

Programmierpraktikum C und C++



```
#include <iostream>

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



ES Real-Time Systems Lab

Prof. Dr. rer. nat. Andy Schürr

Dept. of Electrical Engineering and Information Technology

Dept. of Computer Science (adjunct Professor)

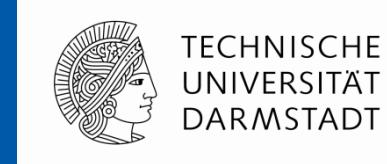
www.es.tu-darmstadt.de

Roland Kluge

roland.kluge@es.tu-darmstadt.de

Programmierpraktikum C und C++

Organisatorisches



ES Real-Time Systems Lab

Prof. Dr. rer. nat. Andy Schürr

Dept. of Electrical Engineering and Information Technology

Dept. of Computer Science (adjunct Professor)

www.es.tu-darmstadt.de

Roland Kluge

roland.kluge@es.tu-darmstadt.de



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

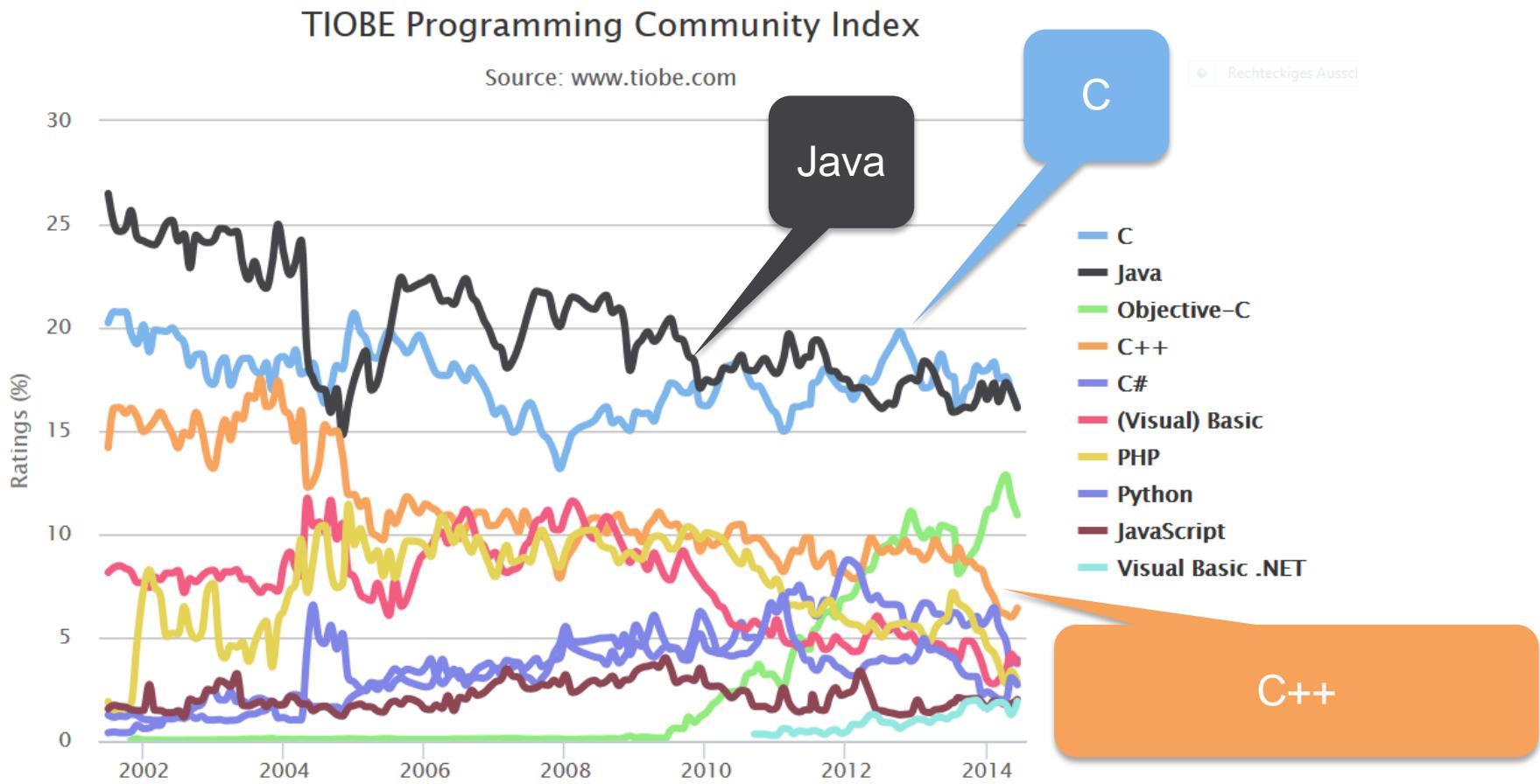
Idee des Praktikums

- Vorlesung vermittelt Konzepte
- Übung vermittelt praktische Kenntnisse

Basisvoraussetzungen

- Allgemeine Programmiererfahrung
- Kenntnisse in Java werden

Wie wichtig sind C und C++?



Organisatorisches



Jeden Tag

09:00 – 11:00: Frontalunterricht im Hörsaal

11:00 – 16:00: praktische Übungen im Pool

Bitte **aktiv** Hilfe fordern
während der Übung!

Anwesenheitspflicht

Ausnahmen persönlich genehmigen lassen (Klausur, Krankheit)

Wer **mehr als 2 Kontrollen** fehlt (**egal wieso**), darf nicht an der Klausur teilnehmen!

Ansprechpartner

Roland Kluge (roland.kluge@es.tu... und via Moodle)

Matthias Gazzari (während der Übungen)

Eugen Lutz (während der Übungen)

Klausur



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Termin

Datum: Dienstag, 13.10.2015
Uhrzeit: 16:15 – 18:15 (Bearbeitungszeit: 90 Minuten)
Raum: S1|01 A03 (+ evtl. A04)

Inhalt

Tag 1 – Tag 4: C++-Programmierung mit Eclipse CDT

Tag 5 – Tag 6: C-Programmierung für Microcontroller

Tage 5 und 6 sind
NICHT klausurrelevant

Vorbereitung

1. Konzepte der Vorlesung verstehen
2. Übungen aus dem Praktikum selbstständig lösen

Zur Teilnahme erforderlich

1. amtlicher Lichtbildausweis
2. Klausuranmeldung (TUCaN!)

Vorlesungs- und Übungsbetrieb



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Übung (nachmittags) im Raum 67

Virtuelle Maschine: <http://tiny.cc/es-cpp-vm>

(User: cppp, PW: >Cppp2015<)

Material

- Vorlesung <https://github.com/Echtzeitsysteme/tud-cpp-lecture>
- Übung <https://github.com/Echtzeitsysteme/tud-cpp-exercises>

Eigenes Projekt erstellen mit Git:

Einführung in Git: <http://git-scm.com/book/de>

Kostenfreie Git-Repositories auf <https://github.com/>

Fachliche Fragen bitte IMMER im Moodle:

<https://moodle.tu-darmstadt.de/course/view.php?id=4827>

Literaturvorschläge – Bücher



Bruce Eckel: Thinking in C++, Volumes One and Two

(frei verfügbar online <http://mindview.net/Books/TICPP/ThinkingInCPP2e.html>)

Scott Meyers: Effective C++

Scott Meyers: More Effective C++

Helmut Schellong: Moderne C Programmierung [Springer]

Ralf Schneeweiß: Moderne C++ Programmierung [Springer]

Jürgen Wolf: Grundkurs C [Galileo]

Jürgen Wolf: Grundkurs C++ [Galileo]

Kompakt und günstig

Bjarne Stroustrup: Einführung in die Programmierung mit C++

Literaturvorschläge – Skripte / Online



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Grundkurs C/C++ @ TU München

<http://www.ldv.ei.tum.de/lehre/programmierpraktikum-c/>, <http://www.ldv.ei.tum.de/lehre/grundkurs-c/>

Sehr umfangreiches Material

Programmieren 1 @ FH Regensburg

<http://fbim.fh-regensburg.de/~sce39014/pg1/pg1-skript.pdf>

Grundlagen (Schleifen, etc.)

LearnCpp.com

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

Alternative Veranstaltungen an der TU Darmstadt



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- 20-00-0747-pr Modern C++ Programming (zuletzt 2014)
<https://www.informatik.tu-darmstadt.de/de/fachbereich/lehrbeauftragte-und-gastdozenten/modern-c-programming/>
- 18-ad-1020-vl Programmierung in der Automatisierungstechnik in
C/C++ (V1+Ü1)
http://www.rmr.tu-darmstadt.de/lehre_rmr/vorlesungen_rmr/wintersemester/programmierung_aut/index.de.jsp

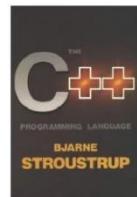
C, C++ und Java



TECHNISCHE
UNIVERSITÄT
DARMSTADT



C „1.0“
(1972)



C++ „1.0“
(1980~85)

ANSI C/C89 (1989)
„Programming
Language C“

C95 (1995)

C99 (1999)

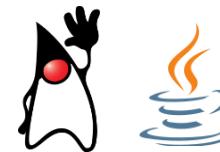
C11 (2011)

C++98 (1998)

C++03 (2003)

C++11 (2011)

C++14 (2014)



Java 1.0 (1996)

Java 1.5 (2004)

Java SE 6 (2006)

Java SE 7 (2011)

Java SE 8 (2014)



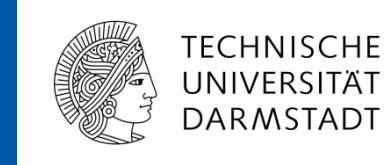
Fragen?



<http://cliparts.co/clipart/2613703>

Programmierpraktikum C und C++

Grundlagen



ES Real-Time Systems Lab

Prof. Dr. rer. nat. Andy Schürr

Dept. of Electrical Engineering and Information Technology

Dept. of Computer Science (adjunct Professor)

www.es.tu-darmstadt.de

Roland Kluge

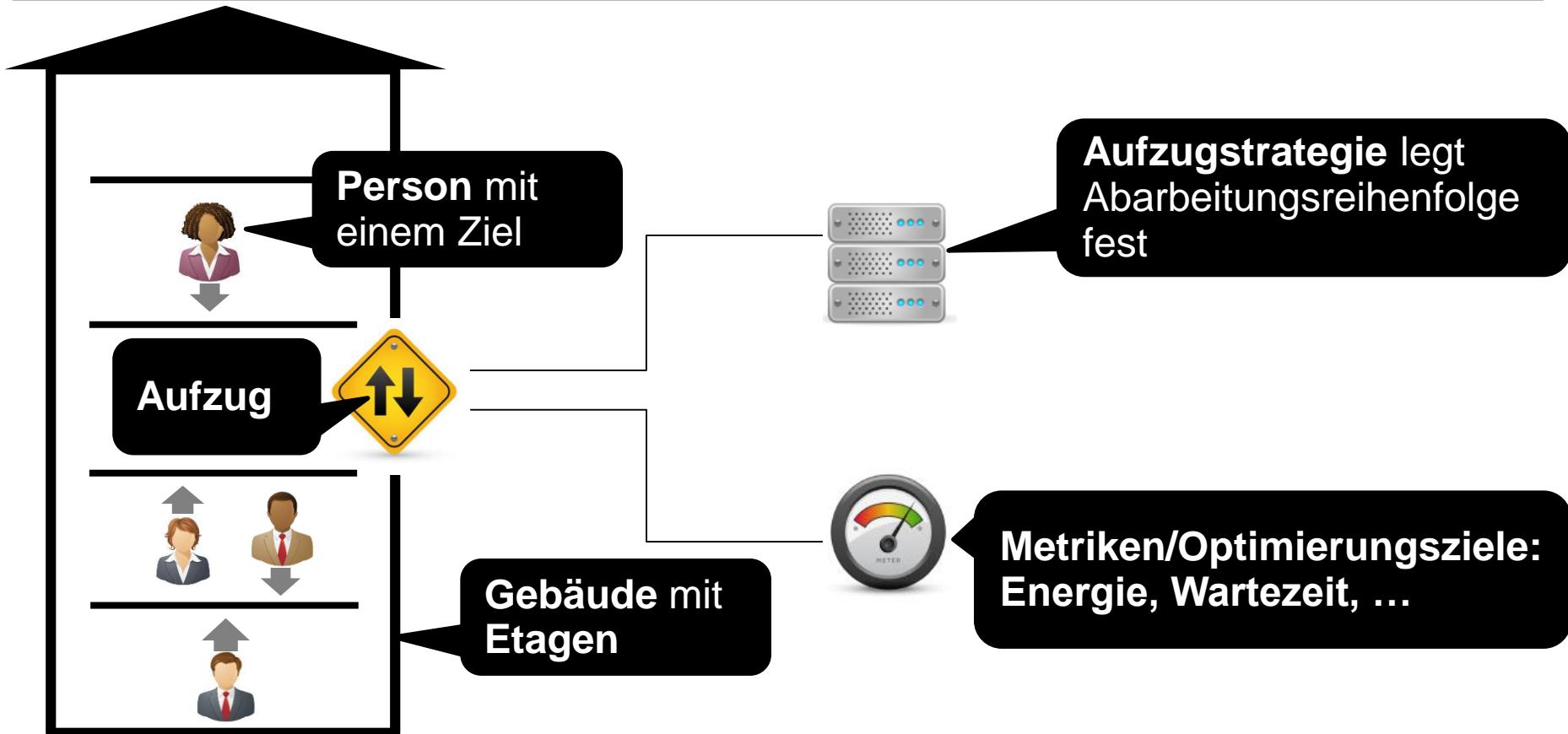
roland.kluge@es.tu-darmstadt.de



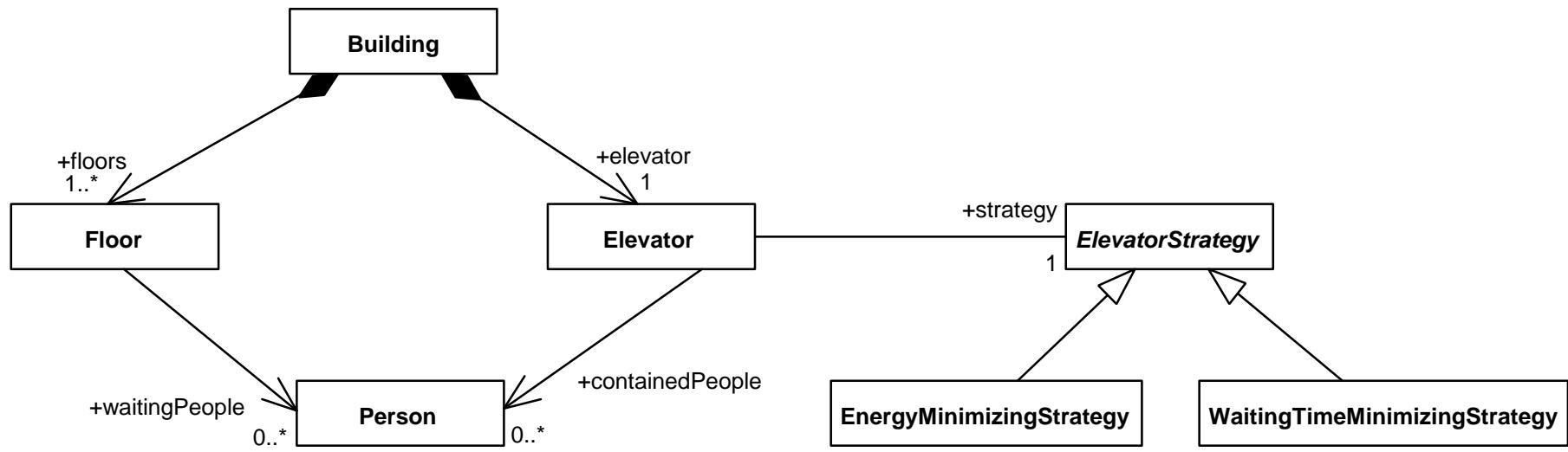
TECHNISCHE
UNIVERSITÄT
DARMSTADT

LAUFENDES BEISPIEL

Laufendes Beispiel: Implementierung einer Aufzugsimulation



Laufendes Beispiel: Klassendiagramm





Vergleich zwischen Java und C++

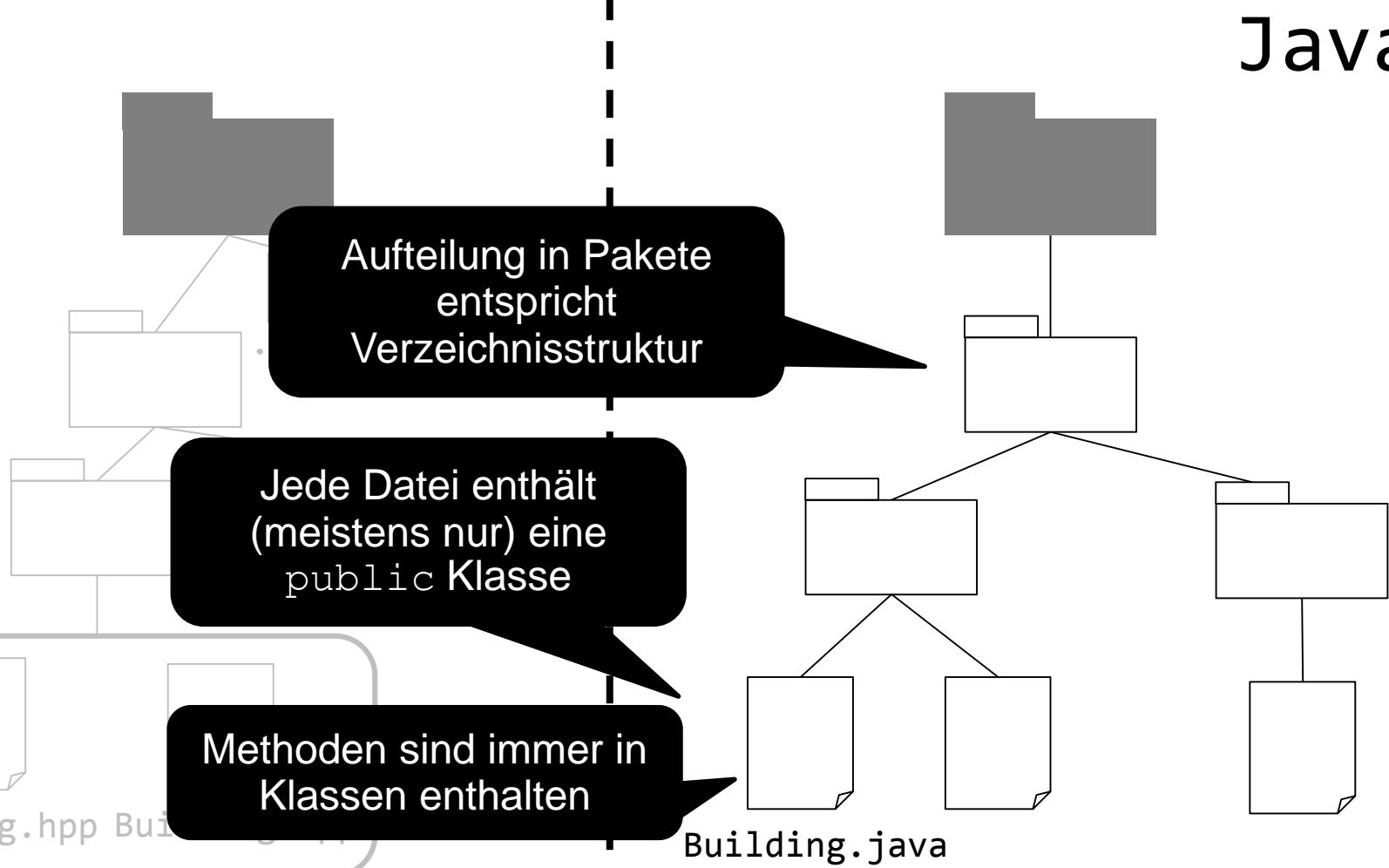
PROJEKTSTRUKTUR

Projektstruktur



C++

Java



Intermezzo



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Wie implementiert man Funktionen in Java?

Ist es sinnvoll, die Paketstruktur an der Verzeichnisstruktur zu binden?

Darf man in Java mehrere Klassen in einer Datei implementieren?

Hier seid Ihr gefragt! 😊



<http://cliparts.co/clipart/2613703>

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
kombiniert werden

Building.hpp Building.cpp

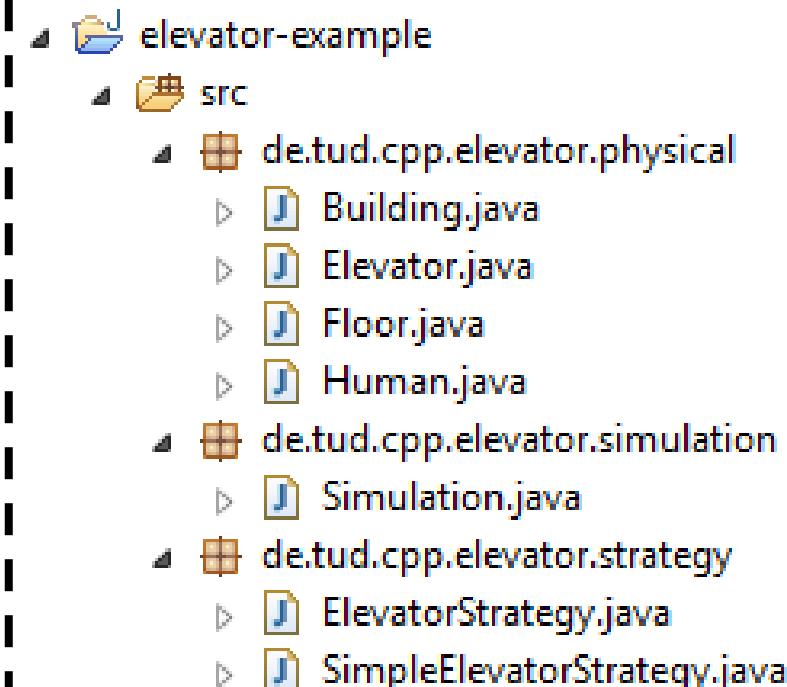
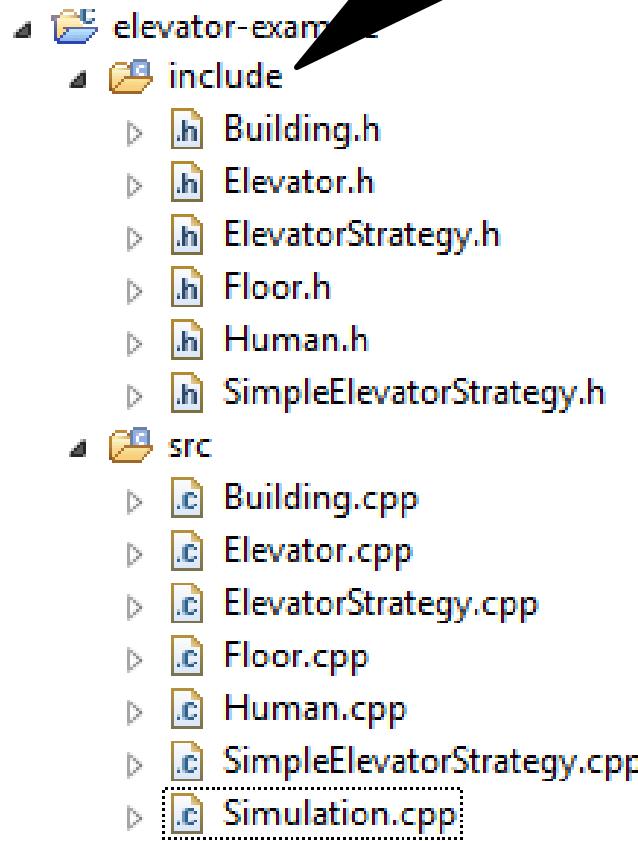
Projektstruktur



C++

Getrennte Ordner für Header
und Implementierung

Java



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 number_of_floors);
    ~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,
"..." für eigenen Code

Deklaration der Klasse ist
wie ein Interface in Java

Header und Implementierungs-Dateien



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

