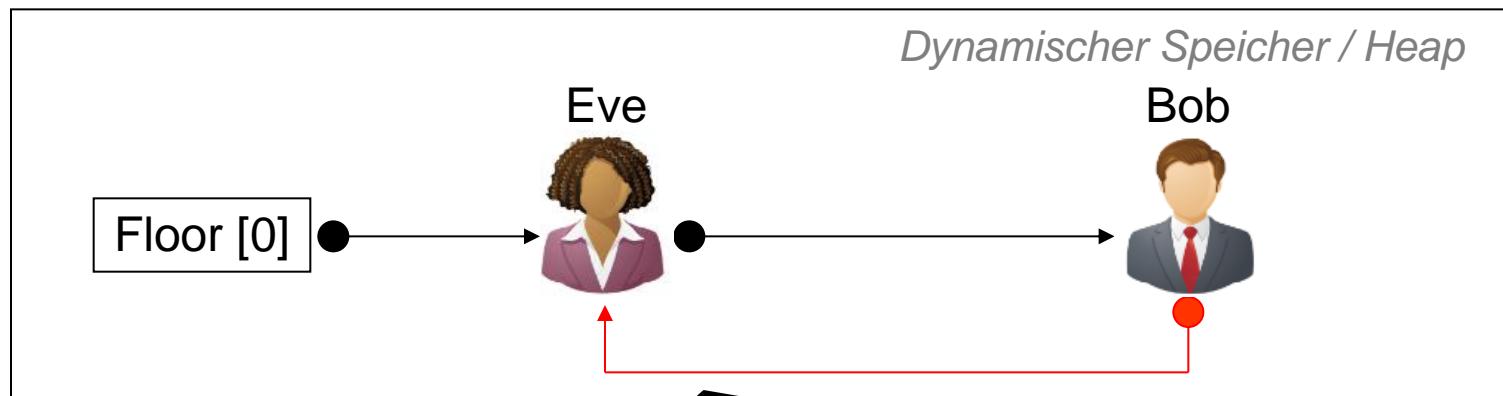


Weak Pointer (`std::weak_ptr`)



- `std::weak_ptr` für **eine Richtung der Beziehung** zwischen Personen verwenden (z.B.: Eve zeigt stark auf Bob, Bob schwach auf Eve)
- `std::shared_ptr` um "**extern**" auf Personen zu zeigen (Floor auf Person)
- Ein schwacher (weak) Zeiger verlangt, das **mindestens ein "starker" (strong) Zeiger** (z.B. ein `std::shared_ptr`) bereits auf die Person zeigt
- Person wird gelöscht, sobald **höchstens noch schwache Zeiger** darauf verweisen

Weak Pointer: Lösung

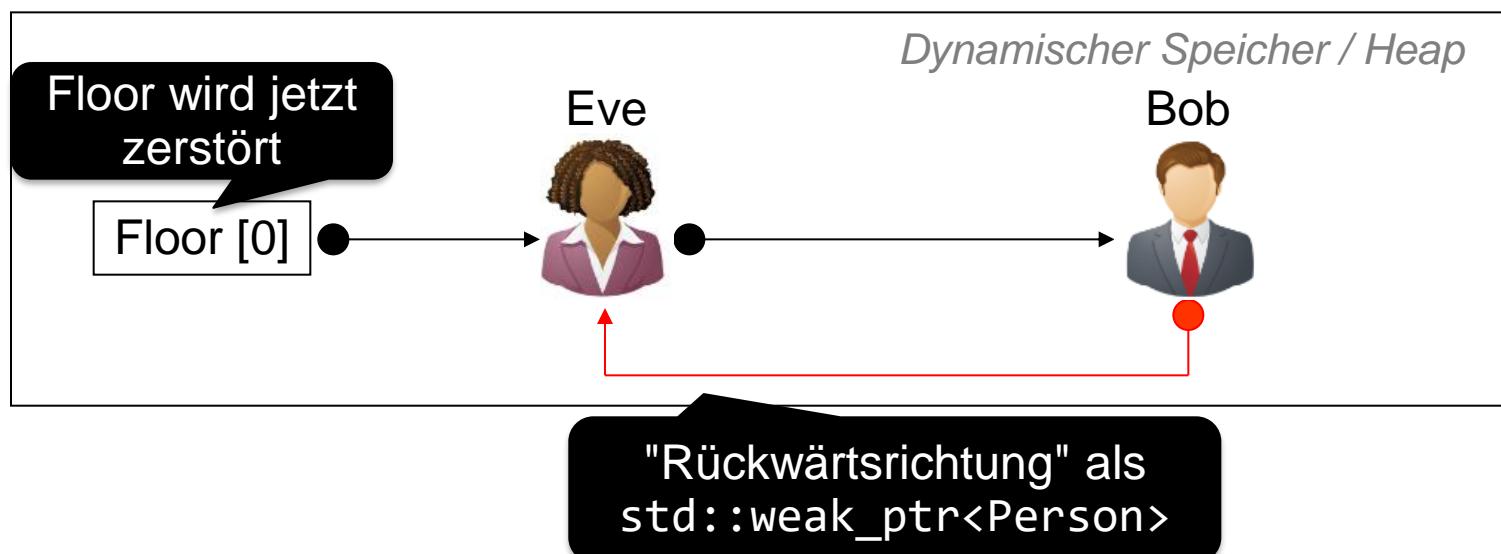


"Rückwärtsrichtung" als
`std::weak_ptr<Person>`

Weak Pointer: Lösung

Ablauf mit `std::weak_ptr<Person>`:

1. Objekt `Floor[0]` wird zerstört
2. Zähler: 0 Smart/ 1 Weak Pointer auf Eve
3. Eve wird zerstört
4. Zähler: 0 Smart/ 0 Weak Pointer auf Bob
5. Bob wird zerstört



Zusammenfassung: Übergabe und Rückgabe



Java

- **Keinerlei "Konfigurationsmöglichkeit"**
 - Primitive "by value" (d.h. int, double, ...)
 - Objekte "by reference" (d.h. Object, String, ...)
- **Übergabe:** Einzige Variation ist final oder nicht final
 - Auswirkung innerhalb der Methode (bzgl. Neuzuweisung)

C++

- **Alles konfigurierbar**, aber anspruchsvoller
- **Übergabe** unabhängig ob primitiver oder komplexer Datentyp
 - "pass/call by value"
 - "pass/call by reference (to const)"
 - "pass/call by pointer (to const)"
- **Rückgabe:**
 - "return by value" (sicher, aber Zusatzaufwand durch Kopie, evtl. Copy Elision)
 - "return by reference (to const)" (effizient, aber Gefahr von Speicherfehlern)
 - "return by pointer (to const)" (effizient, aber Gefahr von Speicherfehlern)

Programmierpraktikum C und C++



Objektorientierung

(Übungsblatt: [0])



Dr.-Ing. Eric Lenz
elenz@iat.tu-darmstadt.de

Fachgebiet Control and Cyber-Physical Systems (CCPS)

Prof. Dr.-Ing. Rolf Findeisen
Dept. of Electrical Engineering and Information Technology
<https://www.ccps.tu-darmstadt.de/>

Was ist (Untertyp-)Polymorphie?



- **Bedeutung:** Eine Variable kann Instanzen verschiedener Klassen enthalten (oder darauf verweisen), die eine Unterklasse des statischen Typs der Variable sind.
- **Beispiel:**

```
ElevatorStrategy *strategy = new EnergyMinimizingStrategy();      // (1)
strategy = new WaitingTimeMinimizingStrategy();                      // (2)
```

 - **Statischer Typ** (zur Compilezeit) von strategy:
ElevatorStrategy *
 - **Dynamischer Typ** (zur Laufzeit) von strategy:
(1) EnergyMinimizingStrategy*
(2) WaitingTimeMinimizingStragy*
- Funktioniert in C++ nur mit Pointern/Referenzen – nicht mit Werten!

[https://en.wikipedia.org/wiki/Polymorphism_\(computer_science\)](https://en.wikipedia.org/wiki/Polymorphism_(computer_science))

Ein einfaches Beispiel für Polymorphie in C++

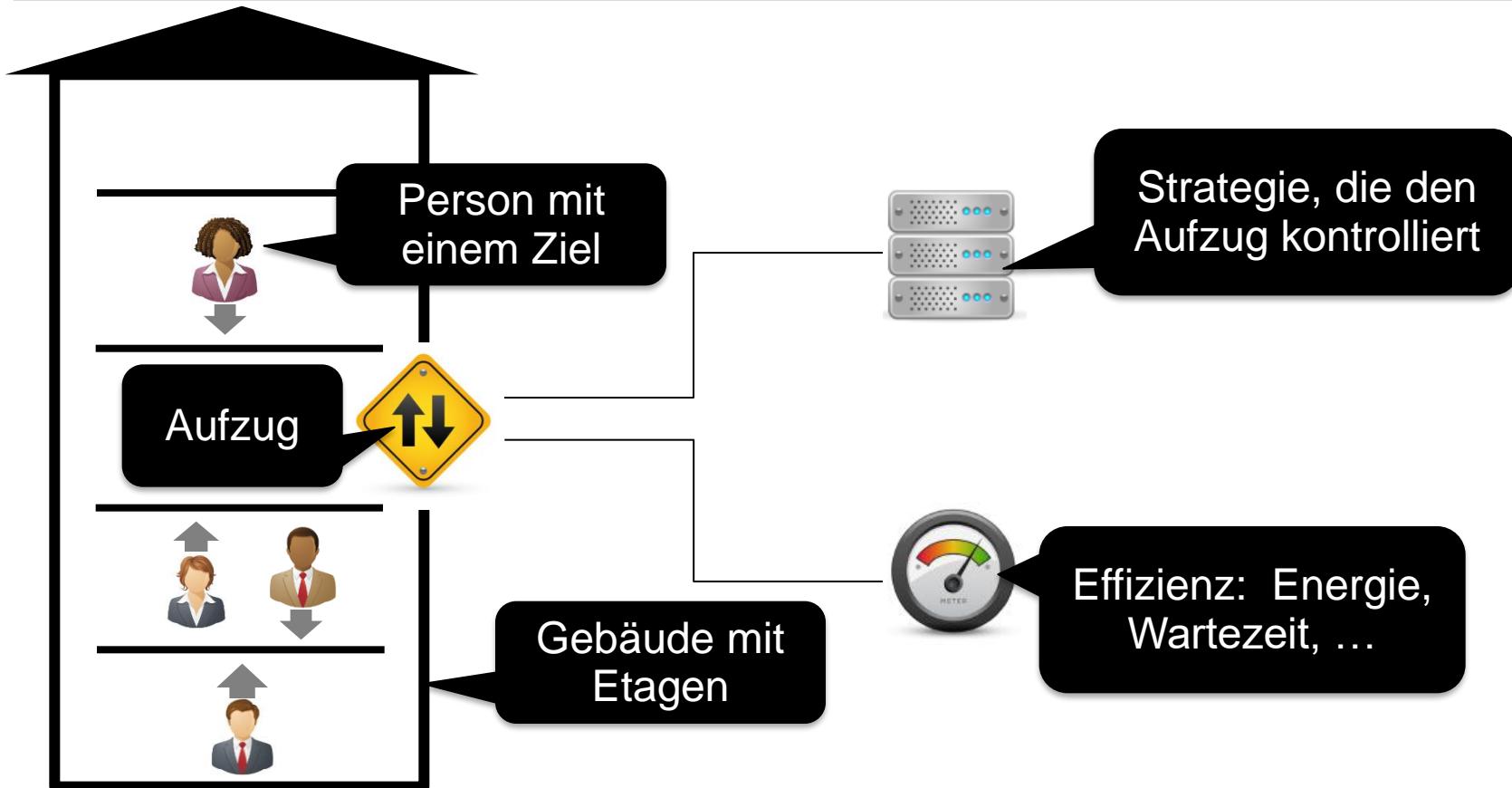


```
class Base {  
public:  
    virtual void print() {  
        std::cout << "B" << std::endl;  
    } };  
  
class Child : public Base{  
public:  
    virtual void print() override {  
        std::cout << "C" << std::endl;  
    } };  
  
void doPrint(Base b) { b.print(); }  
  
void doPrintRef(Base &b) { b.print(); }  
  
// ...
```

Polymorphie funktioniert in C++ nur mit Pointern und Referenzen

```
// ...  
  
int main() {  
    Base b;  
    Base baseFromChild = Child();  
    Child c;  
    Base *basePtrFromChild = new Child();  
  
    b.print(); // B  
    baseFromChild.print(); // B  
    c.print(); // C  
    basePtrFromChild->print(); // C  
  
    doPrint(b); // B  
    doPrint(baseFromChild); // B  
    doPrint(c); // B  
    doPrint(*basePtrFromChild); // B  
  
    doPrintRef(b); // B  
    doPrintRef(baseFromChild); // B  
    doPrintRef(c); // C  
    doPrintRef(*basePtrFromChild); // C  
  
}
```

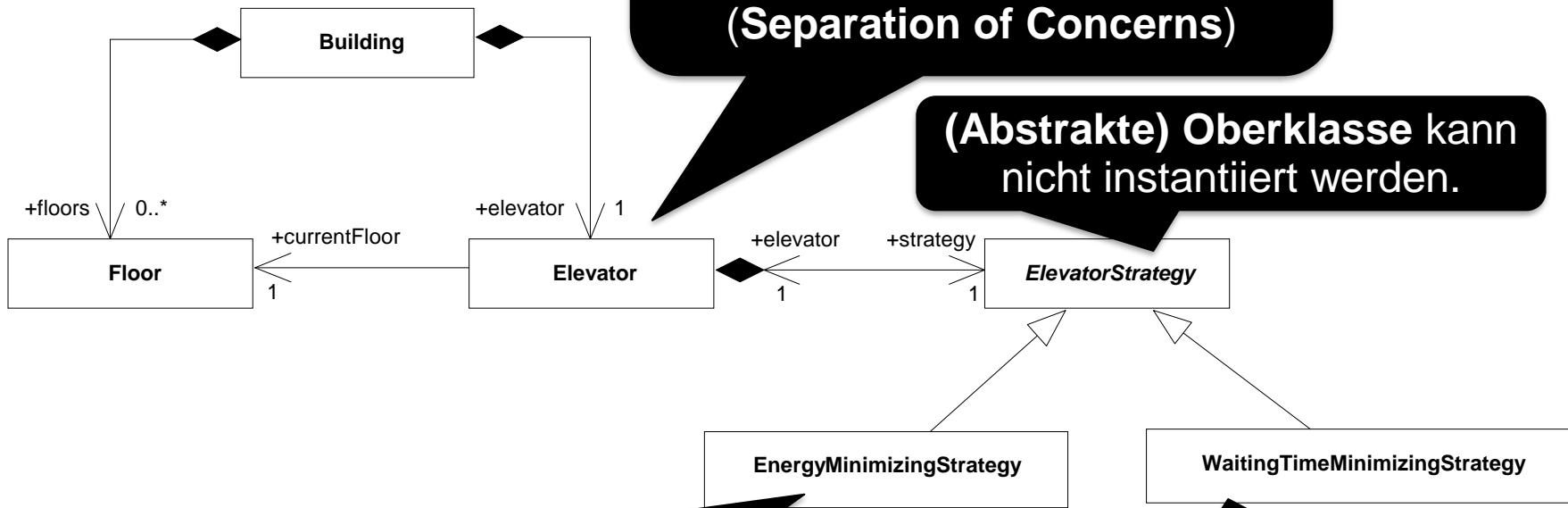
Wozu Polymorphie?



Verschiedene Strategien als Unterklassen



Der Code im Aufzug, der die Strategie verwendet, soll sich nicht ändern, nur weil eine andere Strategie eingesetzt wird.
(Separation of Concerns)



Unterschiedliche Strategien können ergänzt und verwendet werden (**Erweiterbarkeit**). Die richtige Methode wird "magisch" aufgerufen!

Konkrete Unterklassen

Ein Blick auf die Klassen ElevatorStrategy

In der .cpp-Datei ist dies
aber kein Problem!



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
#include <memory>          ElevatorStrategy.hpp
#include "Floor.hpp"

class Elevator;

class ElevatorStrategy {
public:
    ElevatorStrategy();
    ~ElevatorStrategy();

    const Floor*
    next(const Elevator *elevator) const;
};

typedef
std::shared_ptr<ElevatorStrategy>
ElevatorStrategyPtr;

typedef
std::shared_ptr<const ElevatorStrategy>
ConstElevatorStrategyPtr;
```

```
#include "ElevatorStrategy.hpp"
#include "Elevator.hpp"

using namespace std;

ElevatorStrategy::ElevatorStrategy() { /* ... */ }

ElevatorStrategy::~ElevatorStrategy() {/* ... */}

const Floor*
ElevatorStrategy::next(const Elevator *elevator) const
{ /* ... */ }
```

Forward Declaration (statt #include) um
zyklische Abhängigkeit zu vermeiden
Nur Referenzen oder Pointer auf die
referenzierte Klasse können genutzt werden

Ein Blick auf die Klassen Elevator



```
#include "ElevatorStrategy.hpp"  
#include "Floor.hpp"  
  
class Elevator {  
public:  
    Elevator(const Floor*,  
             ConstElevatorStrategyPtr);  
    ~Elevator();  
  
    inline const Floor* getCurrentFloor() const {  
        return currentFloor;  
    }  
  
    void moveToNextFloor();  
  
private:  
    const Floor *currentFloor;  
    ConstElevatorStrategyPtr strategy;  
};
```

Elevator.hpp

Parameter ohne
Namen möglich

```
#include <iostream>  
using std::cout;  
using std::endl;  
  
#include "Elevator.hpp"  
  
Elevator::Elevator(const Floor *currentFloor,  
                    ConstElevatorStrategyPtr  
strategy):  
    currentFloor(currentFloor), strategy(strategy) {  
    cout << "Elevator(): "  
        << "Creating elevator." << endl;  
}  
  
Elevator::~Elevator(){  
    cout << "~Elevator(): "  
        << "Destroying elevator." << endl;  
}  
  
void Elevator::moveToNextFloor(){  
    cout << "Elevator::moveToNextFloor(): "  
        << "Polymorphic call to strategy." << endl;  
  
    currentFloor = strategy->next(this);
```

Elevator.cpp

const Floor* und nicht **const Floor&**,
da der Zeiger sich ändert (aber nicht das
Objekt worauf gezeigt wird!)

Verwendung der Strategie bleibt
gleich, egal welche konkrete
Strategie verwendet wird

Sichtbarkeits-Modifier bei Vererbung



EnergyMinimizingElevatorStrategy.hpp

```
#include "ElevatorStrategy.hpp"

class EnergyMinimizingStrategy
    : public ElevatorStrategy {
public:
    EnergyMinimizingStrategy();
    ~EnergyMinimizingStrategy();

    const Floor*
    next(const Elevator *elevator) const;
};
```

public-Vererbung entspricht dem Vererbungskonzept in Java.

protected- und **private**-Vererbung schränken die Sichtbarkeit weiter ein

EnergyMinimizingElevatorStrategy.cpp

```
#include "EnergyMinimizingStrategy.hpp"
#include "Elevator.hpp"
using namespace std;

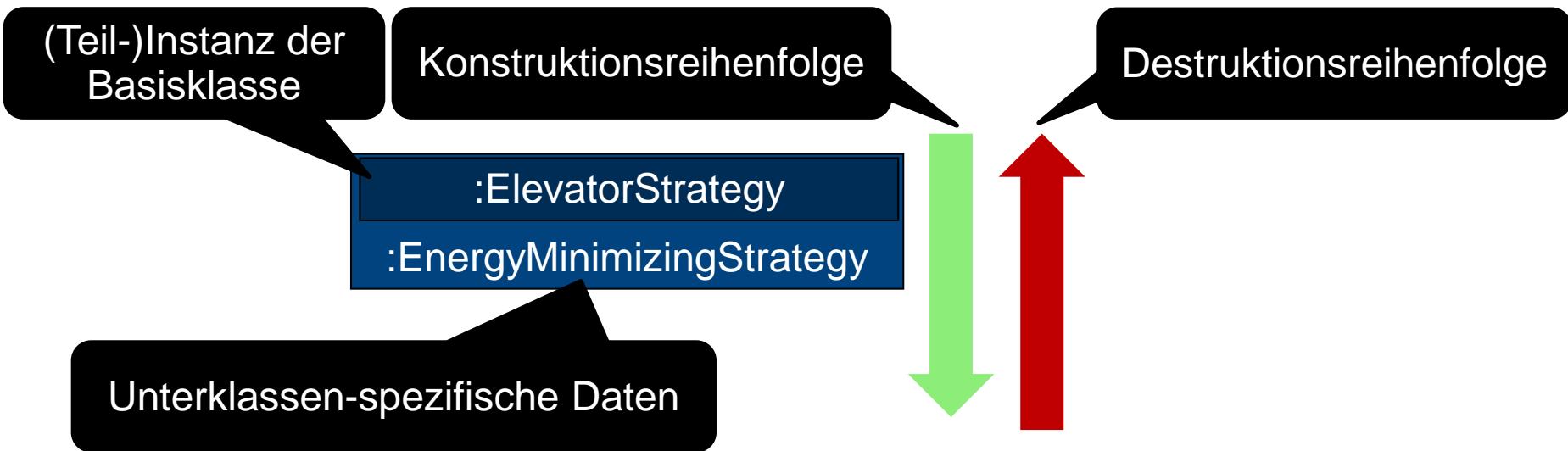
EnergyMinimizingStrategy::EnergyMinimizingStrategy()
    : ElevatorStrategy() {
    // ...
}

EnergyMinimizingStrategy::~EnergyMinimizingStrategy() {
    cout << "~EnergyMinimizingStrategy(): "
        << "Destroying energy minimizing strategy"
        << endl;
}

const Floor* EnergyMinimizingStrategy::
next(const Elevator *elevator) const{
    cout << "EnergyMinimizingStrategy::next(...): "
        << "Perform some complex calculation ..."
        << endl;
    return elevator->getCurrentFloor();
}
```

Wie **super()**-Aufruf in Java

Konstruktion und Destruktion bei Vererbung



- **Vorteil:** Während der Konstruktion von `EnergyMinimizingStrategy` kann auf die Felder von `ElevatorStrategy` zugegriffen werden.
- Wird spannend bei **Mehrfachvererbung** (siehe später)

Probelauf unserer Simulation



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

#include "Building.hpp"
#include "ElevatorStrategy.hpp"
#include "EnergyMinimizingStrategy.hpp"

int main() {
    ConstElevatorStrategyPtr strategy(new
        EnergyMinimizingStrategy());

    Building hbi(6, strategy);
    hbi.getElevator().moveToNextFloor();
}
```

Probelauf unserer Simulation



```
ElevatorStrategy(): Creating basic strategy
EnergyMinimizingStrategy(): Creating energy minimizing strategy
```

```
Floor(): Creating floor [0]
Floor(const Floor&): Copying floor [0]
~Floor(): Destroying floor [0]
```

```
Elevator(): Creating elevator.
Building(...): Creating building with 6 floors.
Building(...): Elevator is on Floor: 0
```

Konstruktoren werden
richtig aufgerufen

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

#include "Building.hpp"
#include "ElevatorStrategy.hpp"
#include "EnergyMinimizingStrategy.hpp"

int main() {
    ConstElevatorStrategyPtr strategy(new
        EnergyMinimizingStrategy());

    Building hbi(6, strategy);
    hbi.getElevator().moveToNextFloor();
}
```

Probelauf unserer Simulation



ElevatorStrategy(): Creating basic strategy
EnergyMinimizingStrategy(): Creating energy minimizing strategy

```
Floor(): Creating floor [0]
Floor(const Floor&): Copying floor [0]
~Floor(): Destroying floor [0]
```

```
Elevator(): Creating elevator.
Building(...): Creating building with 6 floors.
Building(...): Elevator is on Floor: 0
```

Elevator::moveToNextFloor(): Polymorphic call to strategy.
ElevatorStrategy::next(...): Using basic strategy ...

```
~Building(): Destroying building.
~Elevator(): Destroying elevator.
```

Konstruktoren werden
richtig aufgerufen

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

#include "Building.hpp"
#include "ElevatorStrategy.hpp"
#include "EnergyMinimizingStrategy.hpp"

int main() {
    ConstElevatorStrategyPtr strategy(new
        EnergyMinimizingStrategy());
    Building hbi(6, strategy);
    hbi.getElevator().moveToNextFloor();
}
```

Polymorpher Aufruf hat
aber nicht funktioniert!

!

Probelauf unserer Simulation



ElevatorStrategy(): Creating basic strategy
EnergyMinimizingStrategy(): Creating energy minimizing strategy

```
Floor(): Creating floor [0]
Floor(const Floor&): Copying floor [0]
~Floor(): Destroying floor [0]
```

```
Elevator(): Creating elevator.
Building(...): Creating building with 6 floors.
Building(...): Elevator is on Floor: 0
```

Konstruktoren werden
richtig aufgerufen

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

#include "Building.hpp"
#include "ElevatorStrategy.hpp"
#include "EnergyMinimizingStrategy.hpp"

int main() {
    ConstElevatorStrategyPtr strategy(new
        EnergyMinimizingStrategy());
    Building hbi(6, strategy);
    hbi.getElevator().moveToNextFloor();
}
```

Elevator::moveToNextFloor(): Polymorphic call to strategy.
ElevatorStrategy::next(...): Using basic strategy ...

```
~Building(): Destroying building.
~Elevator(): Destroying elevator.
```

```
~Floor(): Destroying floor [0]
~Floor(): Destroying floor [1]
~Floor(): Destroying floor [2]
~Floor(): Destroying floor [3]
~Floor(): Destroying floor [4]
~Floor(): Destroying floor [5]
```

~ElevatorStrategy(): Destroying basic strategy

Polymorpher Aufruf hat
aber **nicht funktioniert!**

Destruktor der
Subklasse wurde nicht
aufgerufen!



Virtuelle Methoden

- Im Gegensatz zu Java ist bei C++ aus Effizienzgründen die **polymorphe Behandlung** von Methoden **per Default ausgeschaltet**
- Es muss explizit mit dem **Schlüsselwort `virtual`** angegeben werden, welche Methoden polymorph zu behandeln sind

Virtuelle Methoden



```
class ElevatorStrategy {  
public:  
    ElevatorStrategy();  
    virtual ~ElevatorStrategy();  
  
    virtual const Floor* next(const Elevator *elevator) const;  
};
```

Regel: Klassen mit virtuellen Methoden sollten einen **virtuellen Destruktor** besitzen!

Methoden werden als virtuell gekennzeichnet (**nur im Header**)

```
class EnergyMinimizingStrategy : public ElevatorStrategy {  
public:  
    EnergyMinimizingStrategy();  
    virtual ~EnergyMinimizingStrategy();  
  
    virtual const Floor* next(const Elevator *elevator) const;  
};
```

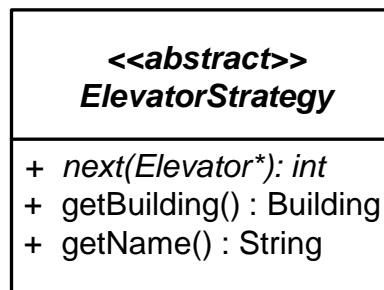
virtual muss nicht in Subklassen wiederholt werden, wird aber häufig der Übersicht halber gemacht

[Exkurs] Virtual Method Table

Der Mechanismus der dynamischen Bindung

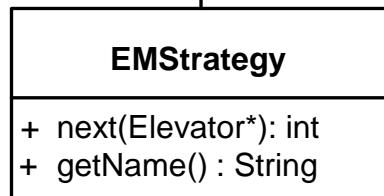


- Egal, wie der Pointer auf ein Objekt deklariert ist (z.B. `ElevatorStrategy*`), **das Objekt behält seinen Typ** (z.B. `EnergyMinimizingStrategy*`).
- Jede Klasse** besitzt eine **Lookup-Tabelle (vtable)**, die jeder virtuellen Methode ihre Implementierung zuweist.



Methode	Implementierung
<code>next</code>	<code>NULL</code>
<code>getBuilding</code>	<code>ElevatorStrategy::getBuilding</code>
<code>getName</code>	<code>ElevatorStrategy::getName</code>

Java: alle Methoden
C++: `virtual` Methoden



Methode	Implementierung
<code>next</code>	<code>EMStrategy::next</code>
<code>getName</code>	<code>EMStrategy::getName</code>

Falls kein
Eintrag/NULL:
Verwende
Methode des
Typs des
Pointers.

`ElevatorStrategy *strategy = new EnergyMinimizingStrategy()`

https://en.wikipedia.org/wiki/Virtual_method_table

Probelauf mit virtuellen Methoden



ElevatorStrategy(): Creating basic strategy

EnergyMinimizingStrategy(): Creating energy minimizing strategy

Floor(): Creating floor [0]

Floor(const Floor&): Copying floor [0]

~Floor(): Destroying floor [0]

Elevator(): Creating elevator.

Building(...): Creating building with 6 floors.

Building(...): Elevator is on Floor: 0

Polymorpher Aufruf
funktioniert jetzt



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

#include "Building.hpp"
#include "ElevatorStrategy.hpp"
#include "EnergyMinimizingStrategy.hpp"

int main() {
    ConstElevatorStrategyPtr strategy(new EnergyMinimizingStrategy());

    Building hbi(6, strategy);
    hbi.getElevator().moveToNextFloor();
}
```

Elevator::moveToNextFloor(): Polymorphic call to strategy.