```
#include "Building.hpp"
```

```
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;
}
```

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

Header-Datei wird eingebunden

Methoden werden implementiert
(Details später)

Intermezzo



TECHNISCHE
UNIVERSITÄT
DARMSTADT

(Warum) Ist die Trennung in Header- und Implementierungsdateien hilfreich?



<http://cliparts.co/clipart/2613703>

Exkurs: C++-Referenzen



TECHNISCHE
UNIVERSITÄT
DARMSTADT

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

The screenshot shows the cplusplus.com website. The navigation bar includes a search bar and a 'Go' button. Below the search bar, the 'Reference' tab is selected, and the URL '<iostream>' is shown. 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 has '<iostream>' selected), and other items like '<fstream>', '<iomanip>', etc. The main content area is titled 'header <iostream>' and describes it as the 'Standard Input / Output Streams Library'. It notes that it defines standard input/output streams and that including this header may automatically include '<iosfwd>'. A note at the bottom says that the 'iostream' class is mainly declared in 'ios' and 'iosfwd'. There are also sections for 'Objects' and 'Related pages'.

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

The screenshot shows the en.cppreference.com website. The title is 'C++ reference' with subversion 'C++98, C++03, C++11, C++14'. The left sidebar lists 'ASCII chart', 'Compiler support', 'Language' (with sub-links for Preprocessor, Keywords, Operator precedence, Escape sequences, and Fundamental types), 'Headers', 'Concepts', 'Utilities library' (with sub-links for Type support, Dynamic memory management, Error handling, Program utilities, Date and time, bitset, Function objects, pair - tuple (C++11), and integer_sequence (C++14)), 'Containers library' (with sub-links for array (C++11), vector - deque, list - forward_list (C++11), set - multiset, map - multimap, unordered_set (C++11), unordered_multiset (C++11), unordered_map (C++11), unordered_multimap (C++11), and stack - queue - priority_queue), 'Algorithms library', 'Iterators library', and 'Numerics library' (with sub-links for Common mathematical functions, Complex numbers, and Pseudo-random number generation).

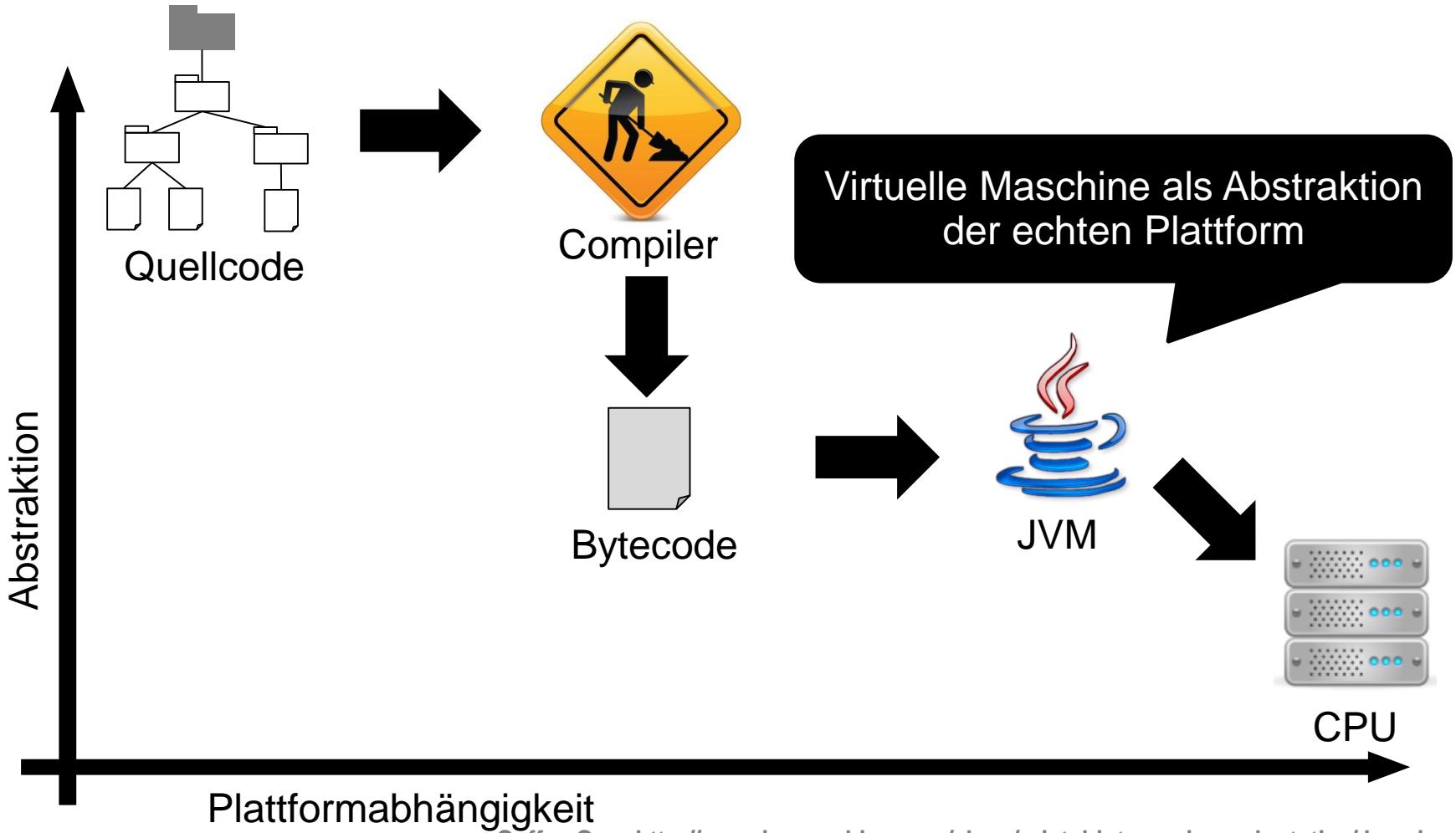


TECHNISCHE
UNIVERSITÄT
DARMSTADT

KOMPILEIERUNG

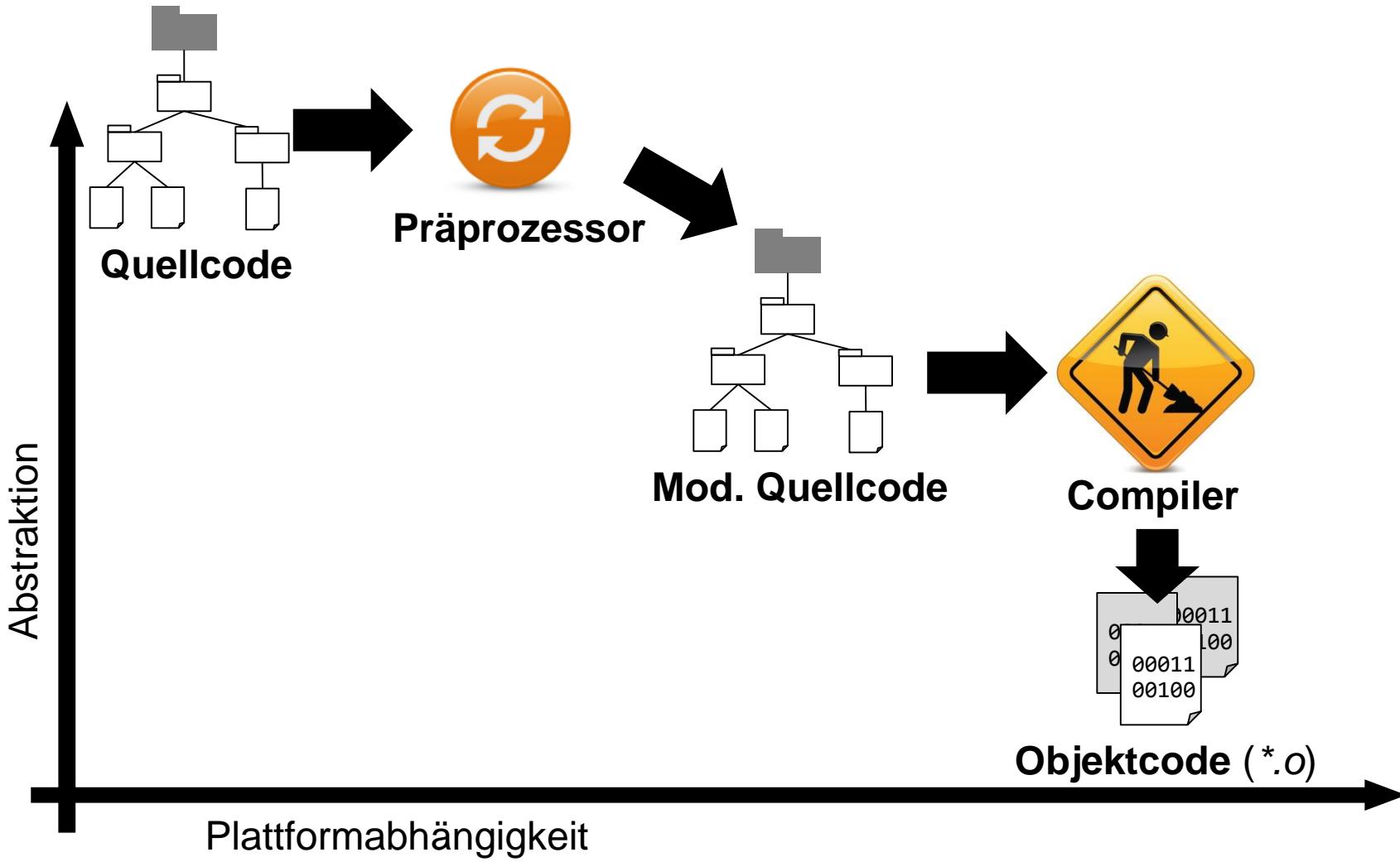


Kompilierung in Java

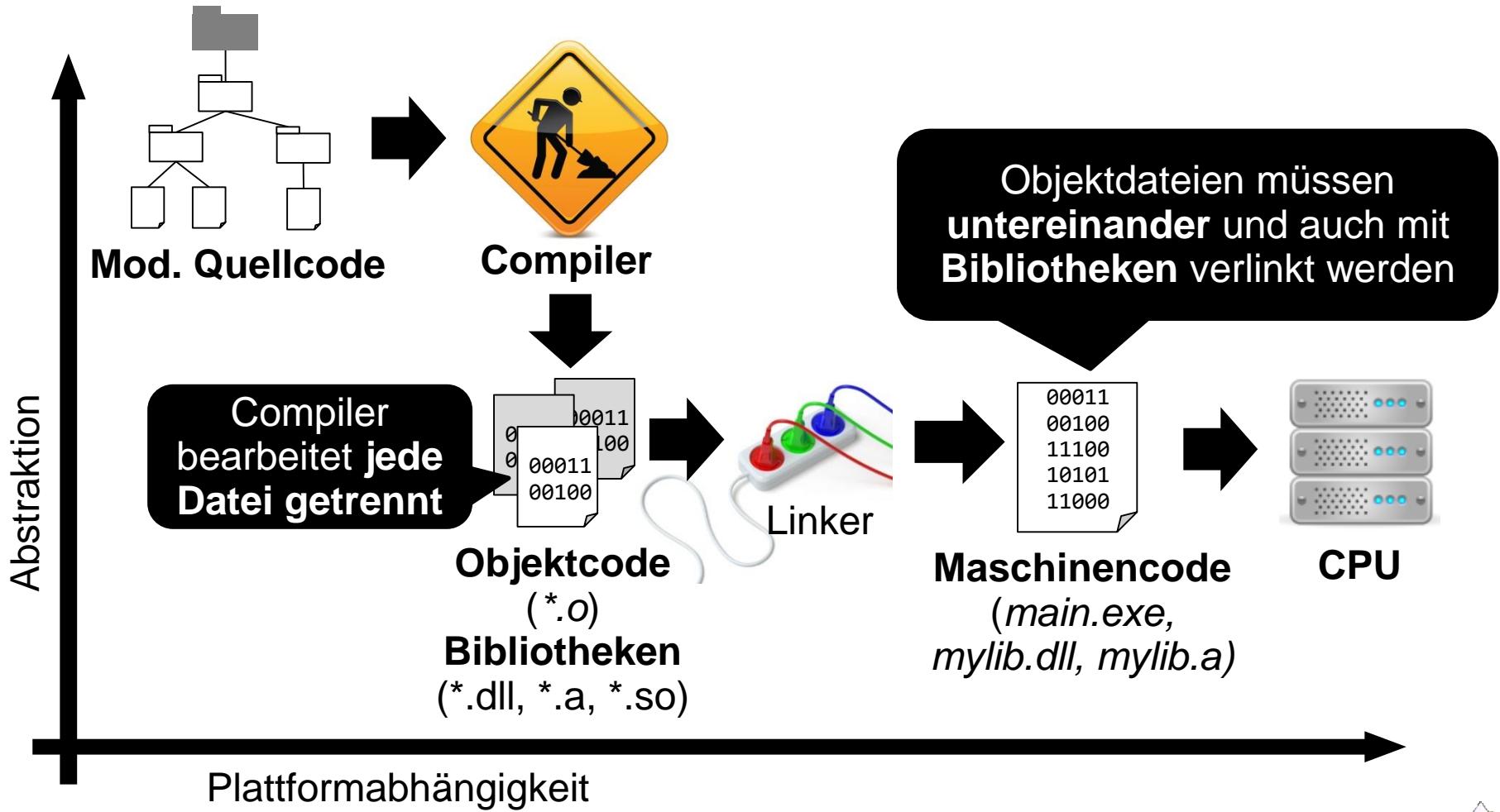


Coffee Cup: <http://www.iconarchive.com/show/cristal-intense-icons-by-tatice/Java-icon.html>

Kompilierung für C/C++ I



Kompilierung für C/C++ II





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 **Laufzeit** vorhanden sein.
- *DLLs* müssen **nicht** zur **Linkzeit** und **nur beim konkreten Aufruf** zur **Laufzeit** verfügbar sein.

→ Compilat ist “minimal”, braucht aber zur Laufzeit zusätzliche Abhängigkeiten

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 davor, dass *Building.h* mehrmals eingebunden wird
(Alternative: `#pragma once`)

Diese Konvention macht es möglich, ohne Bedenken immer **alle benötigten Header überall einbinden** zu können

Weitere Anwendungsfälle des Präprozessors:

- Unterscheidung zwischen Debug- und Release-Build (z.B. Logging)
- Betriebssystemerkennung (z.B. WIN32, UNIX)
- (in älteren C++-Varianten): Konstanten definieren

Exkurs: 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;
};
```

Floor.hpp

- **inline** zeigt an, dass statt eines Methoden-/Funktionsaufrufs direkt der Code an jeder Aufrufstelle eingefügt werden soll.
- Nur ein **Hinweis** an den Compiler – nicht “verpflichtend”.
- Heute **nicht mehr notwendig**, da der Compiler automatisch über Optimierungen entscheidet (Flags -O1, -O2, -O3, ...)

[EN]  /Inline_function

Intermezzo



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Wie ist es möglich, dass man erfolgreich **kompilieren**
aber **nicht linken** kann?

Wozu braucht man einen **Präprozessor**?

Gibt es bei anderen Sprachen ebenfalls einen
Präprozessor?

Welche Konsequenzen zieht eine Änderung an **inline**-
Methoden (im Header) nach sich im Vergleich zu
Änderungen in der Impl-Datei?



<http://cliparts.co/clipart/2613703>



TECHNISCHE
UNIVERSITÄT
DARMSTADT

PROGRAMMSTART

Systemstart



- main-Funktion in C++ entspricht main-Methode in Java.
- Zwei Formen:
 - parameterlos
 - mit Kommandozeilenparametern (argv[0] enthält Pfad zum Programm)

```
#include "Building.hpp"

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

Kein Rückgabewert
(= return 0; = alle OK)

```
#include <iostream>
#include <cstdlib>
#include "Building.h"

int main(int argc, char **argv){
    if (argc >= 2) {
        unsigned int levels = std::atoi(argv[1]);
        Building hbi(levels);
        building.runSimulation();    }
}
```

Arrays – siehe Übung 1

Demo: Virtuelle Maschine



1. Herunterladen der VM:

- <http://tiny.cc/es-cppp-vm>
- User: cппп
- PW: >Cппп2015<

2. Importieren der Appliance *antergos.ova*

WICHTIG (f. Pool):

- Beim Importieren muss der Pfad für das **Virtuelle Plattenabbild** auf **C:/VM/praktikum_1/antergos-disk1.vmdk** gesetzt werden – ansonsten sprengt Ihr die **Quota**!
- Die VM wird **auf dem PC** und **nicht** in eurem Profil gespeichert!

3. Genereller Hinweis: *Ctrl (rechts)* ist die Host-Taste der VM → Kann zu Problemen bei Tastenkürzeln führen.

Im Pool:
\sam\Install\praktikum2.ova

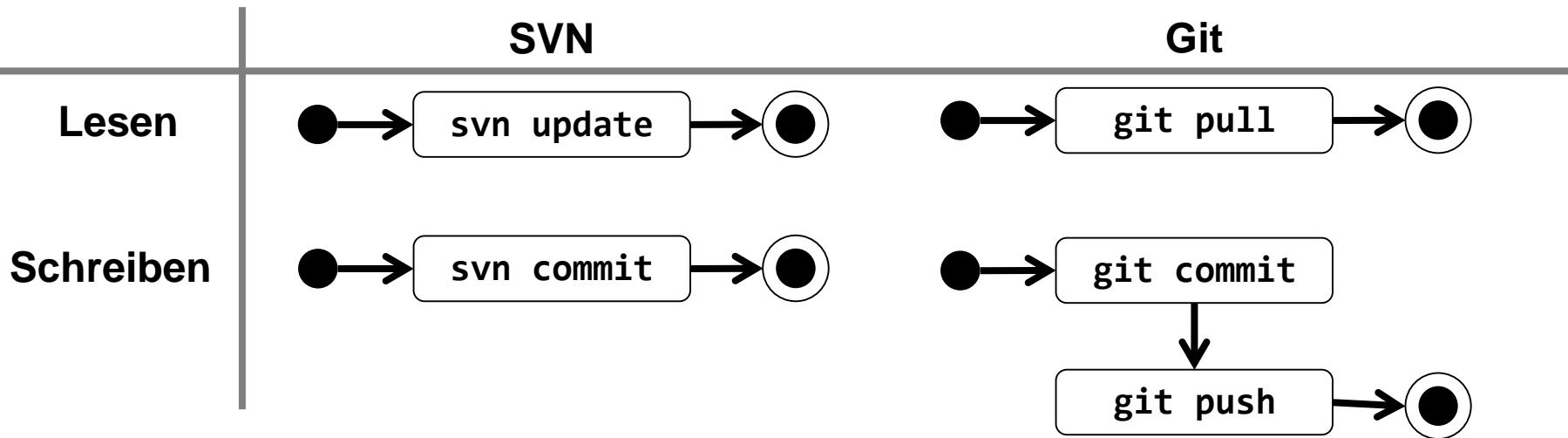
Beschreibung	Konfiguration
Virtuelles System 1	
Name	praktikum_1
Gast-Betriebssystem	Other/Unknown
CPU	1
RAM	2048 MB
DVD-Laufwerk	<input checked="" type="checkbox"/>
USB-Controller	<input checked="" type="checkbox"/>
Netzwerkkarte	<input checked="" type="checkbox"/> PCnet-FAST III (Am79C973)
Festplatten-Controller IDE	PIIX4
Festplatten-Controller IDE	PIIX4
Festplatten-Controller SATA	AHCI
Virtuelles Plattenabbild	C:\VM\praktikum_1\antergos-disk1.vmdk

Ein paar Worte zu Git



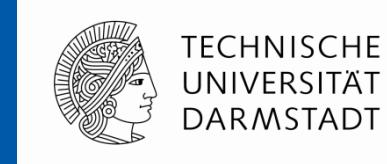
Bereitstellung der Vorlesungs- und Übungsunterlagen

- Bereits auf der VM ausgecheckt (*Window → Perspective → Open Perspective → Git*)
- Vorlesung <https://github.com/Echtzeitsysteme/tud-cpp-lecture>
- Übung <https://github.com/Echtzeitsysteme/tud-cpp-exercises>
- Informationen zu Git:
<https://github.com/Echtzeitsysteme/tud-cpp-exercises/blob/master/README.md>



Programmierpraktikum C und C++

Speicherverwaltung und Lebenszyklus



ES Real-Time Systems Lab

Prof. Dr. rer. nat. Andy Schürr

Dept. of Electrical Engineering and Information Technology

Dept. of Computer Science (adjunct Professor)

www.es.tu-darmstadt.de

Roland Kluge

roland.kluge@es.tu-darmstadt.de



Stack und Heap

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



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

Vier wesentliche Speicherbereiche

- **Programmspeicher**
Enthält den binären Programmcode (+ evtl. Debugging-Symbole); normalerweise read-only.
- **Globaler Speicher**
Enthält die globalen Variablen und Konstanten; für uns hier nicht so wichtig.
- **Heap-Speicher** (aka. dynamischer Speicher)
Frei verwendbar; Benutzer übernimmt Speichermanagement.
- **Stack-Speicher** (aka. statischer Speicher)
Verwendung für lokale Variablen; Speicherverwaltung durch Compiler.

Stack vs. Heap



Stack

- begrenzte Größe (lokale Variablen, Rücksprungadresse)
- Speicherbelegung und – freigabe durch den Compiler
- Speicherverwaltung:
last-in first-out

→ sehr effizient, statisch

Heap

- typ. wesentlich größer als Stack
- Speicherverwaltung:
durch “Benutzer”
(`new`, `delete`)

→ groß aber teuer (Laufzeit)

Intermezzo



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Wieso braucht man überhaupt Speicher auf dem Heap, wenn der Stack die **Speicherverwaltung** übernimmt und auch noch so **viel effizienter** ist?



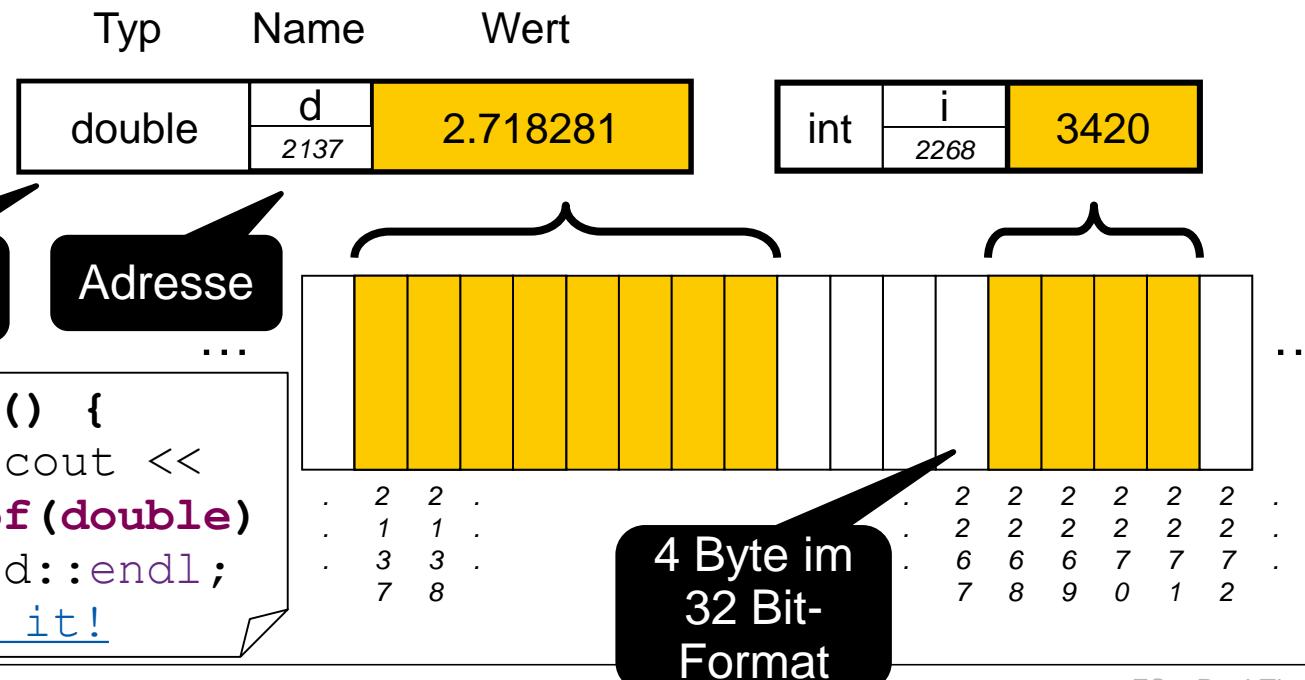
<http://cliparts.co/clipart/2613703>

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

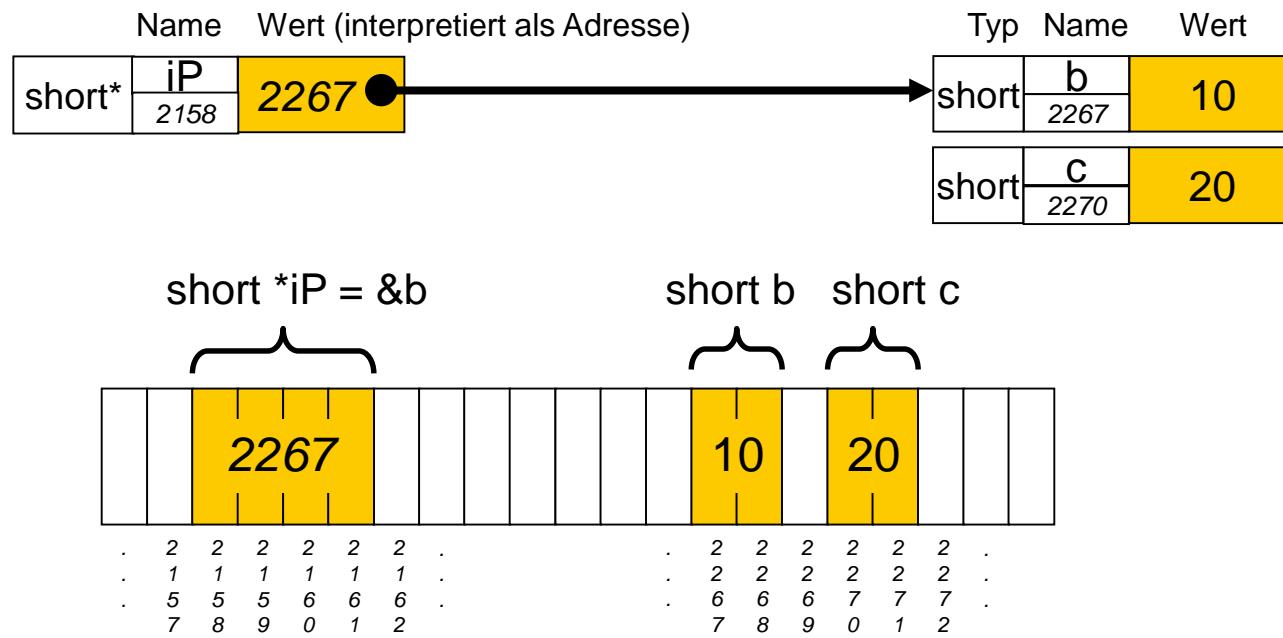
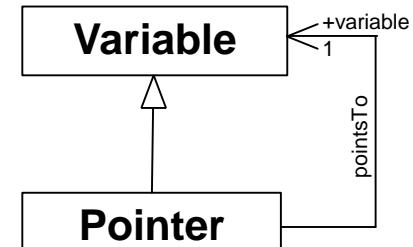


Variablen und Zeiger: Was ist ein Zeiger?



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



Variablen und Zeiger: Syntax



```
int i = 42;
```

Deklaration eines Zeigers vom Typ *int** (Zeiger auf *int*, hat strenggenommen keinen Wert)

```
int *iP;
```

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

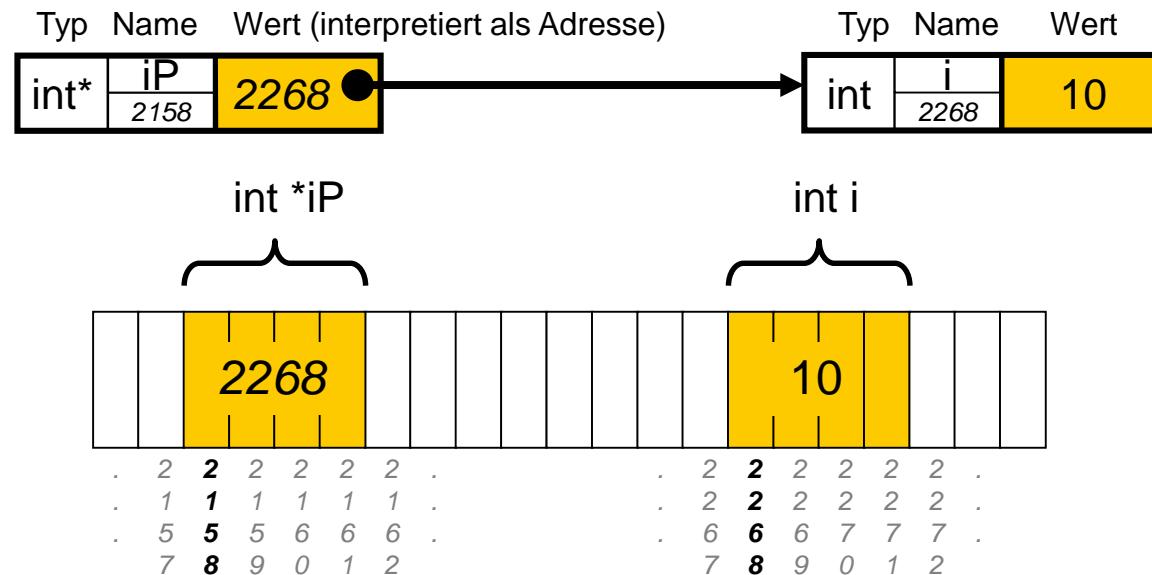
```
iP = &i;
```

Dereferenzierung eines Zeigers, um den Inhalt zu erhalten

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

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

Intermezzo: Pointer und Variablen