EnergyMinimizingStrategy::next(...): Perform some complex calculation ...

~Building(): Destroying building.

~Elevator(): Destroying elevator.

~Floor(): Destroying floor [0]

~Floor(): Destroying floor [1]

~Floor(): Destroying floor [2]

~Floor(): Destroying floor [3]

~Floor(): Destroying floor [4]

~Floor(): Destroying floor [5]

~EnergyMinimizingStrategy(): Destroying energy minimizing strategy

~ElevatorStrategy(): Destroying basic strategy



Und alle Destruktoren werden in der
richtigen Reihenfolge aufgerufen

Pure Virtual = "virtual + =0"



```
class ElevatorStrategy {  
public:  
    ElevatorStrategy();  
    virtual ~ElevatorStrategy();  
  
    virtual const Floor* next(const Elevator *elevator) const = 0;  
};
```

ElevatorStrategy kann durch =0 nicht mehr instantiiert werden.

Methode ist hiermit **rein virtuell** – keine **inline** Implementierung in ElevatorStrategy möglich.

- Entspricht einer **abstrakten Methode** in Java.
- Klasse mit mind. einer rein virtuellen Methode entspricht **abstrakter Klasse** oder **Interface** in Java.
- Methode kann von Unterklassen implementiert werden, muss aber nicht. (~ Hierarchie abstrakter Klassen)

Typumwandlung (Casting)



Ein (Type) Cast ändert den Typ einer Variablen, also die Interpretation der gespeicherten Information.

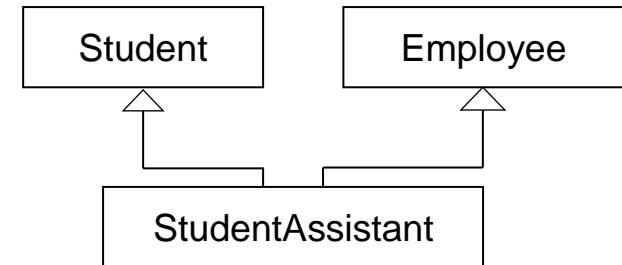
Java

- **Casts in Sonderrolle** (Sprachfeature)
- Nur Typecast: SpecialBuilding sb = (SpecialBuilding)b;
- Laufzeitfehler bei Fehlschlag: java.lang.ClassCastException

C++:

- **Casts als reguläre Funktionen**, große Vielfalt und durch Bibliotheken erweiterbar
- `int i = (int) 3.4;` C-Stil; beliebige Umwandlung ist möglich
- `static_cast<int>(3.0)` Umwandlung ohne Laufzeitcheck
- `dynamic_cast<SC*>(c)` Umwandlung von c in Typ SC* mit Laufzeitcheck
- `reinterpret_cast<C>(x)` beliebige Umwandlung in Typ C
- `const_cast<char*>(c)` Constness entfernen (z.B. `const char* → char*`)

<http://www.cplusplus.com/doc/tutorial/typecasting/>



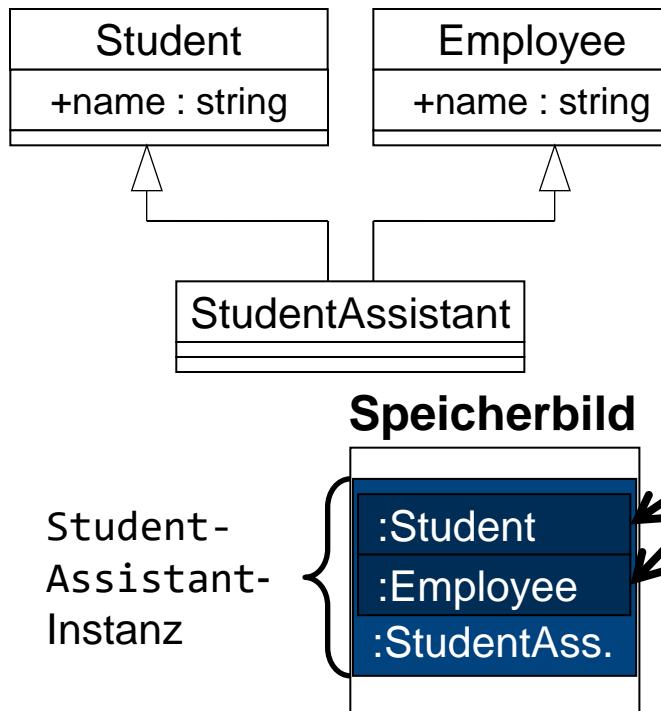
MEHRFACHVERERBUNG

Implementierungsvererbung: Konflikte



▪ Mehrfachvererbung kann zu Mehrdeutigkeit führen

Attribute und Methoden einer Oberklasse sind Bestandteil der Unterklasse (außer **private**-Elemente)



```
#include <string>

class Student {public: std::string name;};
class Employee {public: std::string name;};

class StudentAssistant
: public Student,
public Employee {};

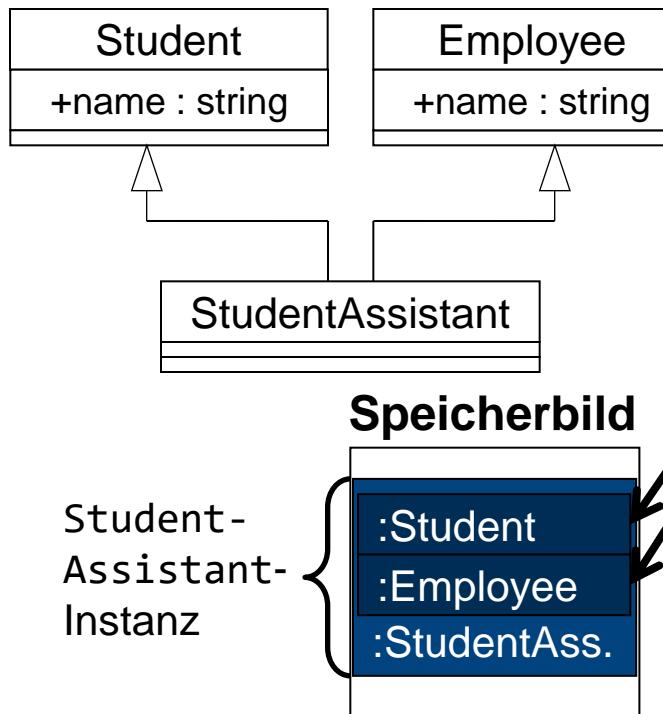
int main() {
    StudentAssistant *h = new StudentAssistant();
    h->name = "Christian";
    /* Error: request for name is ambiguous */
}
```

Namenskonflikt!
Keine eindeutige
Zuweisung ...

Implementierungsvererbung: Konflikte



- Auflösung der Mehrdeutigkeit durch Verwendung des vollständigen Namens (**Scope-Operator ::**)



```
#include <string>

class Student      {public: std::string name; };
class Employee     {public: std::string name; };

class StudentAssistant: public Student,
                      public Employee {};
```

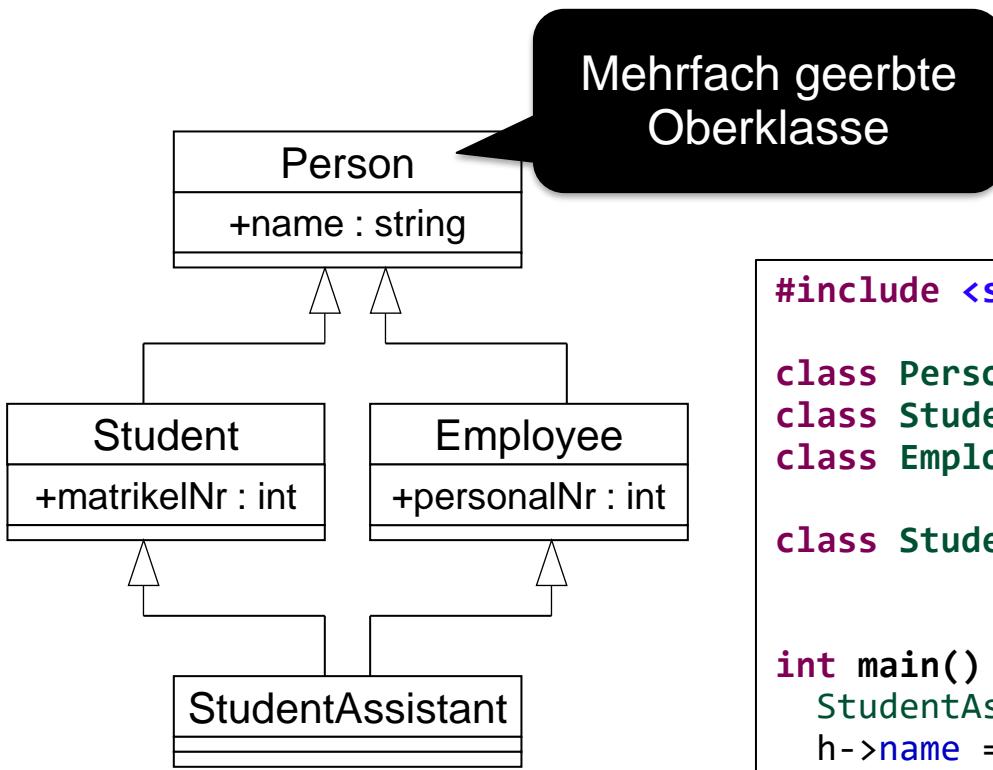
```
int main() {
    StudentAssistant* h = new StudentAssistant();
    h->Student::name = "Christian";
    h->Employee::name = "Mark";
}
```

Scope-Operator nötig!

Implementierungsvererb.: Speicherproblematik

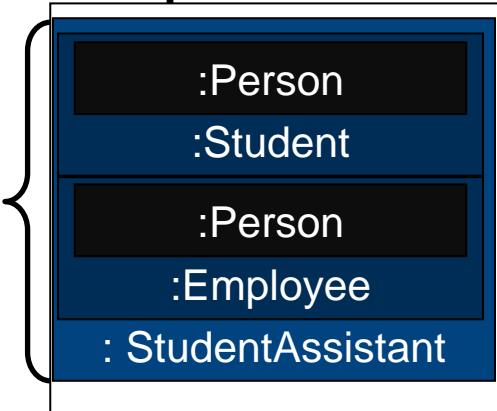


Mehrfach geerbte Oberklassen führen auch zur
unnötigen Bindung von Speicher



Student-
Assistant-
Instanz

Speicherbild



```
#include <string>
```

```
class Person {public: std::string name; };
class Student : public Person {};
class Employee : public Person {};

class StudentAssistant : public Student,
                        public Employee {};
```

```
int main() {
    StudentAssistant* h = new
    h->name = "Christian";
}
```

Fehler! Keine
eindeutige
Zuweisung ...

Implementierungsvererb.: Methoden

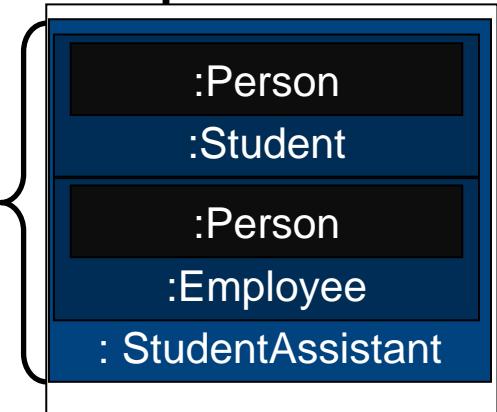


Für Methoden entsteht genau die gleiche Problematik!

```
#include <string>
class Person {
public:
    std::string getName() {return "...";}
};
class Student : public Person {};
class Employee : public Person {};
class StudentAssistant : public Student,
                           public Employee {};
int main() {
    StudentAssistant* h = new StudentAssistant ();
    h->getName();
}
```

Student-
Assistant-
Instanz

Speicherbild



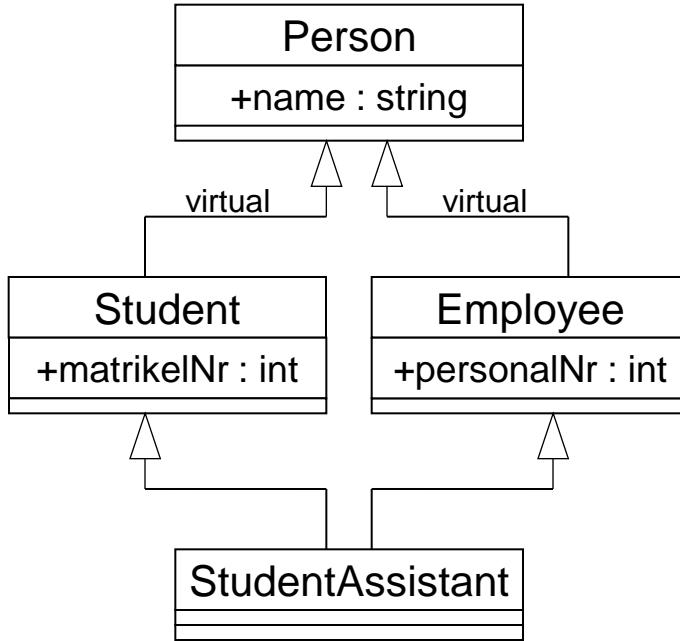
```
In function 'int main()':
11:6: error: request for
      member 'getName' is ambiguous
3:42: note: candidates are:
  std::string Person::getName()
  std::string Person::getName()
```

Hilfreich?