```
std::cout << i << std::endl;           10  
std::cout << iP << std::endl;         2268  
std::cout << &i << std::endl;          2268  
std::cout << *iP << std::endl;        10  
std::cout << &iP << std::endl;          2158
```





Der Null-Pointer



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

– 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;
```

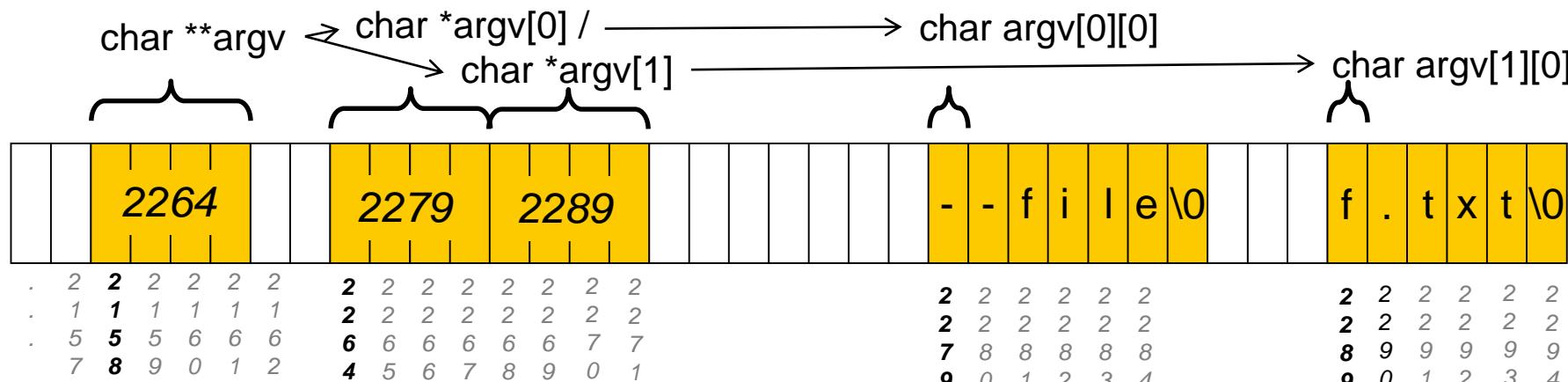
Wie **<stddef.h>**, aber mit Namespaces

0x0
NULL
nullptr

[EN]  /Null_pointer

```
int main(int argc, char** argv)
```

- Siehe: `int main(int argc, char** argv)`
 - Was passiert z.B. beim Aufruf `main.exe --file f.txt`
 - Strings (in C) sind Folgen von `char` (mit `\0` abgeschlossen)



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

2264
--file 2279
f.txt 2289

Spezieller operator<< für *char**

2279

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

void* = rischer“ Pointer

Intermezzo



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Wann braucht man wirklich Zeiger?

Wieso kann man nicht einfach nur normale
Variablen verwenden?



<http://cliparts.co/clipart/2613703>

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.

Was ist eine (C++)-Referenz?



Eine **Referenz** ist ein **Alias auf eine Variable** (braucht keinen eigenen Speicher). 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;
```

Verhält sich
wie Variable

const bei Objekten



```
class Building {  
public:  
    Building(int number_of_floors);  
    ~Building();  
  
    void printFloorPlan() const;
```

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

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

building darf
nicht verändert
werden

```
void iDoNotChangeAnything(const Building &building) {  
    building.printFloorPlan();  
}
```

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

Intermezzo

Wieso soll ich konsequent **const** verwenden?

Wann soll ich **const** verwenden und wann nicht?

Was ist der Unterschied zu **final** in Java?

Gibt es eigentlich einen Unterschied zwischen

`int* iP`

und

`int *iP`

und

`int * iP`

?



<http://cliparts.co/clipart/2613703>

Wieso **const**?



1. **Compiler** kann automatisch die Absichten des Programmierers **statisch** durchsetzen (es gibt einen guten Grund wieso etwas **const** sein soll!)
2. Compiler kann viele **Optimierungen** durchführen mit dem Wissen darüber, was **const** ist und was nicht
3. Absicht des Programms wird für den Leser „**expliziter**“.
4. Wird für **Objekte** und **Methoden** sinnvoll verallgemeinert

Intermezzo: const



```
const int numFloors;  
const Elevator &elevator;
```

Unveränderliches Attribut (-> Initialisierungsliste nötig!).

```
static const int MAX_FLOOR_COUNT = 3;
```

Konstante (innerhalb oder außerhalb einer Klasse)

```
const Elevator &Building::getElevator() const;
```

Methode, die eine unveränderliche *Elevator*-Instanz liefert (1. *const*) und die umgebende Klasse *Building* nicht verändert (2. *const*).

```
void readPerson(const Person *const person);
```

Funktionsparameter *person* als Pointer, der nicht neu zugewiesen werden kann (also kein *person = new Person()*, 2. *const*) und dessen Objekt nicht verändert werden kann (1. *const*).



<http://cliparts.co/clipart/2613703>

Intermezzo: * und &

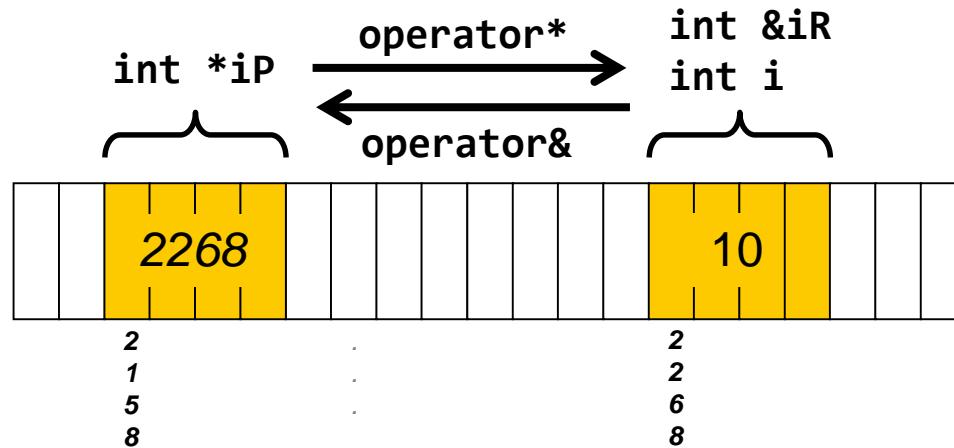
Welche „Rollen“ kann der Asterisk (*) im Code annehmen?

Welche „Rollen“ kann das Ampersand (&) im Code annehmen?



<http://cliparts.co/clipart/2613703>

Beispiel: 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>





<https://isocpp.org/wiki/faq>

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,...



(Copy-)Konstruktor und Destruktor

AUF- UND ABBAUEN VON OBJEKTEN

Konstruktor, Destruktor und Copy-Konstruktor



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

**Konstruktor mit
Initialisierungsliste
(Reihenfolge beachten!)**

Copy-Konstruktor

Destruktor

```
Floor::Floor(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;  
}
```

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(0);  
    iWorkOnACopy(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(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(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(0);  
    iUseAPointer(&floor);  
}
```

Äquivalent zu
(*floor).getNumber()

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

Intermezzo



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Wieso ist die Übergabe per *const&* ein
sinnvoller Default?

Wann ist die Übergabe per *const& nicht möglich?*

Wieso soll (sogar in vielen Fällen muss)
man die **Initialisierungsliste** verwenden?



<http://cliparts.co/clipart/2613703>

Assignment-Operator



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

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

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

void operator=(const EnergyMinimizingStrategy &a) {
cout << "operator= called" << endl;
}
};
```

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

Compiler-generierte Methoden



Der Compiler generiert automatisch eine Reihe von Methoden, falls sie **nicht vorhanden (=deklariert)** sind, z.B.:

- Default-Konstruktor `MyClass ()`
- Copy-Konstruktor `MyClass (const MyClass &a)`
- Assignment-Operator `void operator=(const MyClass &a)`
- Destruktor `~MyClass ()`
- Aber auch:
 - Initialisierungsliste
 - ...

Rule of Three



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

```
#include <fstream>

class AccessController {
public:
    AccessController() {
        logfile.open("logfile.txt");
    }
    // No copy constructor
    ~AccessController() {
        logfile.close();
    }

private:
    std::ofstream logfile;
};
```

Default Copy-Konstruktor kopiert *logFile*.

Aber: `std::ostream` hat **keinen**
Kopierkonstruktor!



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 1:1 kopiert werden (evtl. mittels „rekursivem“ Copy-Konstruktor).
- Wenn ich **Ressourcen** (Speicher, File Handle,...) in einem **Konstruktor** akquiriere, möchte ich sie auch im **Destruktor** freigeben.
- Verwende ich einen **eigenen Copy-Konstruktor** und einen **generierten Assignment-Operator**, kann es zu **inkonsistenten Verhalten** kommen.



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;  
}
```

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

Copy Elision



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

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

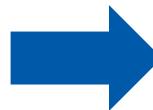
    inline void operator=(
        const EnergyMinimizingStrategy &a) {
        cout << "operator= called" << endl;
    }
};

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

[EN] /Copy Elision

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 😊.

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! Wer sieht ihn?

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]



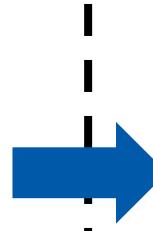
Extrem gefährlich!

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!

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



Creating floor [0]
Destroying floor [1]

Nach dem Löschen
immer auf NULL setzen!

Speicherlecks



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



Was wird hier
gelöscht?



```
Creating floor [0]  
Creating floor [1]  
Destroying floor [1]  
Destroying floor [5706624]
```

Es ist nicht mehr möglich, *floor [0]* freizugeben! Dies wird 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

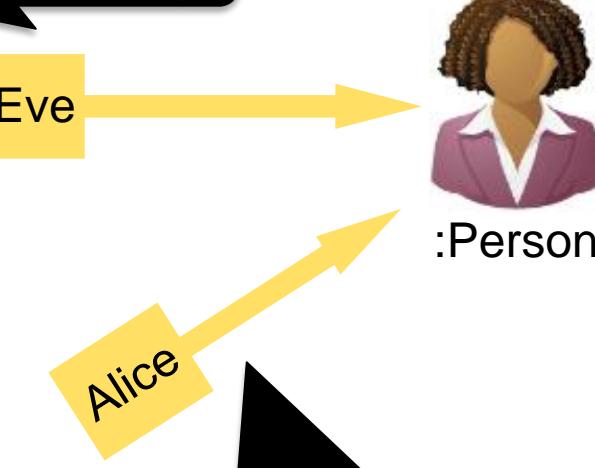


```
Person *Eve = new Person();  
Person *Alice = Eve;
```

„Rohzeiger“
(raw pointer)

Objekt auf dem Heap

Eve



Alice

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

Intermezzo

Wie könnte man das Problem lösen? Wir müssen ja irgendwie entscheiden wann ein Objekt gelöscht werden darf ...



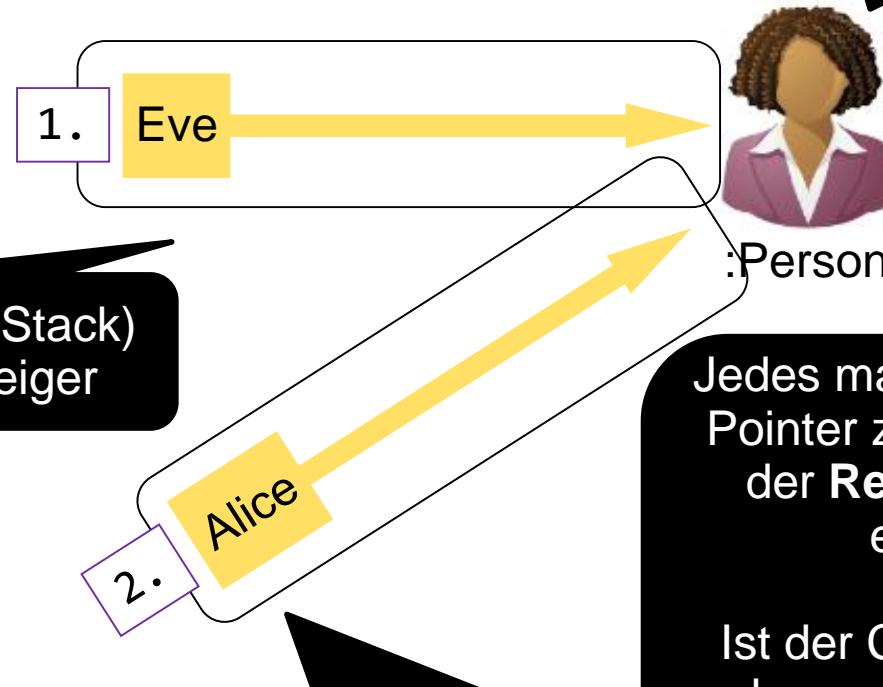
<http://cliparts.co/clipart/2613703>

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 – vorher (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

Mit klassischen Zeigern



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

#include "Person.hpp"

void makeSmallTalkWith(const Person &person){
    cout << "Isn't the weather quite pleasant today, "
        << person.getName() << "?" << endl;
}

void greet(const Person &person){
    cout << "Greeting " << person.getName() << endl;
    makeSmallTalkWith(person);

    Person *passerBy = new Person("Sir");
    makeSmallTalkWith(*passerBy);

    delete passerBy;
    passerBy = 0;
}

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

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

    delete eve;
    eve = 0;
}
```

main.cpp

Created Eve

Greeting Eve

Isn't the weather quite pleasant today,
Eve?

Created Sir

Isn't the weather quite pleasant today,
Sir?

Good bye Sir

Greeting Eve

Isn't the weather quite pleasant today,
Eve?

Created Sir

Isn't the weather quite pleasant today,
Sir?

Good bye Sir

Good bye Eve

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;
}
```

Mit std::shared_ptr



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

using namespace std;
void makeSmallTalkWith(ConstPersonPtr person){
    cout << "Isn't the weather quite pleasant today, "
        << person->getName() << "?" << endl;
}

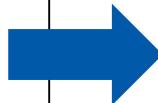
void greet(ConstPersonPtr person){
    cout << "Greeting " << person->getName() << endl;
    makeSmallTalkWith(person);

    ConstPersonPtr passerBy(new Person("Sir"));
    makeSmallTalkWith(passerBy);
}

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

    ConstPersonPtr alice = eve;
    greet(alice);
}
```

main.cpp



Created Eve

Greeting Eve

Isn't the weather quite pleasant today,
Eve?

Created Sir

Isn't the weather quite pleasant today,
Sir?

Good bye Sir

Greeting Eve

Isn't the weather quite pleasant today,
Eve?

Created Sir

Isn't the weather quite pleasant today,
Sir?

Good bye Sir

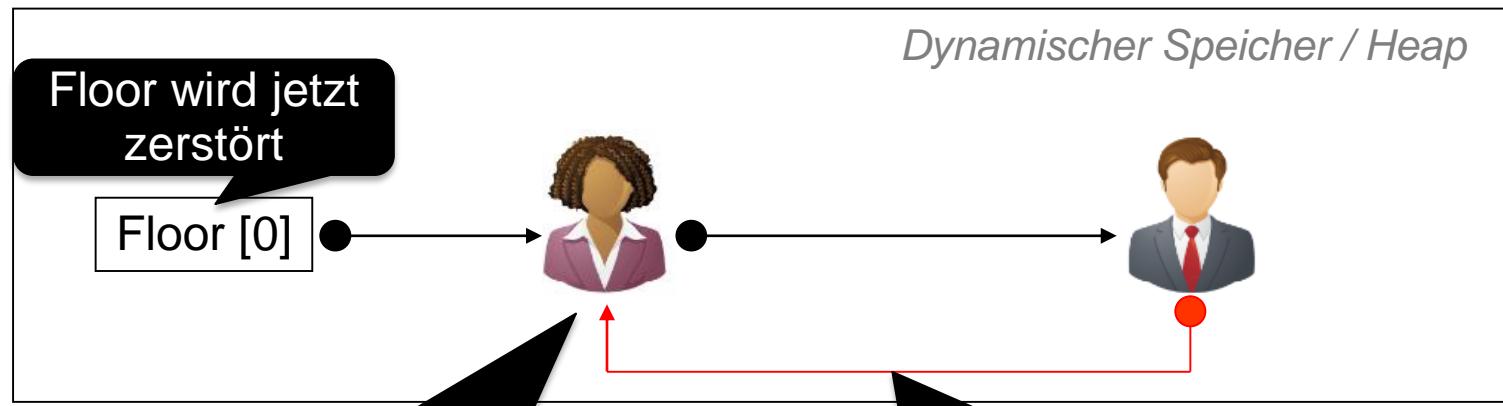
Good bye Eve

Weak SmartPointer: Motivation



`std::shared_ptr` ist nicht perfekt:

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



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

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

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

Intermezzo



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Wir haben das Problem mit einem schwachen
Zeiger für eine Richtung der Beziehung
zwischen Personen gelöst...

Wie hätte man das sonst lösen können?

Was wäre die Konsequenz?



<http://cliparts.co/clipart/2613703>

Mögliche Lösung für zyklische Zeiger



Wir verzichten einfach ganz auf Zeiger.