Virtuelle (Mehrfach-)Vererbung (I)



Lösung: Mehrfach geerbte Oberklassen nur einmal einbinden
Schlüsselwort **virtual** ermöglicht virtuelle Oberklassen / Vererbung



```
#include <string>

class Person { public: std::string name; };
class Student : virtual public Person;
class Employee: virtual public Person

class StudentAssistant:
    public Student,
    public Employee{};

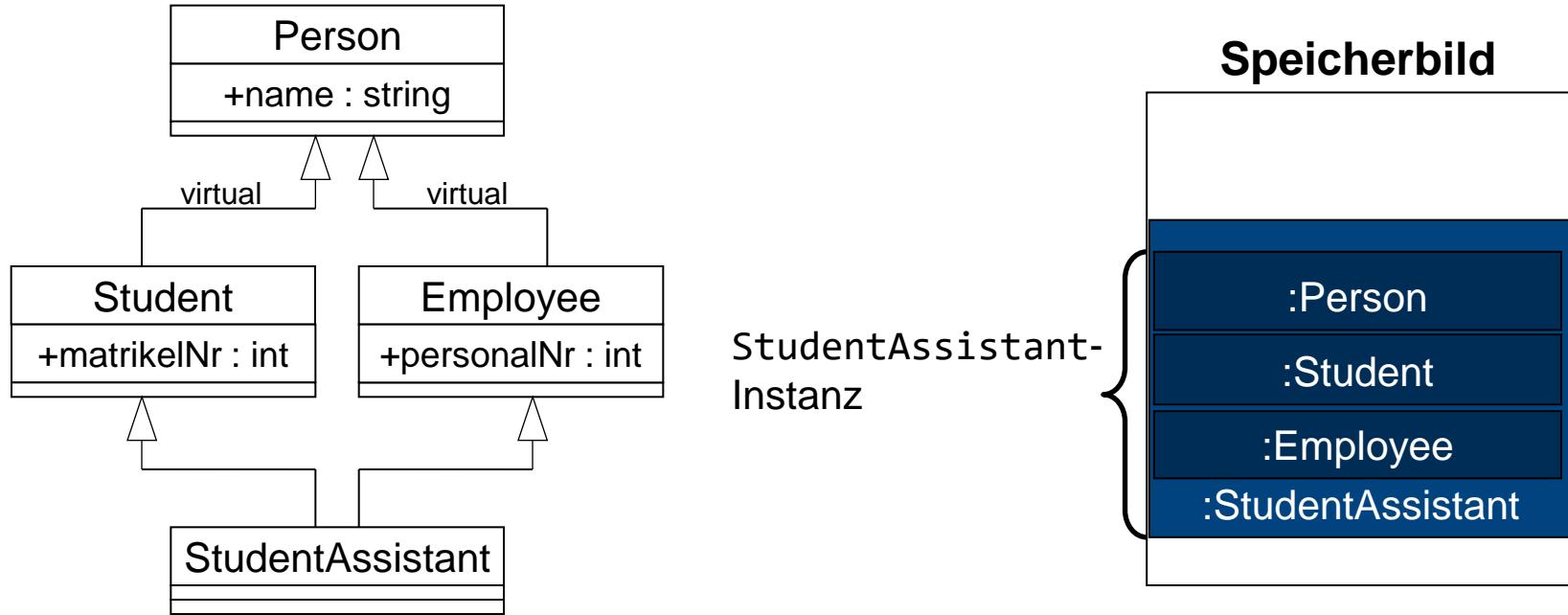
int main() {
    StudentAssistant* h = new StudentAssistant();
    h->name = "Max";
}
```

! Die **virtual**-Deklaration findet nicht an der Stelle statt, die sie nötig macht (**StudentAssistant**)!

Virtuelle (Mehrfach-)Vererbung (II)



Lösung: Mehrfach geerbte Oberklassen nur einmal einbinden
Schlüsselwort **virtual** ermöglicht virtuelle Oberklassen / Vererbung

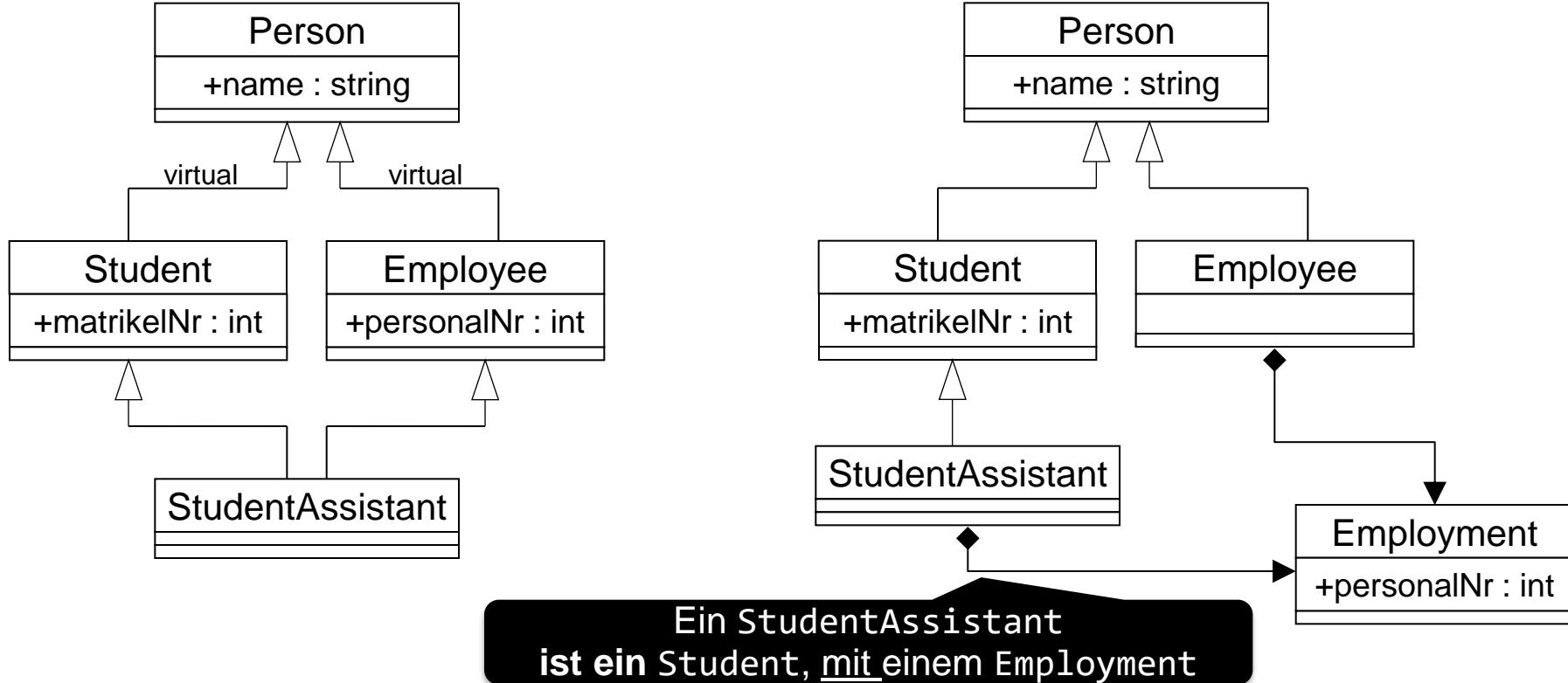


Implementierungsvererbung: Schlechtes Design?



Mehrfachvererbung kann auf schlechtes Design hindeuten:

Gemeinsamkeiten sollen explizit extrahiert und das Design vereinfacht werden



Schnittstellen- vs. Implementierungsvererbung



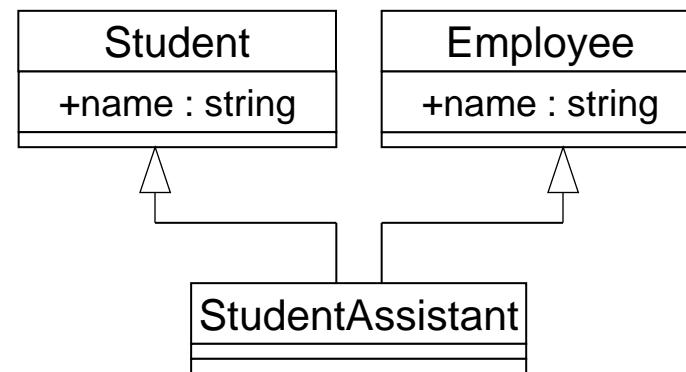
Schnittstellenvererbung:

Wenn die Oberklassen nur **pure virtual** Methoden enthalten, dann ist Mehrfachvererbung überhaupt kein Problem

! Dies entspricht der Verwendung von **Interfaces** in Java!

Implementierungsvererbung:

Wird aber von mehreren Oberklassen wirklich **Implementierung** geerbt, so kann das zu Problemen führen...



Konzept

Programmierpraktikum C und C++



Fortgeschrittene Themen

(Übungsblatt: [F])



Dr.-Ing. Eric Lenz

elenz@iat.tu-darmstadt.de

Fachgebiet Control and Cyber-Physical Systems (CCPS)

Prof. Dr.-Ing. Rolf Findeisen

Dept. of Electrical Engineering and Information Technology

<https://www.ccps.tu-darmstadt.de/>

Fortgeschrittene Themen in C++

1. Templates



2. Funktionszeiger und Funktionsobjekte

```
void (*fp1)(const string&)
            = print<string>;
fp1("foo"); // ::::> foo
```

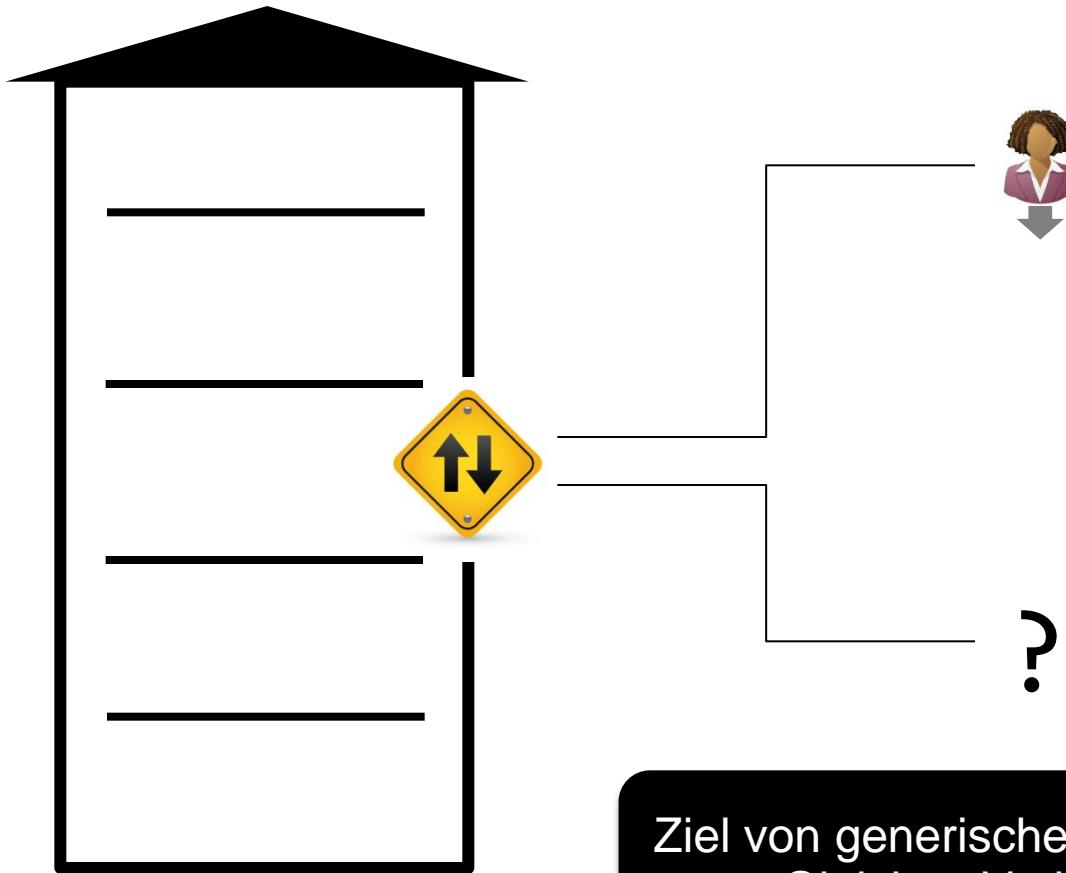
3. Überblick der Standard C++ Library

```
#include <algorithms>
#include <priority_queue>
#include <functional>
```



TEMPLATES

Generische Programmierung: Motivation



Ziel: Aufzüge für bestimmte Zwecke

- Person mit Ziel
- Lastenaufzug
- Reinigungspersonal
- Feuerwehr
- Speisen
- ...