```
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!

Mögliche Lösung für zyklische 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;  
}  
}
```

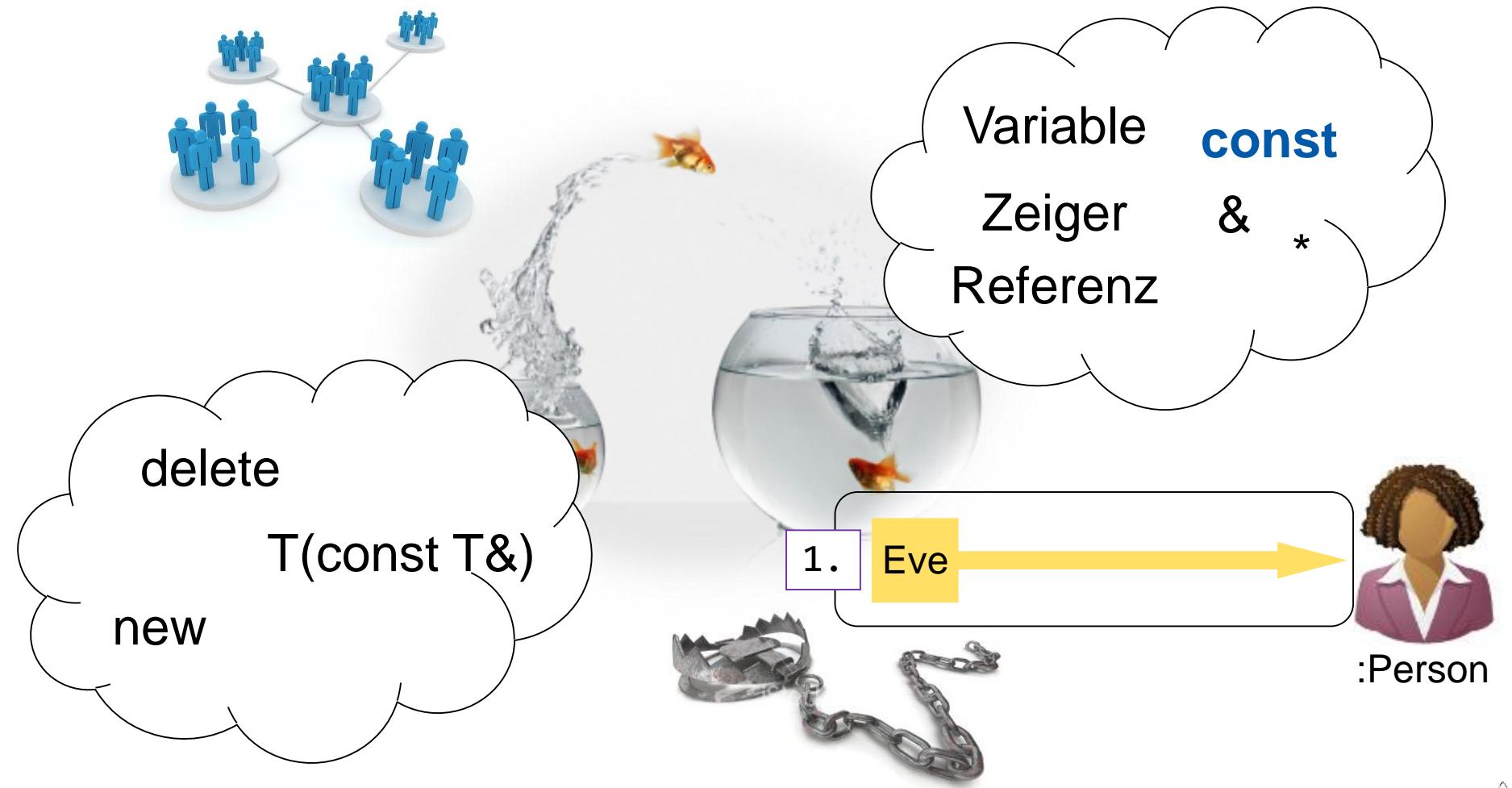
Ausgabe:

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

Kann man mit **immutable** Objekten (wie *java.lang.String*) umgehen.

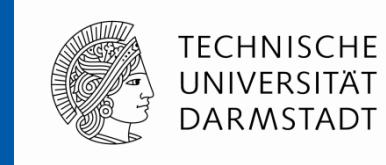
[EN]  /Immutable_object

Zusammenfassung



Programmierpraktikum C und C++

Vererbung und Polymorphie



ES Real-Time Systems Lab

Prof. Dr. rer. nat. Andy Schürr

Dept. of Electrical Engineering and Information Technology

Dept. of Computer Science (adjunct Professor)

www.es.tu-darmstadt.de

Roland Kluge

roland.kluge@es.tu-darmstadt.de

Was ist Polymorphie?



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- (Wir meinen hier **Untertyp-Polymorphie**)
- Eine Variable kann Instanzen verschiedener Klassen enthalten.

[EN]  /Polymorphism (computer science)

Ein einfaches Beispiel für Polymorphie

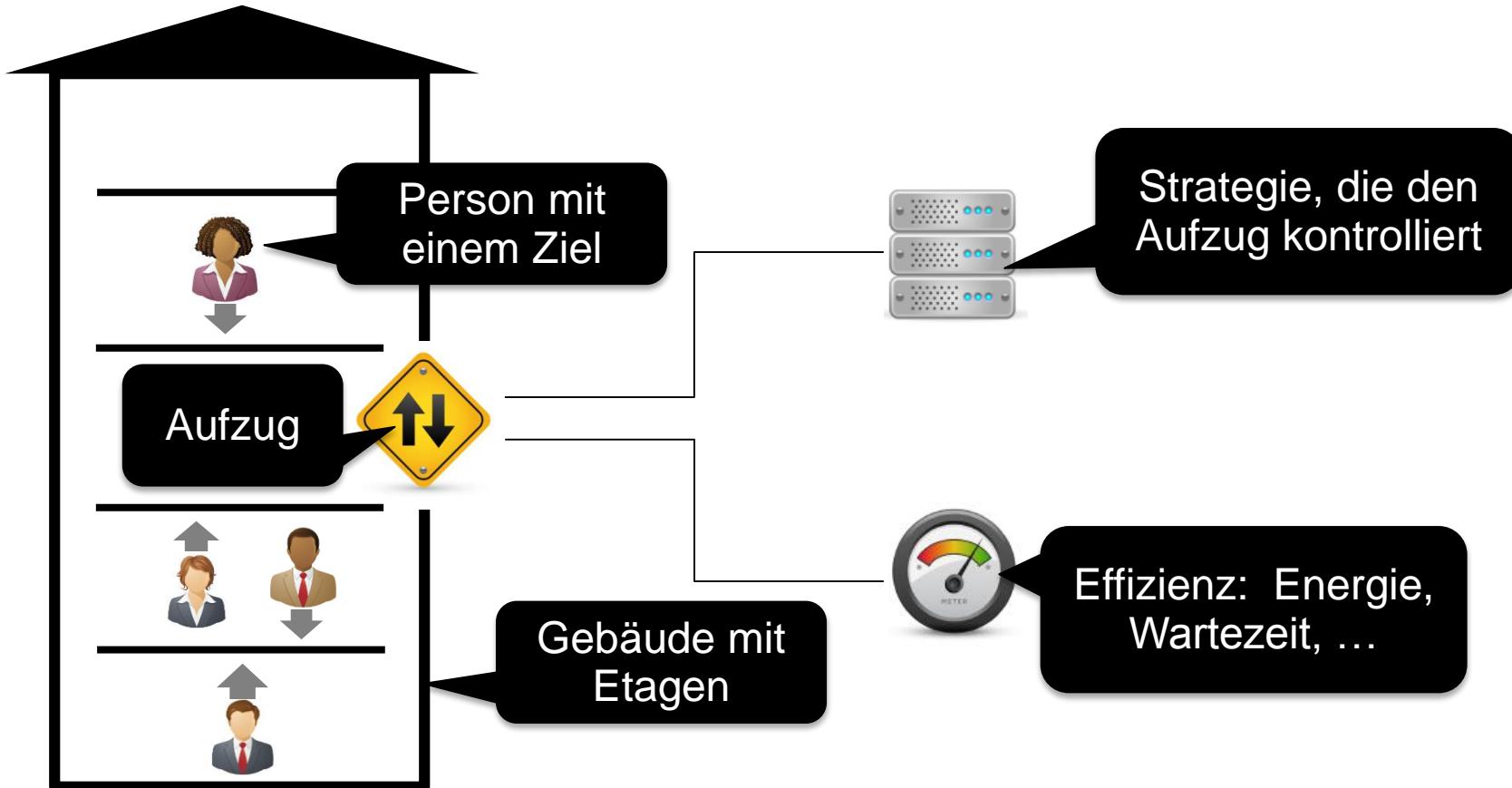


```
class Base {  
public:  
    virtual void print() {  
        std::cout << "B" << std::endl;  
    };  
  
class Child : public Base{  
public:  
    virtual void print() override {  
        std::cout << "C" << std::endl;  
    };  
  
    void doPrint(Base b) { b.print(); }  
  
    void doPrintRef(Base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



Lösung ohne und mit Polymorphie



```
void Elevator::moveToNextFloor(int strategy){  
    switch(strategy){  
        case ENERGY_MINIMIZING_STRATEGY:  
            // ...  
            break;  
        case WAITING_TIME_MINIMIZING_STRATEGY:  
            // ...  
            break;  
  
        // and so on ...  
    }  
}
```

```
void Elevator::moveToNextFloor(){  
    currentFloor =  
        strategy->next(this);  
}
```

„Dispatch“ von Hand

Für jede neue Strategie muss die Logik hier (und eventuell an **etlichen anderen Stellen**) erweitert werden!
(Fluch des switch-case)

Polymorpher Dispatch

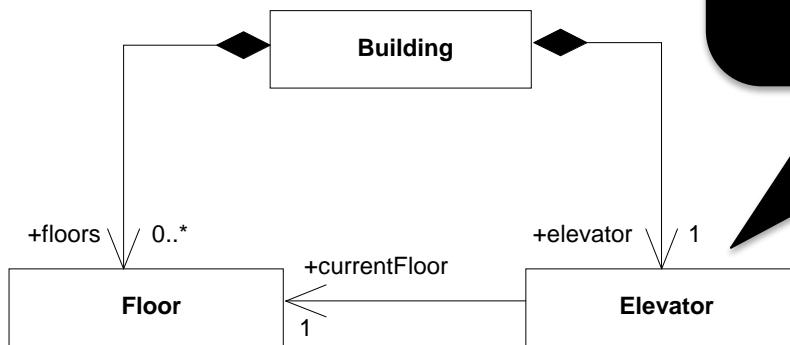
Konkrete **ElevatorStrategy** wird bei der Erzeugung des Aufzugs gesetzt.

Der obige Code ruft die Strategie polymorph auf und muss nicht mehr verändert werden.

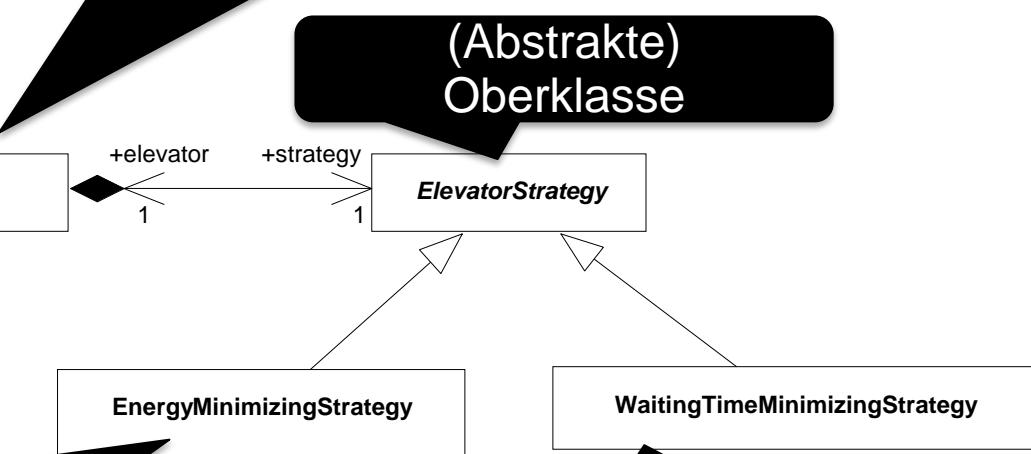
Verschiedene Strategien als Unterklassen



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



(Abstrakte)
Oberklasse



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

Konkrete Unterklassen

Intermezzo



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Was ist der Vorteil von Polymorphie?

Wie kann das so wichtig sein wenn z.B. C das nicht unterstützt (und C doch so weitverbreitet ist)?!

Was hat Polymorphie mit Vererbung zu tun? Geht es auch ohne Vererbung?



<http://cliparts.co/clipart/2613703>

Erläuterungen

Vorausdeklaration (statt `#include`), um zyklische Abhängigkeit zu vermeiden

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

class Elevator;

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

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

typedef std::shared_ptr<ElevatorStrategy>
ElevatorStrategyPtr;

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

In der Impl-Datei ist dies aber kein Problem!

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

using namespace std;

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

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

const Floor*
ElevatorStrategy::next(const Elevator *elevator) const
{
    /* Do nothing */
    return elevator->getCurrentFloor();
}
```

Vorausdeklarationen können nur dann verwendet werden, wenn **nur Referenzen oder Pointer** auf die referenzierte Klasse (Elevator) 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

```
#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

Parameter ohne
Namen möglich

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



ElevatorStrategy.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

ElevatorStrategy.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;
}

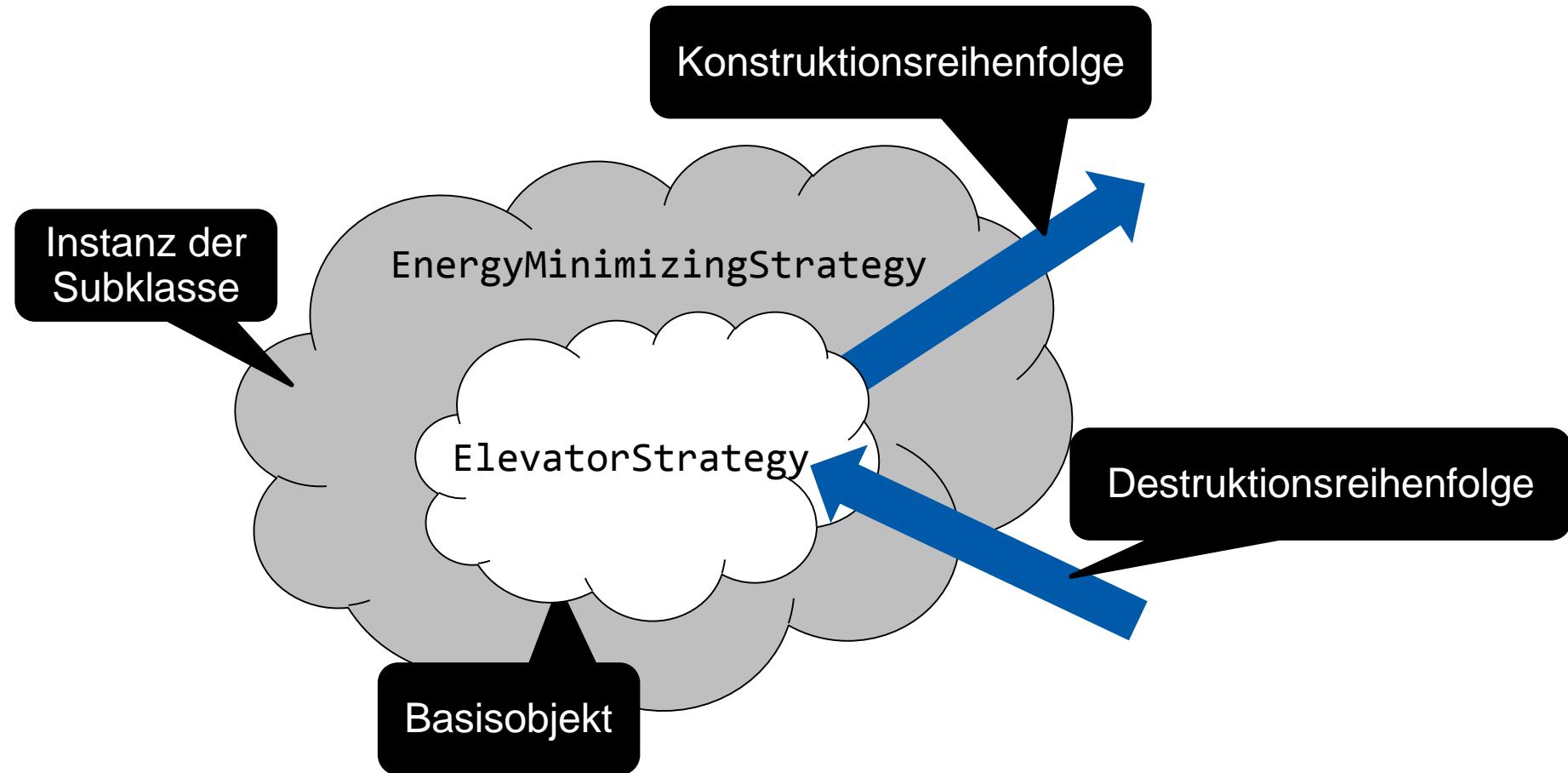
return elevator->getCurrentFloor();
}
```

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

Konstruktion und Dekonstruktion von Objekten bei Vererbung



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Intermezzo



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Wieso werden Konstruktoren von innen nach außen
und Destruktoren von außen nach innen aufgerufen?



<http://cliparts.co/clipart/2613703>

Probelauf unserer Simulation



Eure Aufgabe in der
Übung

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

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

int main() {
    ElevatorStrategy *strg = new EnergyMinimizingStrategy();

    // Do something...

    ConstElevatorStrategyPtr strategy(strg);
    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.
```

```
~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
```

Konstruktoren werden
richtig aufgerufen

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;  
};
```

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

Intermezzo

Warum muss der Destruktor in einer Klasse mit virtuellen Methoden auch virtuell sein?

Wie ist das mit virtuellen Konstruktoren?



<http://cliparts.co/clipart/2613703>

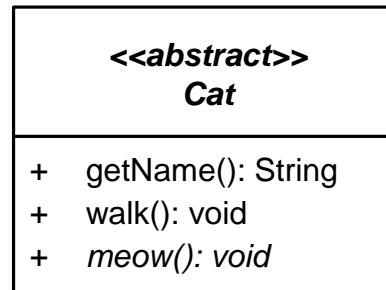
Exkurs: Virtual Method Table

Der Mechanismus der dynamischen Bindung



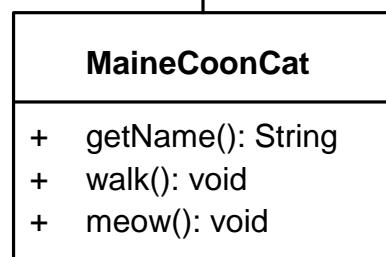
Egal wie der Pointer auf ein Objekt deklariert ist (z.B. *ElevatorStrategy**), das Objekt behält seinen Typ (z.B. *EnergyMinimizingStrategy*).

Jede Klasse besitzt eine „**Lookup**“-Tabelle (**vtable**), die jeder virtuellen Methode ihre Implementierung zuweist.



Methode	Implementierung
getName	Cat::getName
walk	Cat::walk
meow	NULL

Enthält standardmäßig:
Java,... : alle Methoden
C++,... : keine Methode



Methode	Implementierung
getName	Cat::getName
walk	MaineCoonCat::walk
meow	MaineCoonCat::meow

Falls kein Eintrag:
Verwende Methode des
Typs des Pointers.

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

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



Polymorpher Aufruf
funktioniert jetzt



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 nicht mehr instantiiert werden.

Methode ist hiermit **rein virtuell** – keine Default-Implementierung.

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

Intermezzo

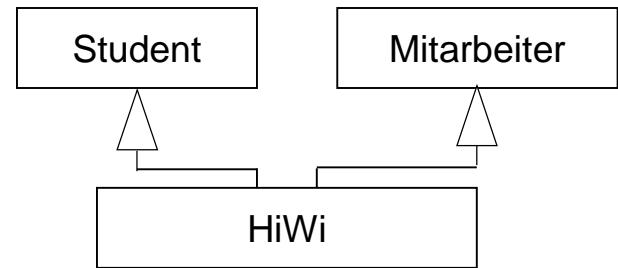
Wieso sind virtuelle Methoden „teuer“?

Was bedeutet jede **const-Verwendung** im folgenden Ausdruck:

```
virtual const Floor* ElevatoryStrategy::next(const  
Elevator *elevator) const = 0;
```

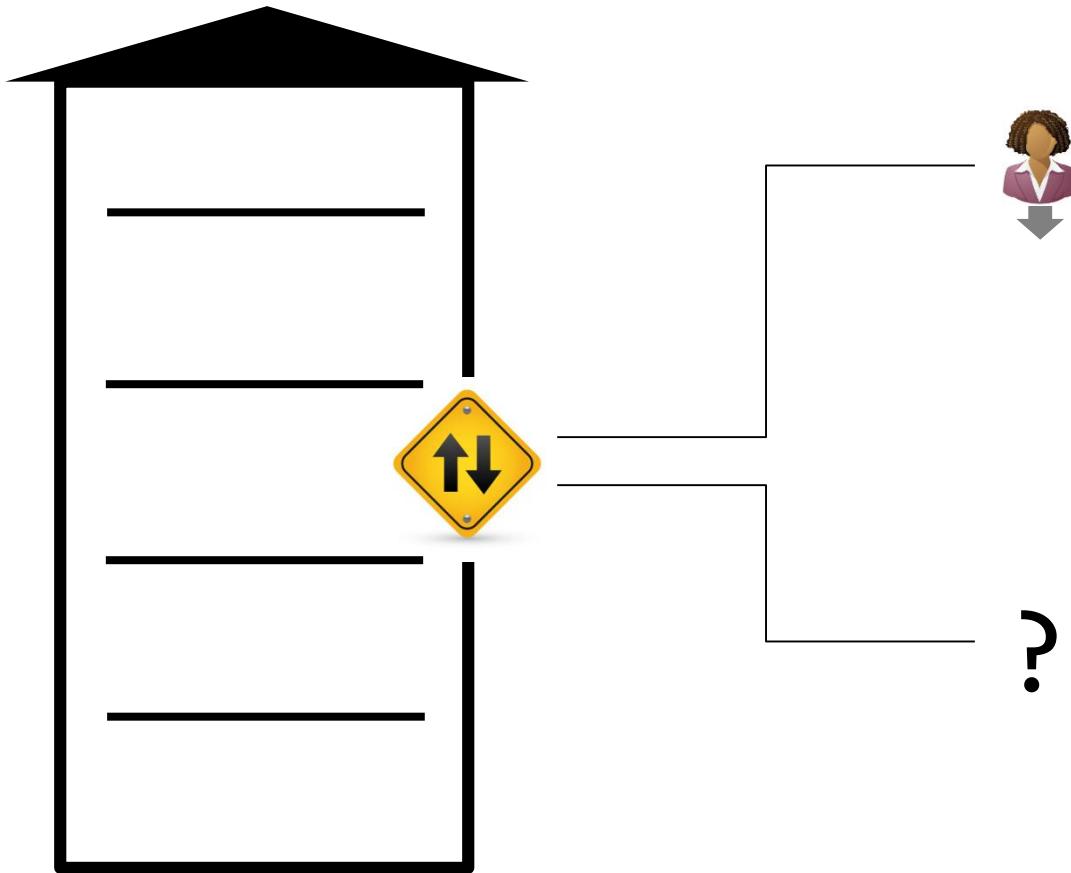


<http://cliparts.co/clipart/2613703>



MEHRFACHVERERBUNG

Mehrfachvererbung: Motivation



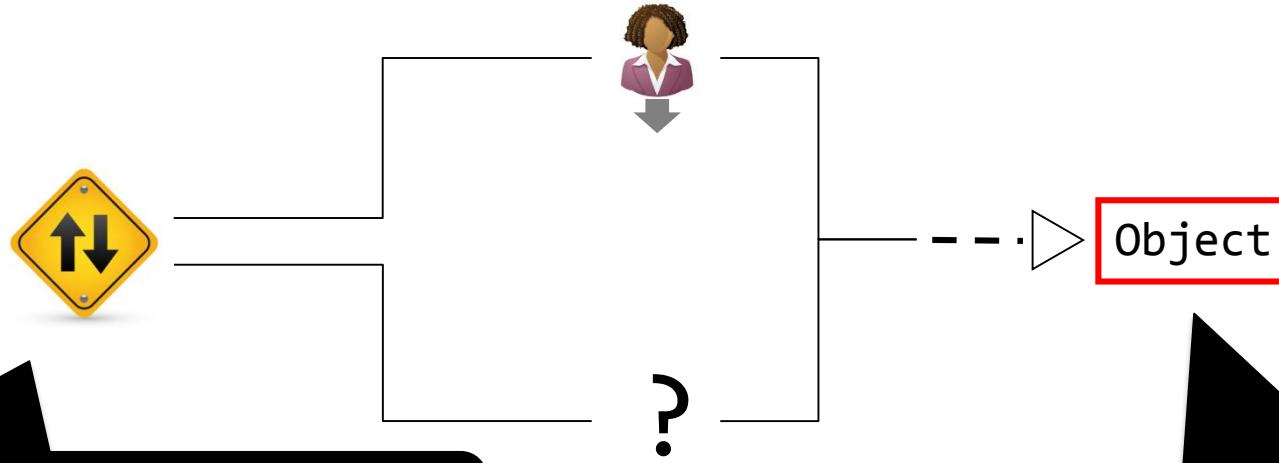
Ziel: Aufzüge für bestimmte Zwecke

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

Historie: Das Containerproblem



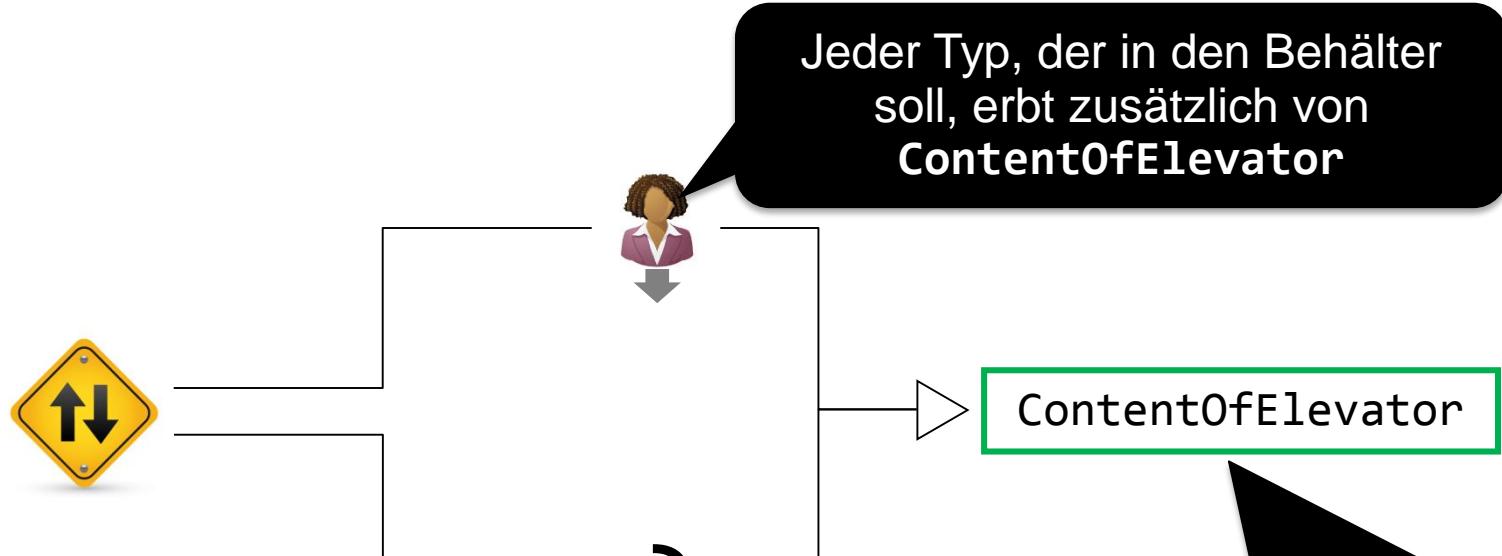
Ursprünglich als Lösung für Containerproblem: Wir wollen Objekte unterschiedlicher Art in den Aufzug (Container) laden.



Wir können also nicht einfach
Objects in den Aufzug laden

Aus Effizienzgründen gibt es in
C++ keine generische
Oberklasse wie
java.lang.Object

Lösung mit Mehrfachvererbung



- (⌚) technisch bedingt,
keine Designentscheidung!
- (⌚) komplexe Vererbungs-
hierarchien



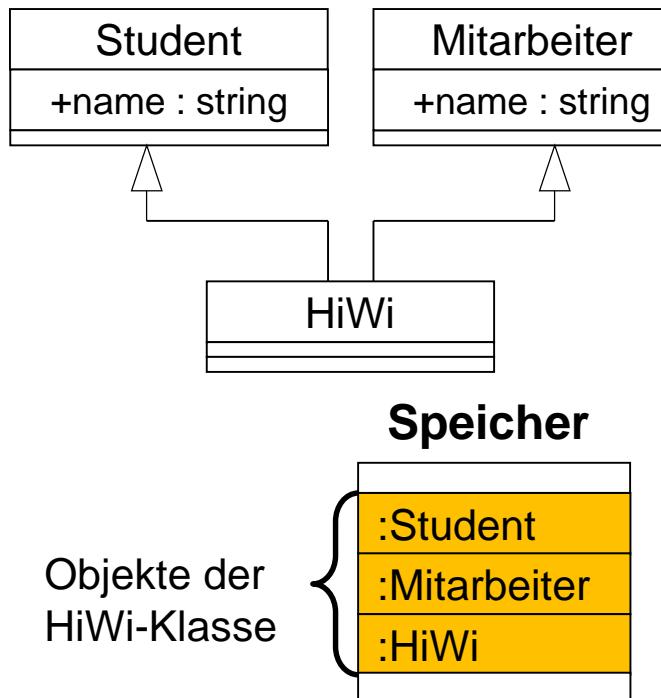
Besserer Ersatz (in
diesem Fall):
Templates (Tag 4)

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 Mitarbeiter {public: std::string name;};

class HiWi: public Student,
             public Mitarbeiter {};

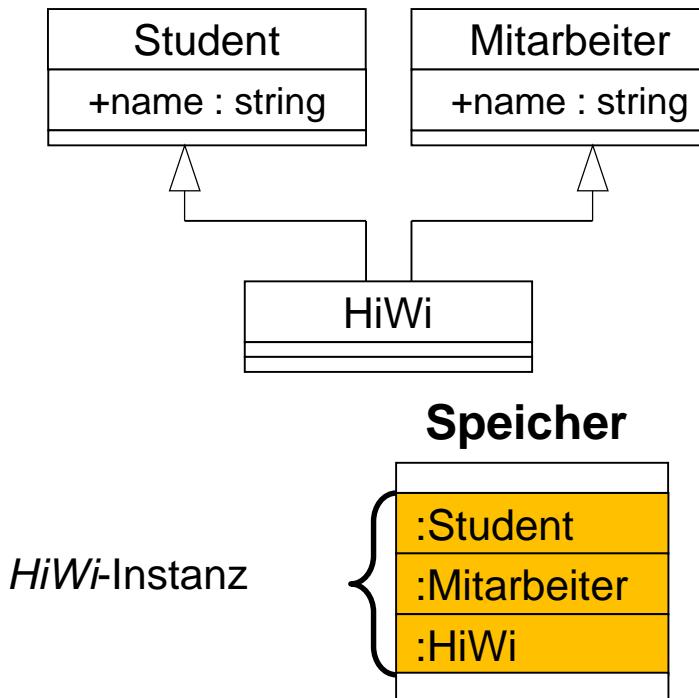
int main() {
    HiWi* h = new HiWi();
    h->name = "Christian";
    /* Error: request for name is ambiguous */
}
```

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 Mitarbeiter {public: std::string name; };

class HiWi: public Student, public Mitarbeiter {};