**Ziel von generischen Datenstrukturen und Algorithmen:
Gleiches Verhalten unabhängig vom Inhalt.**

https://en.wikipedia.org/wiki/Generic_programming

void*: Generische Programmierung in C



```
struct ListElement {
    struct ListElement *next;
    struct ListElement *prev;
    void *content;
};

struct List {
    struct ListElement *firstElement;
};

int main(int argc, char **argv) {
    struct ListElement firstElement;
    firstElement.content = "some string";
    firstElement.next = NULL;

    struct List list;
    list.firstElement = &firstElement;

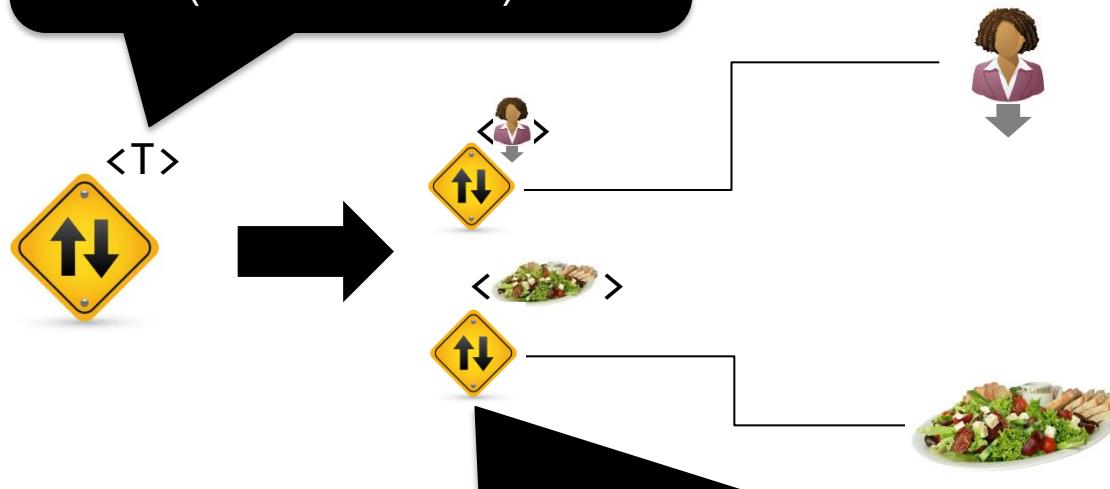
    printf("First address: 0x%p\n", list.firstElement);
    printf("First content: '%s'\n", (const char*)list.firstElement->content);
}
```

Typinformation geht verloren –
so ähnlich wie bei Java-Listen
vor den Generics

Expliziter Cast nötig

Templates in C++: Idee

Implementierung mit einem
Typ **parametrisieren**
(= "Platzhalter")



Bei Bedarf wird die richtige Version der
Implementierung **zur Kompilierzeit generiert**
("textuelle Ersetzung der Platzhalter")

Beispiel: Template-Klasse Elevator<T>



```
template<typename T = Person>
class Elevator<T> {
public:
    Elevator(){
        cout << "Elevator()" << endl;
    }
    ~Elevator(){
        cout << "~Elevator()" << endl;
    }

    void placeInElevator(const T *object){
        cout << "Adding " << object->getName()
            << " with weight: "
            << object->getWeight()
            << " to elevator.";
        cout << endl;

        transportedObjects.push_back(object);
    }

private:
    std::vector<const T*> transportedObjects;
};
```

- C++-Templates induzieren ein **implizites "Interface"** durch die **Art der Verwendung** der generischen Typparameter
- Beim Anlegen von templates sind **typename** und **class** als Schlüsselwörter möglich und bis auf wenige Ausnahmen gleichwertig.

Beispielklassen für Elevator<T>



```
class Person {  
public:  
    Person(const string& name, int weight);  
    ~Person();  
  
    inline const string& getName() const {  
        return name;  
    }  
  
    inline int getWeight() const {  
        return weight;  
    }  
  
private:  
    const string name;  
    int weight;  
};
```

```
class Dish {  
public:  
    Dish(const string& name);  
    ~Dish();  
  
    inline const string& getName() const {  
        return name;  
    }  
  
    inline double getWeight() const {  
        return 1.5;  
    }  
  
private:  
    const string name;
```

Unterschiedliche Rückgabetypen

Template-Spezialisierung: Elevator<T>



- Durch die Belegung des Typparameters (hier: T) entsteht eine (neue) Belegung des Klassentemplates (sog. **Spezialisierung** des Templates)

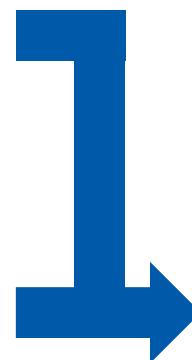
Verwendungsstelle

```
int main() {
    Elevator<> myElevator;
    Elevator<Person> myElevator2;
}
```

```
template<typename T = Person>
class Elevator {
// ...
};
```

Template

C++-Templates sind eher mit einem **Codegenerator** als mit Java-Genericss zu vergleichen!



```
template<>
class Elevator<Person> {
public:
    Elevator(){
        cout << "Elevator()" << endl;
    }
    ~Elevator(){
        cout << "~Elevator()" << endl;
    }

    void placeInElevator(const Person *object){
        cout << "Adding " << object->getName()
            << " with weight: "
            << object->getWeight()
            << " to elevator."
            << endl;

        transportedObjects.push_back(object);
    }

private:
    std::vector<const Person*> transportedObjects;
};
```

Spezialisiertes Template

Function Templates: Syntax am Beispiel



```
template<typename S, typename T>
S totalWeight(T *start, T *end, std::string things)
{
    S total = 0;

    while(start != end){
        total += start->getWeight();
        ++start;
    }

    cout << "Total weight of " << things
        << " is " << total;
    cout << endl;

    return total;
}
```

Mehrere Typparameter möglich
(auch bei Klassen-Templates)

Typ kann genauso wie in einer Klasse **frei verwendet** werden

Dies ist besonders für
generische Algorithmen sehr nützlich

Templates: Verwendung

Defaulttyp *Person* wird verwendet



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
int main(int argc, char **argv) {
Elevator<> elevator;

Person people[] = {Person("Tony", 75),
                   Person("Lukas", 14)};
elevator.placeInElevator(people);
elevator.placeInElevator(people + 1);

int totalAsInt = totalWeight<int, Person>
    (people, people + 2, "people");

Elevator<Dish> dumbwaiter;

Dish dishes[] = {Dish("Jollof Rice"),
                 Dish("Roasted Chicken")};

dumbwaiter.placeInElevator(dishes);
dumbwaiter.placeInElevator(dishes + 1);

double totalAsDouble = totalWeight<double, Dish>
    (dishes, dishes + 2,
     "dishes");
}
```

Primitive Datentypen können auch verwendet werden (anders als bei Java)

Elevator()

Person(Tony,75)
Person(Lukas,14)

Adding Tony with weight: 75 to elevator.
Adding Lukas with weight: 14 to elevator.

Total weight of people is 89

Elevator()

Dish(Jollof Rice)
Dish(Roasted Chicken)

Adding Jollof Rice with weight: 1.5 to elevator.
Adding Roasted Chicken with weight: 1.5 to elevator.

Total weight of dishes is 3

~Dish(Roasted Chicken)
~Dish(Jollof Rice)
~Elevator()
~Person(Lukas,14)
~Person(Tony,75)
~Elevator()

Induzierte Schnittstelle



Template-Code

```
template<typename S, typename T>
S totalWeight(T *start, T *end){
    S total = 0;
    while(start != end){
        total += start->getWeight();
        ++start;
    }

    cout << "Total weight of "
        << " is " << total;
    << endl;

    return total;
}
```

Induzierte Schnittstellen

```
class T {
    double getWeight();
    // or comparable return type
};

class S {
public:
    S(int i);
    void operator+=(double d);
    // or comparable parameter
};

std::ostream& operator<<(std::ostream&, const S&);
```



FUNKTIONSZEIGER UND FUNKTIONSOBJEKTE

Funktionszeiger: Syntax

```
void (*fp1)(const string&) = print<string>;
```

Name der Variable

Liste der **Parametertypen** der Funktionen, auf die gezeigt werden soll

Typ des Rückgabewerts

Zeigertyp, Klammern sind notwendig um Rückgabetyp und Zeiger auseinanderzuhalten

Adresse der Funktion (hier durch Instanziierung eines Funktion-Templates)

```
// Call the function
fp1("foo");
```



Funktionszeiger: Beispiel (I)

```
class Timer {  
public:  
    double measureDuration(  
        unsigned long iterations,  
        void (*fp)(unsigned long)) {  
        tic();  
        fp(iterations);  
        toc();  
        return getElapsedTime();  
    }  
  
    void tic();  
    void toc();  
    double getElapsedTime()  
};
```

```
void mySophisticatedAlgorithm(unsigned long iterations);  
  
#include <iostream>  
  
int main() {  
    Timer t;  
    std::cout << "Duration for 100 iterations: "  
        << t.measureDuration(100, mySophisticatedAlgorithm) << std::endl;  
    std::cout << "Duration for 1000 iterations: "  
        << t.measureDuration(1000, mySophisticatedAlgorithm) << std::endl;  
}
```

Methode, um die Laufzeit von Funktionen zu messen.

Allerdings: Nicht generisch – nur geeignet für Funktionen, die genau einen `unsigned long`-Parameter und `void` als Rückgabewert haben.

Funktionszeiger: Beispiel (II)

```
template<typename F, typename T>
void applyToSequence(F function, T* begin, T* end){
    while (begin != end) function(*begin++);
    /* Äquivalent zu:
     * while(begin != end) {
     *     function(*begin);
     *     ++begin;
     * }
```

```
}
```

```
template<typename S> void print(const S& s){
    std::cout<< "::::> " << s << std::endl;
}
```

```
void validateAges(int a){
    if(a > 100 || a < 0)
        std::cout<< a << " is not a valid age!" << std::endl;
}
```

```
int main() {
    int n[] = {-1, 20, 33, 120};
    applyToSequence(print<int>, n, n + 4);
    applyToSequence(validateAges, n, n + 4);
}
```

function wird hier als Funktion übergeben und kann als solche direkt verwendet werden

Ermöglicht kompakte, elegante, und sehr generische Algorithmen

Verwendung ist **sehr leichtgewichtig** und erfordert keine extra Klassen oder Schnittstellen für viele kleinen Funktionen



Funktionszeiger: Beispiel (III)

```
template<typename S>
void print(const S& s){
    cout << "::::> " << s << endl;
}

void validateAges(int a){
    if(a > 100 || a < 0){
        std::cout << a << " is not a valid age!" << std::endl;
    }
}

int main() {
    void (*fp1)(const string&) = print<string>;
    void (*fp2)(int) = validateAges;

    fp1("foo");    // ::::> foo
    fp2(500);      // 500 is not a valid age
}
```

Zeiger auf eine Funktion
mit const string&
Parameter

Verwendung wie ein
normaler
Funktionsaufruf

Der Fluch des Most Vexing Parse



- Warum funktioniert das Folgende nicht?

```
class Building{  
public:  
    int floorCount;  
};  
int main() {  
    Building b();  
    b.floorCount;  
}
```

- Fehlermeldung:

error: request for member 'floorCount' in 'b', which is of non-class type 'Building()'

- Grund: Der C++-Compiler interpretiert b als einen Funktionszeiger, der auf eine parameterlose, Building-zurückgebende Funktion zeigt.
- Lösung: Klammern weglassen oder Initialisierungsliste (ab C++11)

```
int main() {  
    Building b2;  
    Building b3{};  
}
```

https://en.wikipedia.org/wiki/Most_vexing_parse

Funktoren und Funktionsobjekte



- Ein **Funktor** ist eine Klasse, die `operator()` implementiert.

```
class ConsoleLogger {  
public:  
    std::string prefix("user");  
  
    inline void operator()(int i) const {  
        std::cout << prefix << ":~ /$ "  
            << i << std::endl;  
    }  
};  
  
template<typename F, typename T>  
void applyToSequence(F function, T* begin, T* end){  
    while (begin != end) function(*begin++);  
}  
  
int main() {  
    int n[] = {-1, 20, 33, 120};  
    applyToSequence(ConsoleLogger(), n, n + 4);  
}
```

Konfigurierbares Präfix
(Hier ohne Setter).

operator() erlaubt, Objekte mit Funktionssyntax anzusprechen

Syntax bleibt hier identisch, obwohl wir eine Methode aufrufen

Jetzt kann eine Instanz der Klasse (ein Funktionsobjekt) übergeben werden

Funktionszeiger und Funktoren: Fazit



- Zeiger auf Funktionen ermöglichen einen eher **funktionalen Programmierstil** (ideal für generische Algorithmen).
- Mithilfe von Funktionszeigern kann man auch in C Polymorphie erreichen.
- In Verbindung mit Templates entsteht typischerweise ein **schlankeres, kompakteres Design** als in Java (reine OO)
- **Ideal für kleine Funktionen**, um einen Wildwuchs an kleinen Klassen (z.B. mit jeweils nur einer Methode und ohne Zustand) zu vermeiden
- Vorteil von **Funktoren/Funktionsobjekte** gegenüber Funktionszeigern: Konfigurierbar über Attribute, da Funktoren Klassen sind
- Syntaktische Verwendung von Funktionszeigern und Funktoren ist gleich dank operator(), wenn Templateparameter genutzt werden (s. applyToSequence).



TECHNISCHE
UNIVERSITÄT
DARMSTADT

STANDARD TEMPLATE LIBRARY (STL)

Generische STL-Algorithmen: std::copy



```
template <class InputIterator, class OutputIterator>
OutputIterator copy (InputIterator first, InputIterator last, OutputIterator result);
```

Parameters:
`first, last`

STL-weite Konvention zur Nutzung von Iteratoren

Input iterators to the initial and final positions in a sequence to be copied. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`result`

Output iterator to the initial position in the destination sequence. This shall not point to any element in the range `[first, last)`.

Return Value:

An iterator to the end of the destination range where elements have been copied.

<http://www.cplusplus.com/reference/algorithm/copy/>

Generische STL-Algorithmen: std::copy



```
template <class InputIterator, class OutputIterator>
OutputIterator copy (InputIterator first, InputIterator last, OutputIterator result);
```

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

```
int main(int argc, char **argv) {
    int numbers[] = {1,2,3,4,5};
    vector<int> result;

    std::copy(numbers, numbers + 5, std::back_inserter(result));

    std::copy(result.begin(), result.end(),
              std::ostream_iterator<int>(std::cout, ", "));
}
```

Erzeugt einen **OutputIterator** aus
einem Behälter (vector)

STL-Behälter bieten
InputIteratoren an

Erzeugt einen **OutputIterator**
aus einem Stream (cout)

<http://www.cplusplus.com/reference/algorithm/copy/>

Generische STL-Algorithmen:

std::remove_copy_if



```
template <class InputIterator, class OutputIterator, class UnaryPredicate>
OutputIterator remove_copy_if ( InputIterator first, InputIterator last,
                               OutputIterator result, UnaryPredicate pred);
```

Wie `copy`, aber ein Prädikat definiert, was ausgelassen wird.

Parameters:

`first, last, result` -> [Wie bei `copy`]
`pred`

Unary function that accepts an element in the range as argument, and returns a value convertible to `bool`. The value returned indicates whether the element is to be removed from the copy (if true, it is not copied).

The function shall not modify its argument.

This can either be a function pointer or a function object.

Return Value:

An iterator pointing to the end of the copied range, which includes all the elements in `[first, last)` except those for which `pred` returns true.

http://www.cplusplus.com/reference/algorithm/remove_copy_if/

Generische STL-Algorithmen: std::remove_copy_if



```
template <class InputIterator, class OutputIterator, class UnaryPredicate>
OutputIterator remove_copy_if ( InputIterator first, InputIterator last,
                               OutputIterator result, UnaryPredicate pred);
```

```
bool even(int i){  
    return i % 2 == 0;  
}
```

Funktion even entscheidet
was ausgelassen wird

```
int main(int argc, char **argv) {  
    int numbers[] = {1,2,3,4,5};  
    vector<int> result(numbers, numbers + 5);  
  
    remove_copy_if(result.begin(), result.end(),  
                  ostream_iterator<int>(cout, ", "),  
                  even); // 1, 3, 5  
}
```

Funktionszeiger oder
Funktionsobjekt übergeben

http://www.cplusplus.com/reference/algorithm/remove_copy_if/

Generische Behälter: std::priority_queue



```
template <class T,
```

Typ vom Inhalt der
Warteschlange

```
    class Container = vector<T>,
```

Typ des darunterliegenden
Behälters (std::vector<T> wird
als Default verwendet)

```
    class Compare = less<  
        typename Container::value_type>  
>
```

Binäres Prädikat (less wird
als Default verwendet)

```
class priority_queue;
```

Damit Compiler weiß, dass
value_type ein Typ ist

Default Template-Parameter erlauben **einfache**,
aber bei Bedarf **konfigurierbare** Verwendung!

http://www.cplusplus.com/reference/queue/priority_queue/

Generische Behälter: std::priority_queue



```
template <class T,  
         class Container = vector<T>,  
         class Compare = less<typename Container::value_type> >  
class priority_queue;
```

```
#include <iostream>  
#include <queue>  
#include <functional>  
  
using namespace std;  
  
template<class T>  
void process_queue(T& queue){  
    while(!queue.empty()){

        cout << queue.top()
            << ",";
        queue.pop();
    }
}
```

Einfache Hilfsfunktion
für die Ausgabe

```
int main(int argc, char **argv) {  
    int numbers[] = {3,2,1,5,4};  
  
    std::priority_queue<int>  
        descending(numbers, numbers + 5);  
    process_queue(descending);
                                // 5,4,3,2,1  
  
    std::priority_queue<int,
                        vector<int>,
                        greater<int> >
        ascending(numbers, numbers + 5);  
    process_queue(ascending);
                                // 1,2,3,4,5
}
```

http://www.cplusplus.com/reference/queue/priority_queue/

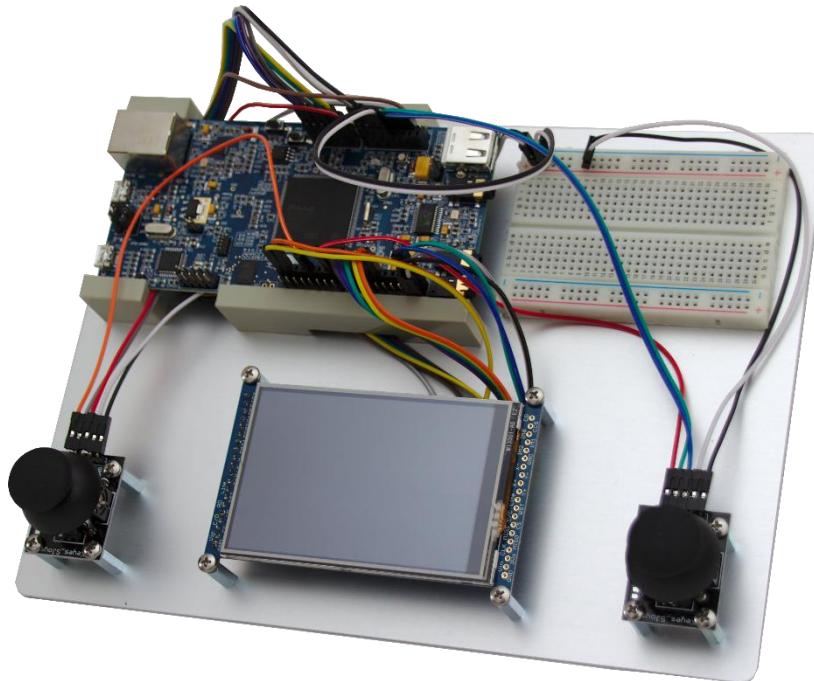
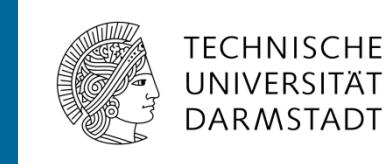
Standard Template Library: Fazit

- **Mächtig, effizient, ausgereift und gut dokumentiert**
- Anspruchsvoll zu erlernen (erfordert Wissen über Templates, Funktoren, Iteratoren, ...)
- **Boost** als "Brutkasten" für die nächsten Standards
- Vielleicht sogar als **der Vorteil** von C++ zu betrachten!

Programmierpraktikum C und C++

Einführung in (Embedded) C

(Übungsblatt: [C])



Fachgebiet Control and Cyber-Physical Systems (CCPS)

Prof. Dr.-Ing. Rolf Findeisen

Dept. of Electrical Engineering and Information Technology

<https://www.ccps.tu-darmstadt.de/>

Dr.-Ing. Eric Lenz

elenz@iat.tu-darmstadt.de

Die Eigenheiten von (Embedded) C kennenlernen.

- **Unterschiede zu C++:** Was macht C anders als C++?
- **Bitoperationen:** Bits setzen, löschen, "kippen", "verschieben" (auch für C++)
- **Peripherie und Speicher:** lesen/schreiben, Memory-mapped I/O, volatile Variablen

Unterschiede von C und C++



▪ C-Standardbibliothek ist eingebettet in C++-Standardbibliothek

- Relativ umfangreich (Stringmanipulation, printf,...)
- z.B. #include <cstdlib> oder #include <stdlib.h>

▪ Limitierungen

- Keine Objektorientierung (Vererbung, Klassen,...) → Nur structs
- Keine Namensräume → Sichtbarkeit über static, extern
- Keine String-Klasse → nur nullterminierte char-Arrays (vgl. Parameterübergabe von main)
- Keine Templates → Ausweichen über void*
- Keine Referenzen → nur Pointer und Werte
- Keine Exceptions → Error Codes (int)

▪ Unterschiede

- **Konstanten** wurden **früher** mittels Präprozessor-Direktiven abgelegt
 - z.B. #define ID "123abc" – gleiche Benennungskonvention ist kein Zufall.
- **Leere Parameterliste**: "don't care" → void signalisiert eine leere Parameterliste.
 - z.B. int getHour(); → int getHour(void);
- **Konventionen** für Dateiendungen: .c/.h statt .cpp/.hpp
- **Speicherverwaltung**: malloc und free statt new und delete



Bits und Bytes

- In Embedded C wird oft **auf einzelnen Bits von (ganzzahligen) Variablen** operiert (char, short, int, long)

- **Basistyp:** unsigned char = 1 Byte
 - Plattformunabhängig!

- **Hexadezimalnotation in C/C++:**
Aufteilung in zwei Halb-Bytes (Nibble)

- **Beispiele:**

- `unsigned char x = 0xA4;`
// Bit pattern: 1010 0100
- `unsigned int x = 0xF1BC;`
// Bit pattern: 1111 0001 1011 1100

Hex. Halbbyte	Bits	Hex. Halbbyte	Bits
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Bitoperationen – Überblick



- Bitoperationen sind nur für **ganzzahlige Datentypen** definiert (char, short, int, long, longlong)
- "outplace"- und "inplace"-Variante (z.B. & und &=)
- **Logische Operatoren** (||, &&, !) behandeln **ganzen Wert**

Auch in C++
verfügbar!

Op.	Symbol	Beschreibung	Beispiel (mit unsigned Halbbytes)
AND	&	Bitweises Und (Logisch: &&)	$0111_2 \& 0101_2 = 0101_2$
OR		Bitweises Oder (Logisch:)	$1000_2 0001_2 = 1001_2$
XOR	^	Bitweises exkl. Oder	$1000_2 1011_2 = 0011_2$
NOT	~	Bitweise Negation (Logisch: !)	$\sim 1000_2 = 0111_2$
Right shift	>>	Verschiebung aller Stellen nach rechts (Füllen mit '0'/'1' von links)	$0100_2 >> 2 = 0001_2$
Left shift	<<	Verschiebung aller Stellen nach links (Füllen mit '0' von rechts)	$0001_2 << 3 = 1000_2$

Bitoperationen – Bytes manipulieren



Grundidee: Erzeuge eine **Maske**, die **nur** an den **gewünschten Stellen** eine '0' bzw. '1' hat.

```
#include <iostream>
int main() {
    unsigned char b = 16; // b = 0x10
    // Set 5th bit of b
    b = b | (1 << 5); // or: b |= 32 , or b |= 0x20
    std::cout << "b=" << (int)b << std::endl; // 48 = 0x30
```

Wir zählen **Bits von 0** an:
7.Bit, ..., 0.Bit

b 00010000
mask 00100000
b' 00110000

Bitoperationen – Bytes manipulieren



Grundidee: Erzeuge eine **Maske**, die nur an den gewünschten Stellen eine '0' bzw. '1' hat.

```
#include <iostream>
int main() {
    unsigned char b = 16; // b = 0x10
    // Set 5th bit of b
    b = b | (1 << 5); // or: b |= 32 , or b |= 0x20
    std::cout << "b=" << (int)b << std::endl; // 48 = 0x30
```

Wir zählen **Bits von 0** an:
7.Bit, ..., 0.Bit

b	00010000
mask	00100000
b'	00110000

```
// Unset 2nd bit of b
b = 7; // = 4+2+1 = 0x07
b = b & (~ (1 << 2)); // or: b &= ~4 , or b &= 0xFB
std::cout << "b=" << (int)b << std::endl; // 3 = 0x03
```

b	00000111
mask	11111011
b'	00000011

Bitoperationen – Bytes manipulieren

Grundidee: Erzeuge eine **Maske**, die nur an den gewünschten Stellen eine '0' bzw. '1' hat.

```
#include <iostream>
int main() {
    unsigned char b = 16; // b = 0x10
    // Set 5th bit of b
    b = b | (1 << 5); // or: b |= 32 , or b |= 0x20
    std::cout << "b=" << (int)b << std::endl; // 48 = 0x30
```

Wir zählen **Bits von 0 an:**
7.Bit, ..., 0.Bit

```
// Unset 2nd bit of b
b = 7; // = 4+2+1 = 0x07
b = b & (~ (1 << 2)); // or: b &= ~4 , or b &= 0xFB
std::cout << "b=" << (int)b << std::endl; // 3 = 0x03
```

```
// Determine status of 6th bit of b
b = 192; // = 128+64 = 0xC0
char isBitSet = b & (1 << 6); // or: b & 64 , or: b & 0x40
std::cout << "6th bit set: " << (int)isBitSet << std::endl; //64
```

b	00010000
mask	00100000
b'	00110000

b	00000111
mask	11111011
b'	00000011

b	11000000
mask	01000000
b'	01000000

Bitoperationen – Bytes manipulieren



Grundidee: Erzeuge eine **Maske**, die nur an den gewünschten Stellen eine '0' bzw. '1' hat.

```
#include <iostream>
int main() {
    unsigned char b = 16; // b = 0x10
    // Set 5th bit of b
    b = b | (1 << 5); // or: b |= 32 , or b |= 0x20
    std::cout << "b=" << (int)b << std::endl; // 48 = 0x30
```

Wir zählen **Bits von 0 an:**
7.Bit, ..., 0.Bit

```
// Unset 2nd bit of b
b = 7; // = 4+2+1 = 0x07
b = b & (~ (1 << 2)); // or: b &= ~4 , or b &= 0xFB
std::cout << "b=" << (int)b << std::endl; // 3 = 0x03
```

```
// Determine status of 6th bit of b
b = 192; // = 128+64 = 0xC0
char isBitSet = b & (1 << 6); // or: b & 64 , or: b & 0x40
std::cout << "6th bit set: " << (int)isBitSet << std::endl; // 64
```

```
// Flip 3rd bit of b
b = 9; // = 8+1 = 0x09
b = b ^ (1 << 3); // or: b ^= 8 , or: b ^= 0x08
std::cout << "b=" << (int)b << std::endl; // 1 = 0x01
b = b ^ (1 << 3); // or: b ^= 8 , or: b ^= 0x08
std::cout << "b=" << (int)b << std::endl; // 9 = 0x09
```

b	00010000
mask	00100000
b'	00110000

b	00000111
mask	11111011
b'	00000011

b	11000000
mask	01000000
b'	01000000

b	00001001
mask	00001000
b'	00000001
b''	00001001

Bitoperationen – Rechnen



▪ Positive Zahlen

- **Left shift** entspricht Multiplikation mit 2
- **Right shift** entspricht Division durch 2
- Verhalten bei **negativen Zahlen** abhängig von Zahlendarstellung (z.B. Zweierkomplement)

```
#include <iostream>
using namespace std;
int main() {
    // Shifting positive numbers
    cout << " 1 << 1 = " << (1 << 1) << endl; // 1 << 1 = 2
    cout << " 1 << 2 = " << (1 << 2) << endl; // 1 << 2 = 4
    cout << " 1 >> 1 = " << (1 >> 1) << endl; // 1 >> 1 = 0 (= 1 div 2)
    cout << "16 >> 2 = " << (16 >> 2) << endl; // 16 >> 2 = 4
    cout << "17 >> 2 = " << (17 >> 2) << endl; // 17 >> 2 = 4 (= 17 div 4)

    // Shifting negative numbers
    cout << " -1 << 1 = " << (-1 << 1) << endl; // -1 << 1 = -2
    cout << " -1 << 2 = " << (-1 << 2) << endl; // -1 << 2 = -4
    cout << " -1 >> 1 = " << (-1 >> 1) << endl; // -1 >> 1 = -1
    cout << "-16 >> 2 = " << (-16 >> 2) << endl; // -16 >> 2 = -4
    cout << "-17 >> 2 = " << (-17 >> 2) << endl; // -17 >> 2 = -5
}
```

Übrigens: Undefined Behavior falls Shift-Weite negativ

Bitoperationen – Rechnen



▪ Positive Zahlen

- **Left shift** entspricht Multiplikation mit 2
- **Right shift** entspricht Division durch 2
- Verhalten bei **negativen Zahlen** abhängig von Zahlendarstellung (z.B. Zweierkomplement)