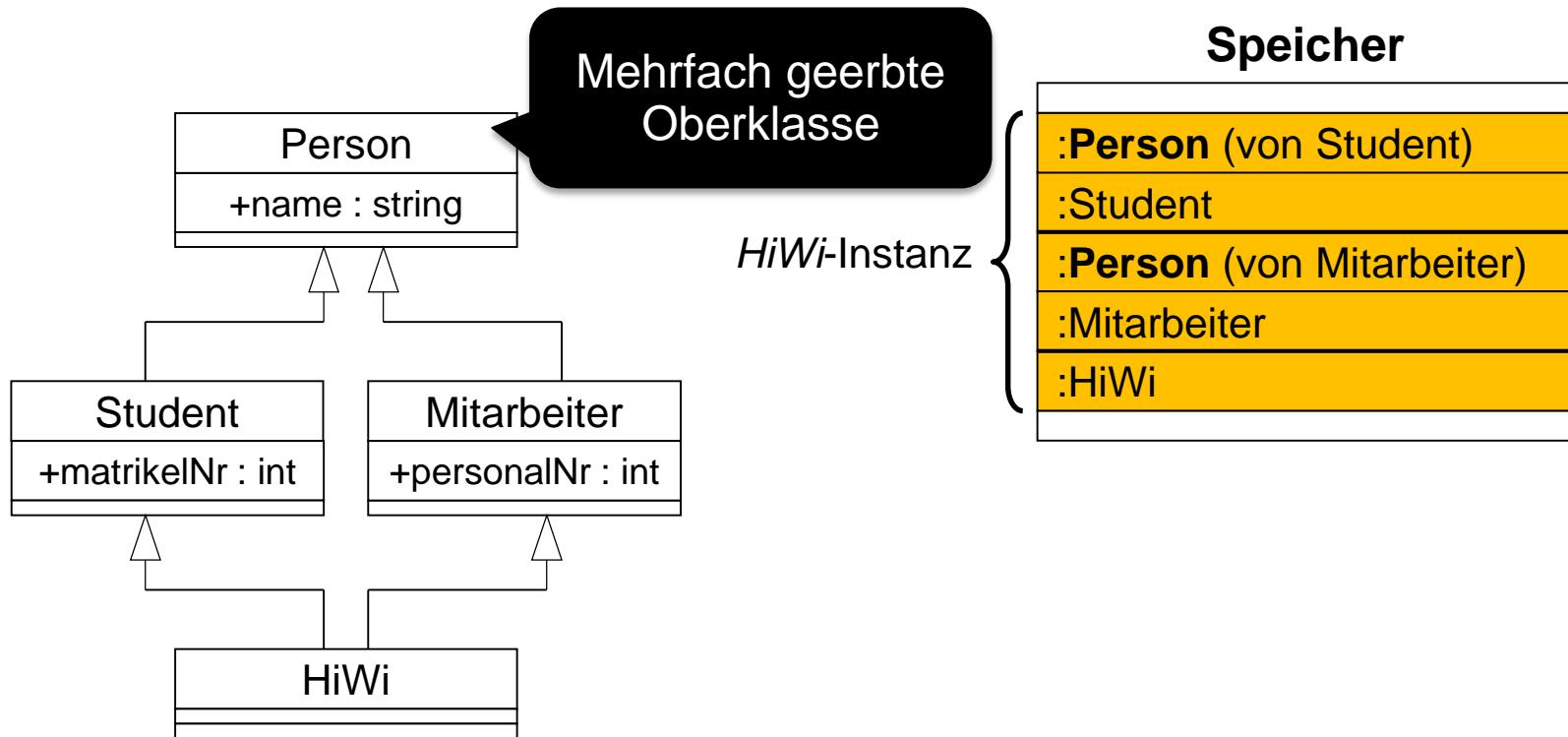
int main() {
    HiWi* h = new HiWi();
    h->Student::name = "Christian";
    h->Mitarbeiter::name = "Mark";
}
```

Scope-Operator nötig!

Implementierungsvererbung: Speicherproblematik



Mehrfach geerbte Oberklassen führen auch zur unnötigen Bindung von Speicher

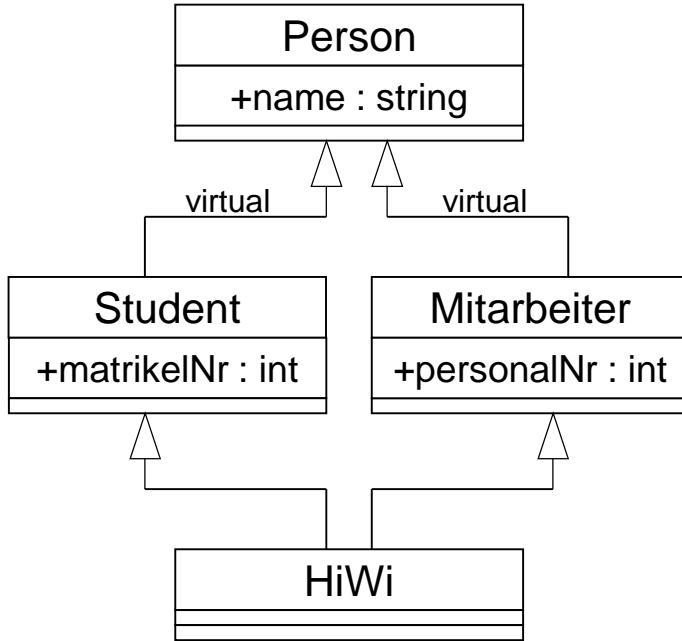


Implementierungsvererb.: Speicherproblematik



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 Mitarbeiter : virtual public Person

class HiWi:
    public Student,
    public Mitarbeiter {};

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

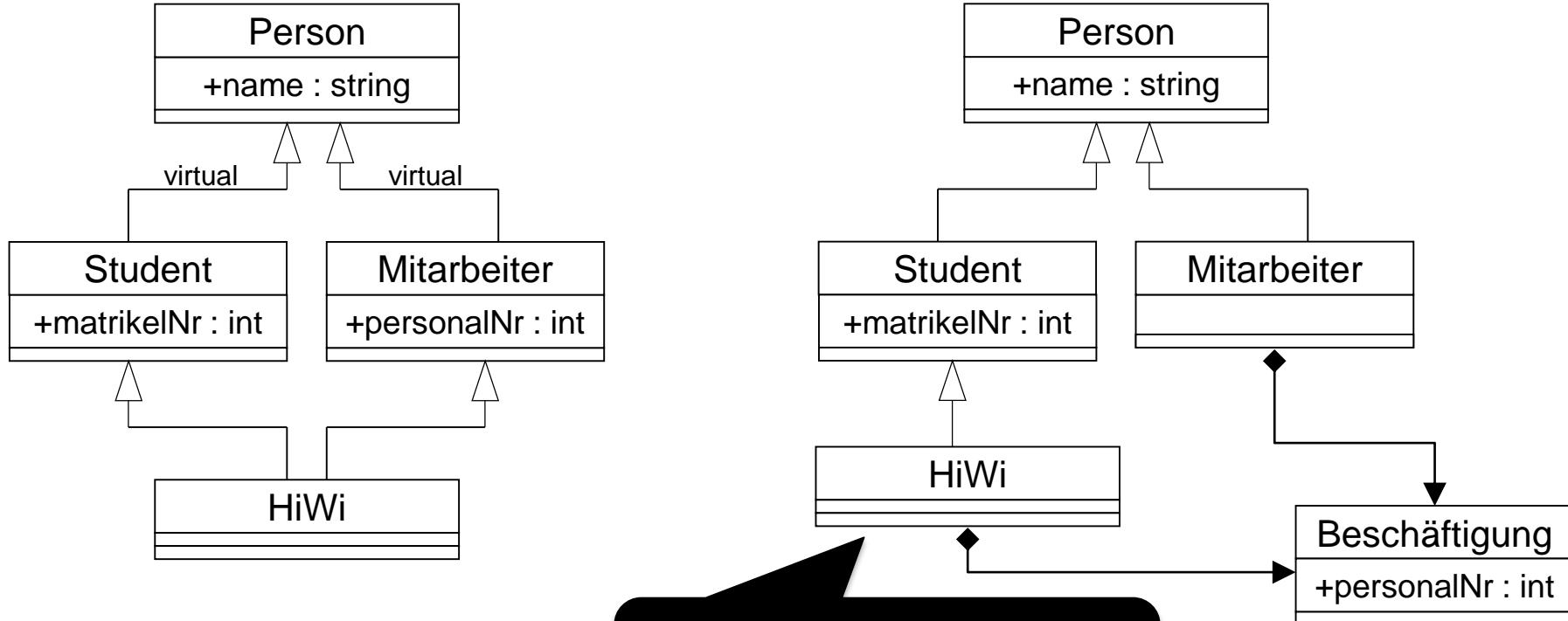
! Die **virtual**-Deklaration findet nicht an der Stelle statt, die sie nötig macht (**HiWi**)!

Implementierungsvererbung: Schlechtes Design?



Mehrfachvererbung kann auf „schlechtes“ Design hindeuten

Gemeinsamkeiten sollen explizit extrahiert bzw. das Design vereinfacht werden



Ein HiWi ist ein Student,
mit einer Beschäftigung

Schnittstellen- vs. Implementierungsvererbung



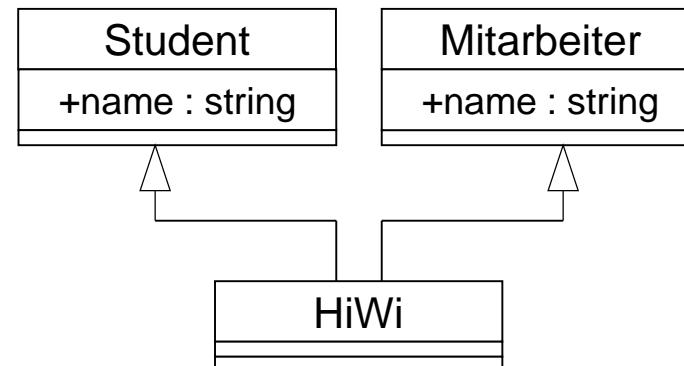
Schnittstellenvererbung:

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...



Wie war das eigentlich mit der Mehrfachvererbung in Java?



Frage: Wie wird in Java die folgende Situation gelöst?

Antwort: Gar nicht – darf so nicht vorkommen!

```
interface InterfaceA {      int run();      }
interface InterfaceB {      boolean run();     }

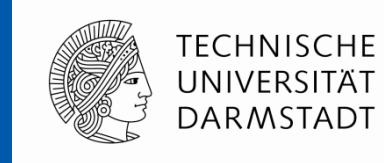
public class MyClass implements InterfaceA, InterfaceB {

    @Override
    public int run() {
        return 0;
    }
}
```

Error: The return type
is incompatible with
InterfaceB.run()

Programmierpraktikum C und C++

Fortgeschrittene Themen



ES Real-Time Systems Lab

Prof. Dr. rer. nat. Andy Schürr

Dept. of Electrical Engineering and Information Technology

Dept. of Computer Science (adjunct Professor)

www.es.tu-darmstadt.de

Roland Kluge

roland.kluge@es.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>
```

4. Buildprozess mit Makefiles

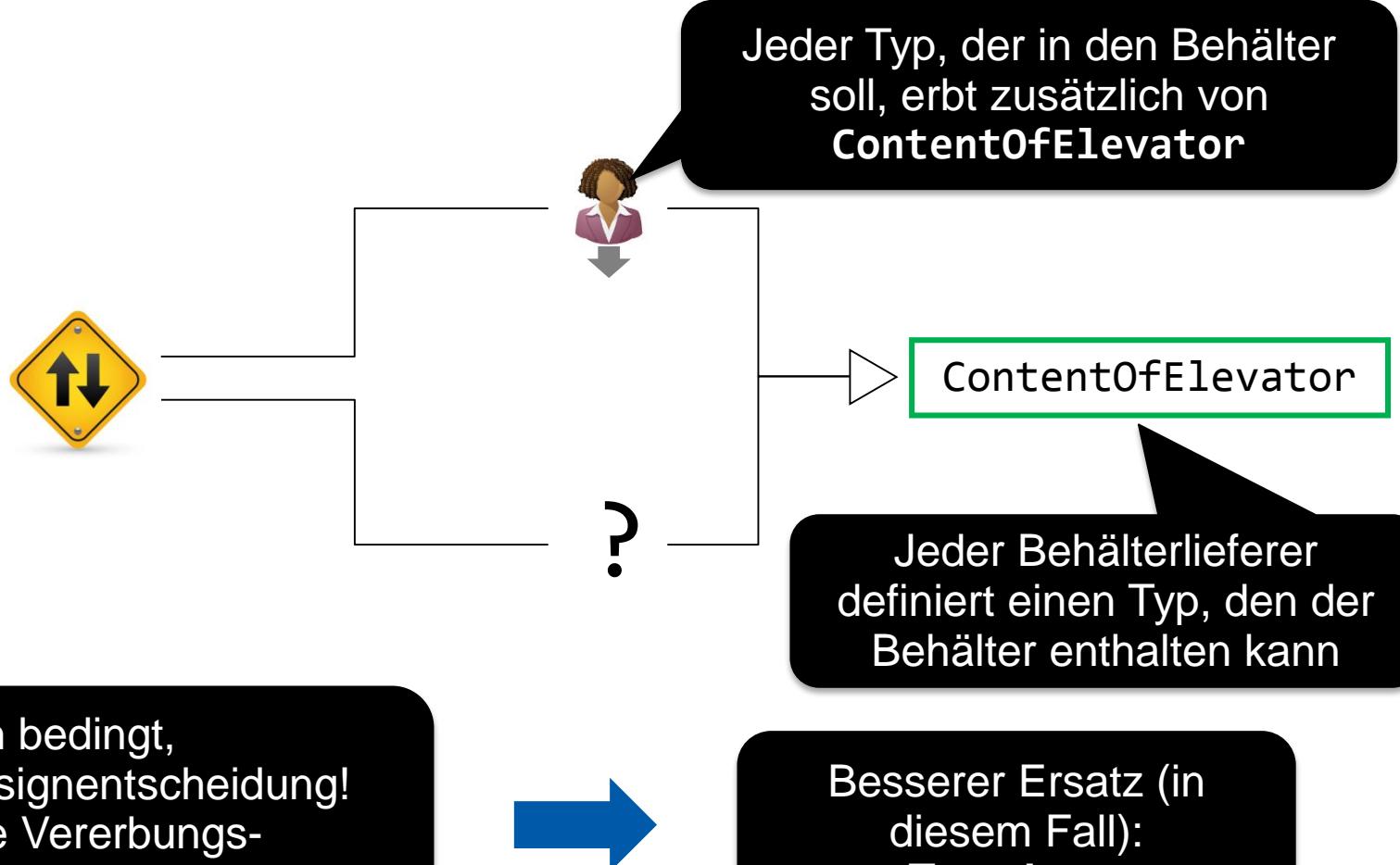
```
all: main.exe

main.exe: main.o Cat.o Dog.o
g++ -o main.exe main.o Cat.o Dog.o
```



TEMPLATES

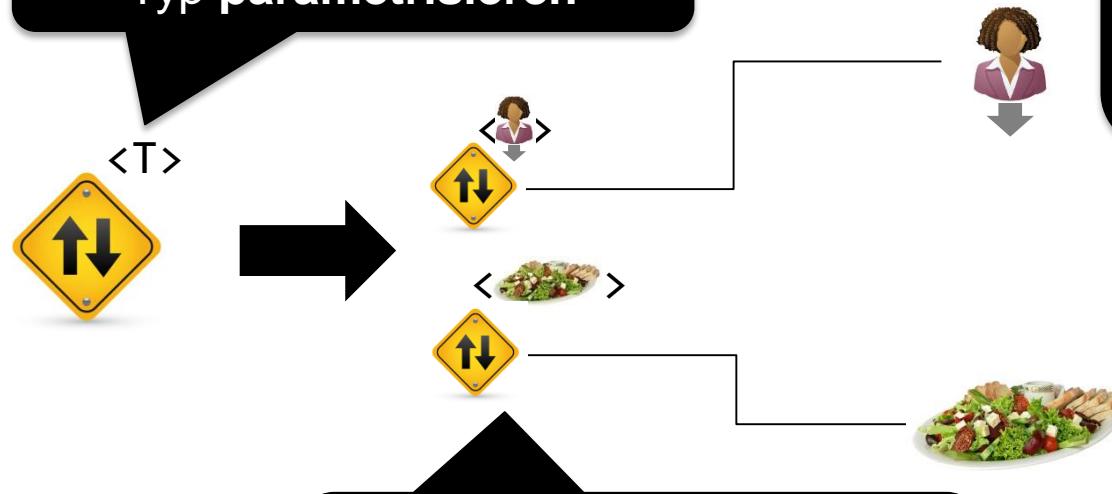
Rückschau: Containerproblem und Mehrfachvererbung



Templates: Idee



Implementierung mit einem
Typ parametrisieren



Bei Bedarf wird die richtige
Version der Implementierung
zur Kompilierzeit generiert

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

C++-Templates induzieren ein
implizites „Interface“ durch
die Art der Verwendung des
generischen Typparameters

Intermezzo



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Wieso ist „Object“ teuer?

Was ist der Unterschied zwischen Templates in C++
und Generics in Java?

Wie wird dieses „Problem“ in anderen Sprache
gelöst?

- C
- Scheme
- Haskell
- Python
- Ruby
- ...



<http://cliparts.co/clipart/2613703>

Generics 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("1st address: 0x%p\n",  
          list.firstElement);  
    printf("1st content: '%s'\n",  
          (const char*)list.firstElement->content);  
}
```

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

Expliziter cast nötig

Class Templates: Syntax am Beispiel



```
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

Class Templates: Syntax am Beispiel



```
template<class T = Person>
class Elevator {
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:
    vector<const T*> transportedObjects;
};
```

T wird deklariert als **Typparameter**.
(Mit optionalem **Defaulttyp Person**)

Der Typparameter wird als **Platzhalter** für den konkreten Typ eingesetzt.

Erst bei der Expansion des Templates wird sich herausstellen, ob der Typparameter wirklich diese Methoden hat (~ **Duck Typing**).

Bei Templates ist **keine Trennung** in Header und Impl-Datei möglich.

WARUM?

Function Templates: Syntax am Beispiel



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

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

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

    return total;
}
```

Mehrere Typparameter möglich
(auch bei Class 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“
können auch
verwendet
werden

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()

Intermezzo



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Was ist genau damit gemeint, dass Templates eine Schnittstelle induzieren?

Was sind Vorteile und Nachteile dieser Art von „impliziten“ Schnittstellen?

Was ist genau der Unterschied zwischen C++-Templates und Java-Generics?



<http://cliparts.co/clipart/2613703>

Induzierte Schnittstelle



Template-Code

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

Mixins: Mehrfachvererbung trifft Templates



```
template<
    class Logger,
    class Security,
    class OperatingSystem,
    class Platform
>
class System :
    public Logger,
    public Security,
    public OperatingSystem,
    public Platform
{  
};
```

Mixins werden als Typparameter definiert...

...und „reingemischt“ mit Mehrfachvererbung!

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

Vergleich mit Mehrfachvererbung



1. **Schnittstellenvererbung** sinnvoll, nützlich (Design!) und zumeist unproblematisch
2. **Implementierungsvererbung** problematisch und zu vermeiden (Komposition vorziehen)
3. **Mixins** durchaus sinnvoll - eigentlich eine Art Komposition



FUNKTIONSZEIGER, FUNKTIONSOBJEKTE UND METHODENZEIGER

Funktionszeiger: Motivation (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: Motivation (II)

```
template<class F, class T>
void applyToSequence(F function, T* begin, T* end){
    while (begin != end) function(*begin++);
}

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

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

Auch Funktionen können Typparameter tragen.

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

Funktionszeiger: Beispiel II



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

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

Funktionszeiger: Syntax



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

Typ des
Rückgabewerts

Zeigertyp, Klammern
sind notwendig um
Rückgabetyp und Zeiger
auseinanderzuhalten

Name der
Variable

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

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

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

Funktionsobjekte und Templates



```
class ConsoleLogger {  
public:  
    ConsoleLogger();  
    ~ConsoleLogger();  
  
    inline void operator()(int i) const {  
        std::cout << "user:~ /$ " << i << std::endl;  
    }  
};  
  
template<class F, class 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);  
}
```

operator() erlaubt, Objekte mit Funktionsyntax anzusprechen

Syntax bleibt hier identisch, obwohl wir eine Methode aufrufen

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

Methodenzeiger: Beispiel



```
class ConsoleLogger {  
public:  
    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

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

Funktionszeiger vs. Methodenzeiger



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

```
template<class C, class F, class 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.

Intermezzo

Wieso sind Zeiger auf Funktionen nützlich?

Gibt es auch Nachteile?

(*) Sind Zeiger auf Funktionen in C++
genauso flexibel wie richtige „Zeiger auf
Funktionen“ in (funktionalen)
Programmiersprachen wie
Scheme/Lisp/Haskell/Ruby/Python?



<http://cliparts.co/clipart/2613703>

Zeiger auf Funktionen: 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

Sobald die implementierte Funktionalität komplexer wird (-> Zustand), sind **Funktionsobjekte** oder **Methodenzeige** (je nach Kontext) sinnvoll.