```
#include <iostream>
using namespace std;
int main() {
    // Shifting positive numbers
    cout << " 1 << 1 = " << (1 << 1) << endl; // 1 << 1 = 2
    cout << " 1 << 2 = " << (1 << 2) << endl; // 1 << 2 = 4
    cout << " 1 >> 1 = " << (1 >> 1) << endl; // 1 >> 1 = 0 (= 1 div 2)
    cout << "16 >> 2 = " << (16 >> 2) << endl; // 16 >> 2 = 4
    cout << "17 >> 2 = " << (17 >> 2) << endl; // 17 >> 2 = 4 (= 17 div 4)

    // Shifting negative numbers
    cout << " -1 << 1 = " << (-1 << 1) << endl; // -1 << 1 = -2
    cout << " -1 << 2 = " << (-1 << 2) << endl; // -1 << 2 = -4
    cout << " -1 >> 1 = " << (-1 >> 1) << endl; // -1 >> 1 = -1
    cout << " -16 >> 2 = " << (-16 >> 2) << endl; // -16 >> 2 = -4
    cout << " -17 >> 2 = " << (-17 >> 2) << endl; // -17 >> 2 = -5
}
```

Vorsicht bei
negativen Zahlen

Problematik – Zweierkomplement
Beispiel: (L-Shift)
1011 1111 << 1
(eigentlich: $-65 * 2 = -130$)
= 0111 1110
(tatsächlich: 126)

Übrigens: Undefined Behavior falls Shift-Weite negativ

Memory-mapped I/O – Motivation



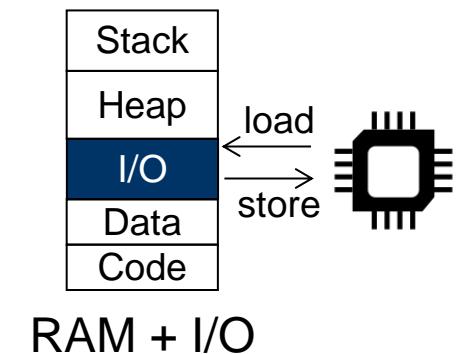
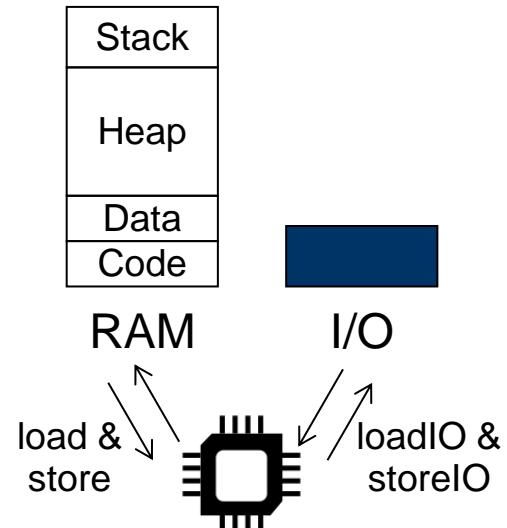
Wie greife ich auf die Peripherie eines Prozessors zu?

Port-mapped I/O

- Der Prozessor besitzt **spezielle Befehle** und einen **eigenen Adressraum**, um auf Peripherie zuzugreifen.
- (+) **Vollständiger Adressraum** für Applikation verfügbar
- (-) **Größerer Befehlssatz** (Software, Hardware, Lernkurve)

Memory-mapped I/O

- Ein **Teil des Arbeitsspeichers** ist "virtuell" für die Peripherie reserviert.
- (+) **Einheitlicher Zugriff** auf "normalen" Speicher und Peripherie-Daten
- (-) Verlust eines Teils des Adressraums
- **Variablen, die auf den gemappten Adressraum zugreifen, müssen volatile sein, da sich die Werte der Peripherie jederzeit ändern können!**



Schlüsselwort volatile



- **Beispiel:** Warten auf einen Tastendruck mittels Polling/Busy Waiting
- **DDRX:** Data Direction Register von Port X
 - 0: Output, 1: Input
- **PDIRX:** Data Input Register von Port X
 - Hier: 0: Button gedrückt, 1: Button frei
- **PDIR2_f.P0:**
 - Pin 0 des Port 2 (Direktzugriff über struct)
- **Ohne volatile:** Compiler kann Endlosschleife erzeugen (**while** (**true**) ;), da **FM4_GPIO->PDIR2_f.P0** nie gesetzt wird.

```
#include "s6e2ccx1.h"

// #define FM_GPIO_DDR2
//    *((volatile uint32_t*) (0x4006F208UL))
// #define FM4_GPIO_PDIR2
//    *((volatile uint32_t*) (0x4006F308UL))

int main() {
    //FM4_GPIO->DDR2_f.P0 = 0; Set to output
    FM4_GPIO->DDR2_f.P0 = 1; // Set to input

    while(FM4_GPIO->PDIR2_f.P0 == 1) {
        // Polling loop...
    }

    // Button has been pressed
}
```

Schlüsselwort volatile – Überblick



- Mithilfe des Schlüsselworts volatile deklariert man Variablen, **deren Wert sich jederzeit unerwartet (aus Compiler-Sicht) ändern kann.**
- Ähnlich wie const ist volatile Teil des Typs einer Variablen (*qualifier*)

Auch in C++
verfügbar!

▪ Syntax

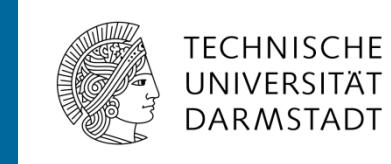
- `volatile int i;` → Der Wert von i kann sich ändern
 - (äquivalent zu) `int volatile i;`
- `volatile int *iP;` → Der referenzierte Speicher kann sich ändern.
- `volatile int *volatile iP;` → Adresse und Wert können sich ändern

▪ Einsatzgebiete

- Hardwarezugriff bei Memory-Mapped I/O
- Signal Handling (nicht gezeigt)
- Manipulation von globalen Variablen durch Interrupt Service Routinen (nicht gezeigt)

<https://barrgroup.com/Embedded-Systems/How-To/C-Volatile-Keyword>

Programmierpraktikum C und C++



Epilog



Dr.-Ing. Eric Lenz
elenz@iat.tu-darmstadt.de

Fachgebiet Control and Cyber-Physical Systems (CCPS)
Prof. Dr.-Ing. Rolf Findeisen
Dept. of Electrical Engineering and Information Technology
<https://www.ccps.tu-darmstadt.de/>



Nützliche Kommentare
finden sich
auch in den PowerPoint-Notizen!

Das war noch lange nicht das Ende... 😊

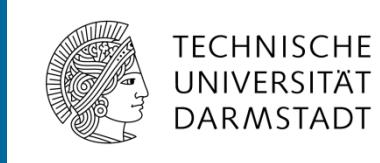


TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Weitere Lehrveranstaltungen an der TU Darmstadt
 - Advanced Multithreading in C++
- Wissenswertes (ein paar Ideen)
 - C++ Rvalue References Explained
 - Seit C++11 unterstützt C++ die sogenannte Move-Semantik, die z.B. beim Zuweisen von Objekten einen Speicher-/Laufzeit-effizienten Transfer von Objekten ermöglicht
 - Tipps zum Überladen von Operatoren
 - "Wie überlade ich Operatoren für meine Klasse, sodass niemand überrascht wird."
 - <http://courses.cms.caltech.edu/cs11/material/cpp/donnie/cpp-ops.html>

Programmierpraktikum C und C++

[Exkurs] Zusätzliche Materialien
(allesamt nicht prüfungsrelevant)



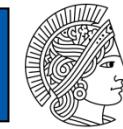
Dr.-Ing. Eric Lenz
elenz@iat.tu-darmstadt.de

Fachgebiet Control and Cyber-Physical Systems (CCPS)
Prof. Dr.-Ing. Rolf Findeisen
Dept. of Electrical Engineering and Information Technology
<https://www.ccps.tu-darmstadt.de/>



TECHNISCHE
UNIVERSITÄT
DARMSTADT

RULE OF THREE



Implementiert man **Copy-Konstruktor**, **Assignment-Operator** oder **Destruktor**, muss man vermutlich auch die anderen Beiden implementieren.

```
Floor::Floor(  
    std::string label, int number):  
    label(label),  
    number(number) {  
    cout << "Creating floor" << number << "]" << endl;  
}  
  
Floor::Floor(const Floor &floor):  
    label(floor.label),  
    number(floor.number+1) {  
    cout << "Copying floor" << floor.number << "]" << endl;  
}  
  
Floor::~Floor() {  
    cout << "Destroying floor [" << number << "]" << endl;  
}
```

Inkonsistentes Verhalten mit Compiler-generiertem operator=

Vergleiche:

```
Floor f1("f1", 1);  
Floor f2 = f1;  
Floor f3("f3", 3);  
f3 = f1;
```



Implementiert man **Copy-Konstruktor**, **Assignment-Operator** oder **Destruktor**, muss man vermutlich auch die anderen Beiden implementieren.

- Der Compiler generiert einen der Drei bei Bedarf automatisch, indem Felder kopiert werden (evtl. mittels "rekursivem" Copy-Konstruktor).
- Wenn ich **Resourcen** (Speicher, File Handle,...) in einem **Konstruktor** akquiriere, möchte ich sie auch im **Destruktor** freigeben.
- Wenn ich im Konstruktor Heap-Speicher allokiere, dann muss ich diesen im Destruktor per delete freigeben. Was passiert aber wenn ich das notwendige Pointer-Attribut im Kopierkonstruktor einfach kopiere?
(→Double Delete!)
- Verwende ich einen **eigenen Copy-Konstruktor** und einen **generierten Assignment-Operator**, kann es zu **inkonsistentem Verhalten** kommen.



TECHNISCHE
UNIVERSITÄT
DARMSTADT

IMMUTABLE DATENTYPEN

[Exkurs] Weak SmartPointer: Motivation

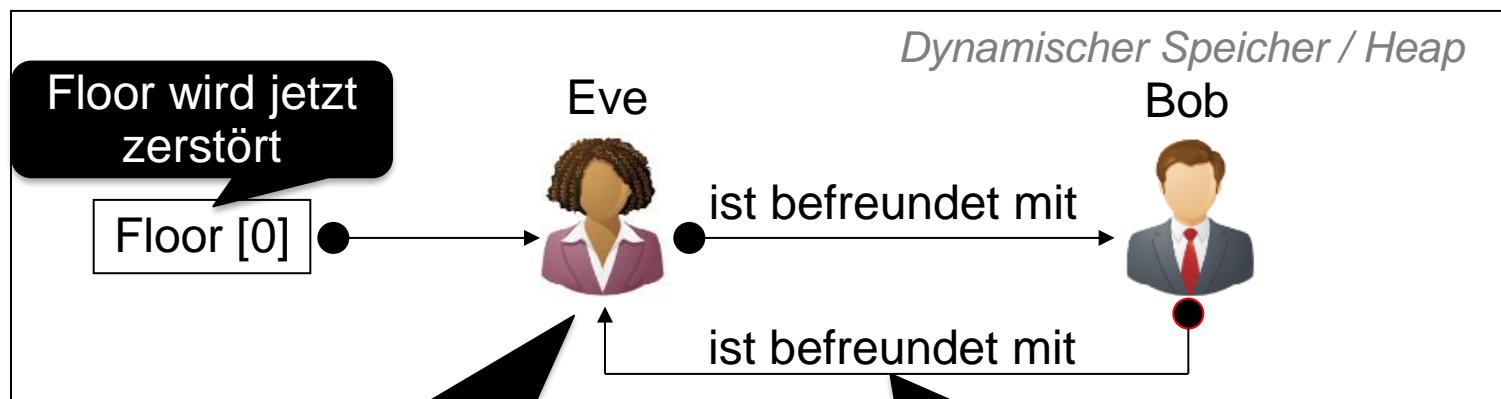


`std::shared_ptr<>` ist nicht perfekt:

- Etwas langsamer als Rohzeiger
- Erkennt **zirkuläre Abhängigkeiten** nicht:

Ablauf:

1. Objekt `Floor[0]` wird zerstört
2. Fertig – Eve und Bob halten sich gegenseitig am Leben.



Eve wird nicht zerstört, weil Bob auf Eve zeigt, und umgekehrt!

Eve ist mit Bob befreundet, und (natürlich) auch Bob mit Eve ...

[Exkurs] Lösung: Verzicht auf Zeiger (I)



```
class Person {  
public:  
// ...  
private:  
    std::vector<Person> friends;  
// ...  
};
```

```
class Elevator {  
public:  
// ...  
private:  
    std::vector<Person> containedPersons;  
// ...  
};
```

```
class Floor {  
public:  
// ...  
private:  
    std::vector<Person> containedPersons;  
// ...  
};
```



Welches neue Problem handeln wir uns damit ein?



Eine Person existiert jetzt **mehrfach!** (s. nächste Folie)

[Exkurs] Lösung: Verzicht auf Zeiger (II)



```
int main(int argc, char **argv) {  
  
Person eve("Eve", 55.0); // initial weight: 55kg  
Person bob("Bob", 80.0); // initial weight: 80kg  
  
cout << bob.getName() << " has weight " << bob.getWeight() << endl;  
  
Person::makeFriends(eve, bob);  
  
Person &bobAsEvesFriend = eve.getFriends().at(0);  
bobAsEvesFriend.setWeight(95);  
cout << bobAsEvesFriend.getName() << " [as Eve's friend] has weight " <<  
    bobAsEvesFriend.getWeight() << endl;  
  
cout << bob.getName() << " has weight " << bob.getWeight() << endl;  
}  
}
```

Ausgabe:

Bob has weight 80
Bob [as Eve's friend] has weight 95
Bob has weight 80

Mit **immutablen Objekten**
(→ `java.lang.String`) umgehbar

https://en.wikipedia.org/wiki/Immutable_object



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Mehrfachvererbung mit Templates mischen

MIXINS

[Exkurs] Mixins: Mehrfachvererbung trifft Templates



```
template<
    class Logger,
    class Security,
    class OperatingSystem,
    class Platform
>
class System :
    public Logger,
    public Security,
    public OperatingSystem,
    public Platform
{
    // Nothing else needed!
};
```

Mixins werden als Typparameter definiert...

...und "reingemischt" mit Mehrfachvererbung!

[Exkurs] Mixins: Mehrfachvererbung trifft Templates



```
int main(int argc, char **argv) {  
  
    System<ConsoleLogger, PasswordSecurity, MacOSX, Enterprise> system;  
  
    system.print("Yihaa!");  
  
    std::cout << "Password accepted: " << system.checkPassword("*****")  
        << std::endl;  
  
}
```

Benutzer kann eine konkrete
Implementierung
"zusammenmischen"

Und das Verhalten der Instanz
wird dadurch flexibel
kombiniert und konfiguriert



Die C++ **Standard Template Library** (STL) macht ausgiebigen
Gebrauch von Mixins



Brüder von Funktionszeigern und Funktoren

METHODENZEIGER UND LAMBDAS

[Exkurs] Methodenzeiger: Beispiel



```
class ConsoleLogger {
```

```
    ConsoleLogger();
```

```
    ~ConsoleLogger();
```

```
    inline void print(const string& message) const {  
        cout << "user:~ /$" << message << endl;  
    }  
};
```

```
void main() {
```

```
    void (ConsoleLogger::*fp3)(const string&) const =  
        &ConsoleLogger::print;
```

```
    ConsoleLogger logger;
```

```
(logger.*fp3)("bar"); // user:~ /$ bar
```

```
}
```

Normale Methode
einer Klasse

Methodenzeiger sind
spezielle Funktionszeiger

Beim Zeiger auf Methoden muss die
Klasse als "Scope" angegeben werden

Aufruf **nur** mit einer Instanz
der Klasse möglich

[Exkurs] Methodenzeiger: Syntax



Klasse der
Methode

Name der
Variable

Liste der **Parametertypen**
der Funktionen, auf die
gezeigt werden soll

```
void (ConsoleLogger::*fp1)(const string&) =  
&ConsoleLogger::print;
```

Typ des
Rückgabewerts

Zeigertyp, Klammern
sind notwendig um
Rückgabetyp und Zeiger
auseinanderzuhalten

Adresse der Methode

```
ConsoleLogger logger;  
ConsoleLogger *loggerPtr;  
(logger.*fp3)("bar");  
(loggerPtr->*fp3)("bar");
```

Aufruf über
Dereferenzierung des
Methodenzeigers

[Exkurs] Funktionszeiger vs. Methodenzeiger



```
class C {  
public:  
    template<typename S>  
    void print(const S& s) { /* ... */}  
    void validateAges(int a) { /* ... */}  
};
```

```
template<typename C, typename F, typename T>  
void applyToSequence(C object, F method, T* begin, T* end) {  
    while (begin != end)  
        (object.*method)(*begin++);  
}
```

```
int main() {  
    int n[] = { -1, 20, 33, 120 };  
    applyToSequence(print<int>, n, n + 4);  
    applyToSequence(validateAges, n, n + 4);  
  
    applyToSequence(C(), &C::print<int>, n, n + 4);  
    applyToSequence(C(), &C::validateAges, n, n + 4);  
}
```

Zeiger auf **Methoden** können nicht auf die gleiche Art und Weise übergeben werden

... entsprechend ändert sich der Aufruf.

[Exkurs] Automatische Typableitung



- C++-Typen können **komplex** werden
 - `std::vector<std::string>::const_iterator x = v.begin();`
 - `const std::string& (*fp)(const std::string&);`
- **Neues Schlüsselwort auto** macht das Leben einfacher
 - `auto x = v.begin();`
 - `for (auto x : v) {std::cout << x << std::endl;}`
- In der Klausur aus didaktischen Gründen **verboten** ☺

[Exkurs] Lambdas (C++11)



- **Lambda-Ausdruck** = anonyme Funktion (ohne zugewiesenen Namen)

- **C++11:**

- Weiterer Mechanismus, um "Verhalten als Parameter zu übergeben"
 - In Kombination mit `auto` extrem mächtig und zugleich kompakt!

- Beispiel:

```
auto prefix = "Info: ";
auto print = [=] (const std::string &msg)
    {std::cout << prefix << msg << std::endl;};
print("Hello World!"); // Output: Info: Hello World!
```

- Mittels `[]` kann die Variable `prefix` aus dem Kontext von `print` "**eingefangen**" werden (`[=]` "by value", `[&]` "by reference")

- **In Java seit 1.8**

- Beispiel: `Arrays.asList(1,2,3).stream()
 .map(x -> x*x)
 .filter(x -> x < 7).collect(Collectors.toString());`

z.B. <http://www.cprogramming.com/c++11/c++11-lambda-closures.html>
viele Beispiele: https://en.wikipedia.org/wiki/Anonymous_function



TECHNISCHE
UNIVERSITÄT
DARMSTADT

MAKEFILES

[Exkurs] Makefiles: Motivation



- Indem wir Eclipse/CodeLite/... verwenden, **binden wir uns an diese IDE**.
- Tatsächlich gab es früher gar keine so mächtigen IDEs wie heute ...
- ... aber trotzdem große C/C++-Projekte mit **hunderten von Dateien/Klassen und noch mehr Abhängigkeiten**.

?

Wie soll man da den Überblick bewahren?

!

Mittels Regeln!

1 Source File *x.cpp*

→ 1 Object File *x.o*

n Object Files *x1.o x2.o x3.o ...*

→ 1 Executable *main.exe*

Target

Abhängigkeiten

Makefile

all: main.exe

main.exe: main.o Building.o Floor.o #...

g++ \$^ -o \$@

% .o: %.cpp

g++ -MMD -MP -c \$< -o \$@

Befehl, um Target zu bauen

1 Tab Einrückung zur Gruppierung von Befehlen

[Exkurs] "Make is an expert system." [1]



- **Eingabe:** Regelmenge (fix) + Zustand des Workspaces (variabel)
- **Ausgabe:** Notwendige Buildschritte (z.B. "Erzeuge main.o, Erzeuge prog")

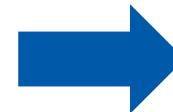
```
OBJ = main.o parse.o
prog: $(OBJ)
    $(CC) -o $@ $(OBJ)
main.o: main.c parse.h
    $(CC) -c main.c
parse.o: parse.c parse.h
    $(CC) -c parse.c
```



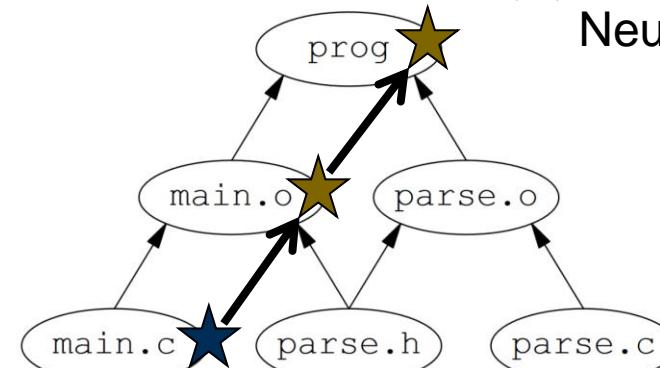
Makefile (Regelmenge)

★ Veränderung
★ Notwendige Neuberechnung

Project
└── Makefile
└── main.c ★
└── parse.c
└── parse.h



Workspace



Directed Acyclic Graph

[1] Miller, P.A. (1998), "Recursive Make Considered Harmful," AUUGN Journal of AUUG Inc., 19(1), pp. 14-25.

[Exkurs] Makefiles: Struktur



```
srcs = $(wildcard *.cpp)  
objs = $(srcs:.cpp=.o)  
deps = $(srcs:.cpp=.d)
```

```
all: main.exe
```

```
main.exe: $(objs)  
g++ $^ -o $@
```

```
%.o: %.cpp  
g++ -MMD -MP -c $< -o $@
```

```
clean:  
rm -rf $(objs) $(deps) main.exe
```

```
-include $(deps)
```

Erzeugt Listen aller Impl-Dateien und der entsprechenden *Object Files*.

Erstes Target ist immer der **Default-Einstiegspunkt**. Eclipse will *all*.

Platzhalter: \$^ - Abh.; \$@ - Target

"Suffixregel"; \$< - Input; \$@ - output

Löschen-Regel

Include-Dependencies (später)



[Exkurs] Makefiles: Ablauf

```
srcs = $(wildcard *.cpp)
objs = $(srcs:.cpp=.o)
deps = $(srcs:.cpp=.d)
```

```
all: main.exe
```

```
main.exe: $(objs)
g++ $^ -o $@
```

```
%.o: %.cpp
g++ -MMD -MP -c $< -o $@
```

```
clean:
```

```
rm -rf $(objs) $(deps) main.exe
-inlude $(deps)
```

1. Damit ich *all* erfüllen kann, brauche ich *main.exe*.

2. Falls ich kein *main.exe* habe, brauche ich alle .o-Dateien, um *main.exe* daraus zu linken.

3. Falls eine der .o-Dateien neuer ist als *main.exe*, muss ich *main.exe* trotzdem neu bauen.

4. Analog läuft es für die Kompilierung der .o-Dateien.

[Exkurs] Makefiles: Include-Dependencies



```
srcs = $(wildcard *.cpp)
objs = $(srcs:.cpp=.o)
deps = $(srcs:.cpp=.d)

all: main.exe

main.exe: $(objs)
    g++ $^ -o $@

%.o: %.cpp
    g++ -MMD -MP -c $< -o $@

clean:
    rm -rf $(objs) $(deps) main.exe

    -include $(deps)
```

- Wo sind eigentlich die **Header**?
- Wenn sich ein Header ändert, müssen **alle abhängigen Dateien** (mit `#include` des Headers) neu gebaut werden.
- Dazu dienen die Flags **-MMD -MP** und **-include \$(deps)**.

z.B.

Building.d

```
Building.o: Building.cpp Floor.hpp Person.hpp #...
# nop

Floor.hpp:
# nop

Person.hpp
# nop
```

[Exkurs] Makefiles: .PHONY



- **Ziel = Dateinamen(smuster)** (bspw. main.exe)
 - In unserem Beispiel: verletzt durch Ziele all und clean
 - Kein Problem, solange es keine Datei mit namen *all* oder *clean* gibt – andernfalls würde keines der Recipes ausgeführt werden, da zumindest *clean* keine Vorbedingungen hat
- **Lösung:** .PHONY-Deklaration

```
# Declares that targets 'all' and 'clean'  
# shall always be executed  
  
.PHONY: all clean  
  
clean:  
    rm -rf $(objs) $(deps) main.exe
```

https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html

[Exkurs] Makefiles: Fazit



- **Buildtools** sind ab einer bestimmten Projektgröße **unabdingbar**.
- Makefiles erlauben **inkrementelles Bauen von Projekten**...
 - ... müssen aber gepflegt werden und sind **nicht-trivial zu erlernen**.
- **Alternativen:** Makefile-Generatoren und andere Buildtools
 - *cmake, qmake*: Generatoren für Makefiles (letzterer von Qt)
 - *Ant, Maven, Ivy, Gradle*: ... eher für Java gedacht



TECHNISCHE
UNIVERSITÄT
DARMSTADT

MEHRFACHVERERBUNGSPROBLEME IN JAVA

[Exkurs] Mehrfachvererbung in Java? (I)



- **Frage:** Wie wird in Java die folgende Situation gelöst?
- **Antwort:** Gar nicht – darf so nicht vorkommen!
- **Mögliche Lösung:** InterfaceA separate implementieren und die Impl. in MyClass einbetten – dadurch ist MyClass aber kein Untertyp von InterfaceA mehr.

```
interface InterfaceA {      int run();      }  
interface InterfaceB {      boolean run();     }  
  
public class MyClass implements InterfaceA, InterfaceB {  
  
    @Override  
    public int run() {  
        return 0;  
    }  
}  
  
Error: The return type  
is incompatible with  
InterfaceB.run()
```

[Exkurs] Mehrfachvererbung in Java? (II)



- Seit Java 1.8: Statische Methoden mittels **default** in Interfaces mögliche
- ... und damit auch neue Probleme ☹

```
interface InterfaceA {      default int run() { return 0; }      }
interface InterfaceB {      default int run() { return 1; }      }
```

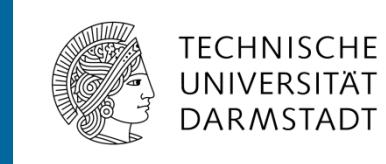
```
public class MyClass implements InterfaceA, InterfaceB {
}
```

Error: class MyClass inherits
unrelated defaults for run() from
types InterfaceA and InterfaceB



Programmierpraktikum C und C++

Technische Anmerkungen



Dr.-Ing. Eric Lenz
elenz@iat.tu-darmstadt.de

Fachgebiet Control and Cyber-Physical Systems (CCPS)

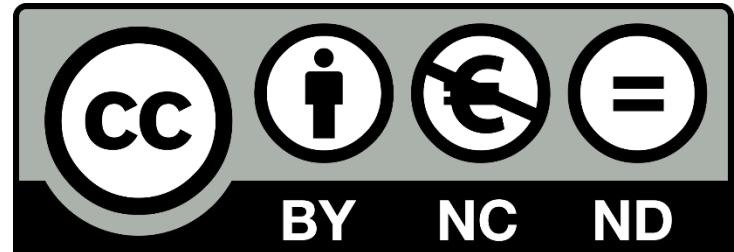
Prof. Dr.-Ing. Rolf Findeisen
Dept. of Electrical Engineering and Information Technology
<https://www.ccps.tu-darmstadt.de/>

Lizenz

Dieses Werk ist lizenziert unter einer
Creative Commons Namensnennung - Nicht
kommerziell - Keine Bearbeitungen 4.0
International Lizenz

<http://creativecommons.org/licenses/by-nc-nd/4.0/>

Die Logos der TU Darmstadt und des
Fachgebiets CCPS unterliegen der Fair-
Use-Konvention.



Beteiligte Autoren (alphabetisch):
Anthony Anjorin,
Sebastian Ehmes,
Matthias Gazzari,
Nicolas Himmelmann,
Puria Izady,
Philipp Joncyk,
Roland Kluge,
Maurice Rohr

Bildnachweis und Credits

- **Titelbild "Organisatorisches" (Papierstapel):** CC BY-SA 3.0, by Jonathan Joseph Bondhus on Wiki Commons, URL: https://commons.wikimedia.org/wiki/Paper#/media/File:Stack_of_Copy_Paper.jpg
- **Lächelndes Fragezeichen:** "attribution", by katieyunholmes: smiley face clip art animated, URL: <http://cliparts.co/clipart/2613703>
- **Fotos des Experimentierboards:** CC BY-SA 3.0, Roland Kluge 2017, Real-Time Systems Lab
- **Code-Highlighting:** Danke an <https://tohtml.com/c/>
- **Online-Beispiele:** Danke an <http://cpp.sh/>