TECHNISCHE
UNIVERSITÄT
DARMSTADT

STANDARD-BIBLIOTHEKEN IN C++

Standard-Bibliotheken in C++



- ISO-Standard legt Funktionsumfang der Standardbibliothek fest
 - Alle Komponenten sind im **namespace std**
 - Komponenten:
 - I/O (z.B. <iostream>)
 - Strings (z.B. <string>)
 - Standard Template Library (STL)
 - Generische Datenstrukturen
(z.B. <vector>, <array>, <list>, <priority_queue>)
 - Generische Algorithmen
(z.B. <algorithm>, <iterator>, <functional>)
- Flexible, erweiterbare IO
- auch: Regex, ...
- siehe später
- siehe später: **copy** und **remove_copy_if**

Boost:

„Brutschränk“ für C++-Standardkomponenten



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

Herb Sutter, Andrei Alexandrescu, C++ Coding Standards

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

Array

Filesystem

Lambda

Odeint

Chrono

Function(al)

Math
(advanced)

Smart Ptr

Date Time

Graph

MPI

System

Generische STL-Algorithmen: `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/>

Intermezzo



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

InputIterator: muss Operatoren `++`, `*`, `==`, und `!=` unterstützen
OutputIterator: muss Operatoren `++` und `*` unterstützen

Wieso ist diese Forderung notwendig?



<http://cliparts.co/clipart/2613703>

Generische STL-Algorithmen: `copy`



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

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

using namespace std;

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

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

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

STL-Behälter bieten
InputIteratoren an

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

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

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

Generische STL-Algorithmen: *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: *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: *priority_queue*



```
template <class T,
```

Typ vom Inhalt der
Warteschlange

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

Typ des darunterliegenden
Behälters (*vector* wird als Default
verwendet)

```
class Compare = less<
```

Binäres Prädikat (*less* wird
als Default verwendet)

```
typename Container::value_type>
```

Damit Compiler weiß, dass
value_type ein Typ ist

```
class priority_queue;
```

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

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

Generische Behälter: *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};

    priority_queue<int>
        descending(numbers, numbers + 5);
    process_queue(descending);
                                // 5,4,3,2,1

    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/

Intermezzo – Schleife vs. remove_copy_if



```
remove_copy_if( result.begin(),           // first
                result.end(),           // last
                ostream_iterator<int>(cout, ", "), // result
                even                   // predicate
);
```

vs.

```
for (T iter = first; iter != last; ++iter)
{
    if (!P(*iter)) {
        (*result) = *iter
        ++result;
    }
}
```

Was ist „schöner“ oder zumindest praktischer?



<http://cliparts.co/clipart/2613703>

Standard Template Library: Fazit



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Mächtig, effizient, ausgereift und gut dokumentiert

Anspruchsvoll zu erlernen (???) (erfordert Wissen über Templates, Funktoren, Iteratoren, Mixins, ...)

Boost als „Brutkasten“ für die nächsten Standards

Vielleicht sogar als **der Vorteil** von C++ zu betrachten!



TECHNISCHE
UNIVERSITÄT
DARMSTADT

MAKEFILES

Makefiles: Motivation



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

?

Wie soll man da den Überblick bewahren?

!

Mittels Regeln!

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

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)

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.

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)
```

- Wenn sich ein Header ändert, müssen alle abhängigen Dateien (`#include`) neu gebaut werden.
- Wo sind eigentlich die **Header**?
- 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
```

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

ABSCHLUSS DES C++-TEILS



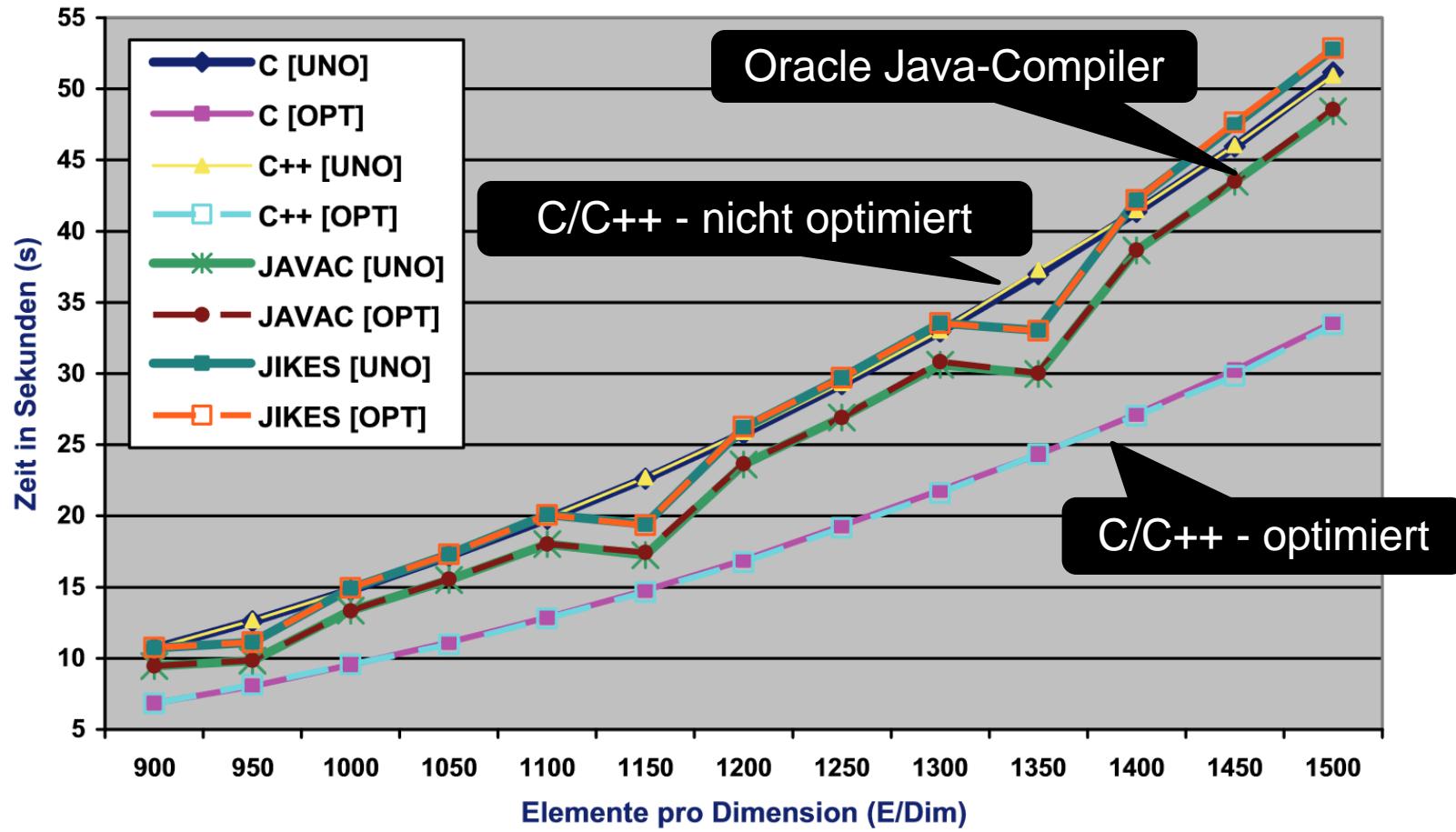
Java vs. C++: Stärken und Schwächen?

z.B. Stimmt es wirklich, dass Java „plattformunabhängig“ ist und C++ nicht?

<http://cliparts.co/clipart/2613703>

Laufzeitunterschied zwischen Java und C++

Beispiel Matrixmultiplikation



Manuel Prager: Laufzeitvergleiche für die Implementierung von Algorithmen in Java und C/C++
Hochschule Neubrandenburg, 2010

Look down!

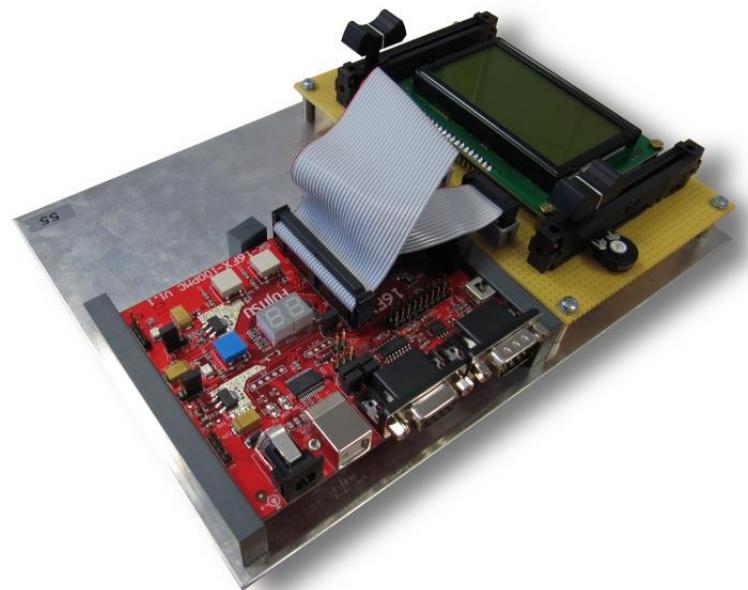


TECHNISCHE
UNIVERSITÄT
DARMSTADT

Nützliche Kommentare
finden sich
auch in den PowerPoint-Notizen!

Programmierpraktikum C und C++

C für Microcontroller – Einführung



ES Real-Time Systems Lab

Prof. Dr. rer. nat. Andy Schürr

Dept. of Electrical Engineering and Information Technology

Dept. of Computer Science (adjunct Professor)

www.es.tu-darmstadt.de

Roland Kluge

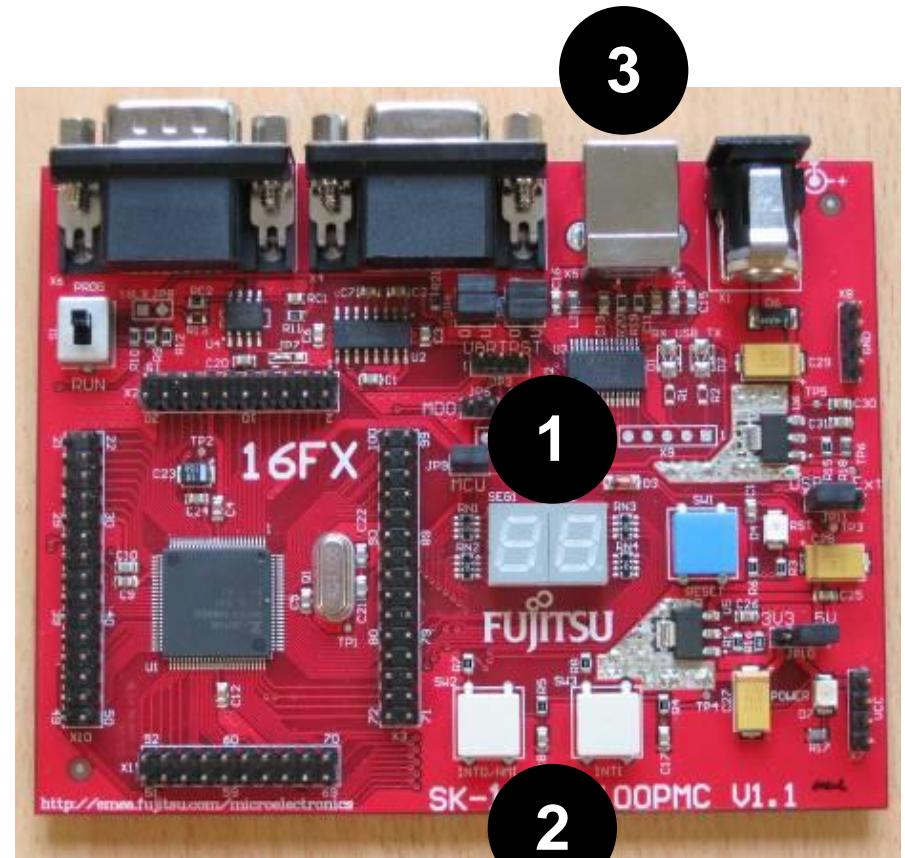
roland.kluge@es.tu-darmstadt.de

Entwicklungsboard



MB96F348HSB Mikrocontroller

- Prozessortaktung: bis 56 MHz
- RAM: 24 KiB
- Flash: 576 KiB
- 82 I/O Pins
- Analog/Digital-Wandler mit 24 Kanälen
- CAN-Controller



Starterkit SK-16FX-EUROscope

- Zwei 7-Segment-Anzeigen 1
- Zwei Druckschalter 2
- Stromversorgung über USB (5V) 3

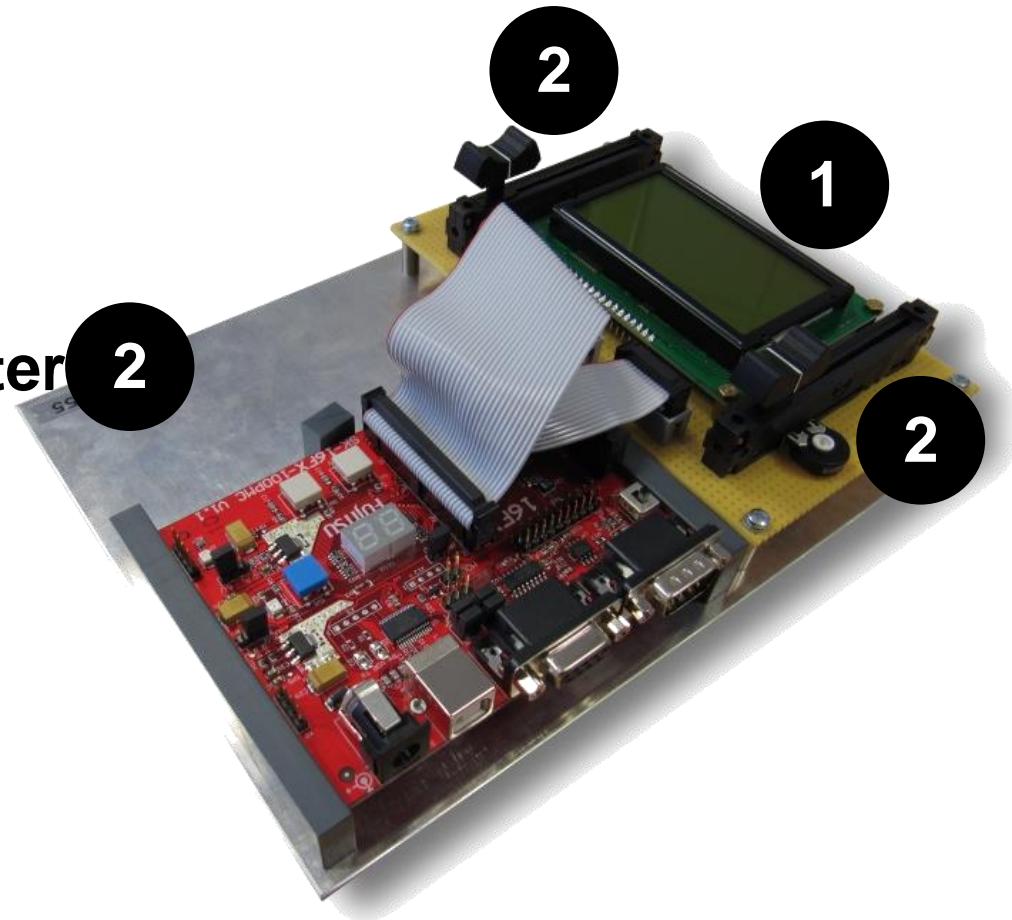
Erweiterungen gegenüber der Standardausführung



LC-Display 1

- AV128641 von Anag Vision
- Vollgraphisch
- 128 x 64 Pixel
- hintergrundbeleuchtet

Zwei Schiebepotentiometer 2





- Von **Fujitsu Microelectronics Ltd.**
 - Später: Spansion – Heute: Cypress
- Unterstützt nur **ANSI C90**,
 - zusätzlich auch **einzeilige Kommentare (//)**
 - Variablendeclaration **am Anfang einer Funktion** (sogar Schleifenzähler)
- **Busy Waiting:** Compiler enthält eine interne Funktion namens **`__wait_nop()`**, die eine CPU-Instruktion zum Warten für einen Taktzyklus („NOP“) auslöst
- **Konstanten** werden standardmäßig im **ROM** gespeichert, nicht im RAM (RAM ist wertvoll, da nur 24 KiB zur Verfügung stehen)

Mikrocontroller: Keine standardisierte „Umgebung“



- Compiler kann nicht wissen, welche Komponenten angeschlossen sind
- Es gibt **keine Ausgabe über printf()**
 - Alternative: 7-Segment-Anzeige, LCD(, LEDs)
 - Es ist sehr empfehlenswert, sich eine eigene kleine Debugging-Bibliothek zu schreiben
- Ansteuerung externer Komponenten muss vom Entwickler selber durchgeführt werden
 - wird zum Teil unterstützt durch fertige Bibliotheken



Umfangreiche und flexible Hardware → erfordert Konfiguration

- Realisiert über Register
 - Im Controller integrierte „Variablen“ mit unterschiedlicher Größe
 - Zugriff im Code über Präprozessor-Konstanten (z.B. PDR00, DDR01,...)
 - Bedeutung unterschiedlich je nach Register
 - Ganzes oder Teil des Registers als Zahlenwert, z.B. als Zähler
 - Einzelne Bits als „Schalter/Switch“ für bestimmte Funktion, z.B. einzelnes Ausgangspin auf High oder Low

Kommunikation mit Außenwelt über

- Einzelne digitale Ein/Ausgänge
- Analoge Eingänge
- Schnittstellen, z.B.
 - UART (serielle Schnittstelle)
 - CAN (serieller Bus)



8 Pins = Port

Je Pin mehrere Register, u.a.:

- **Port-Data-Register (PDR)**

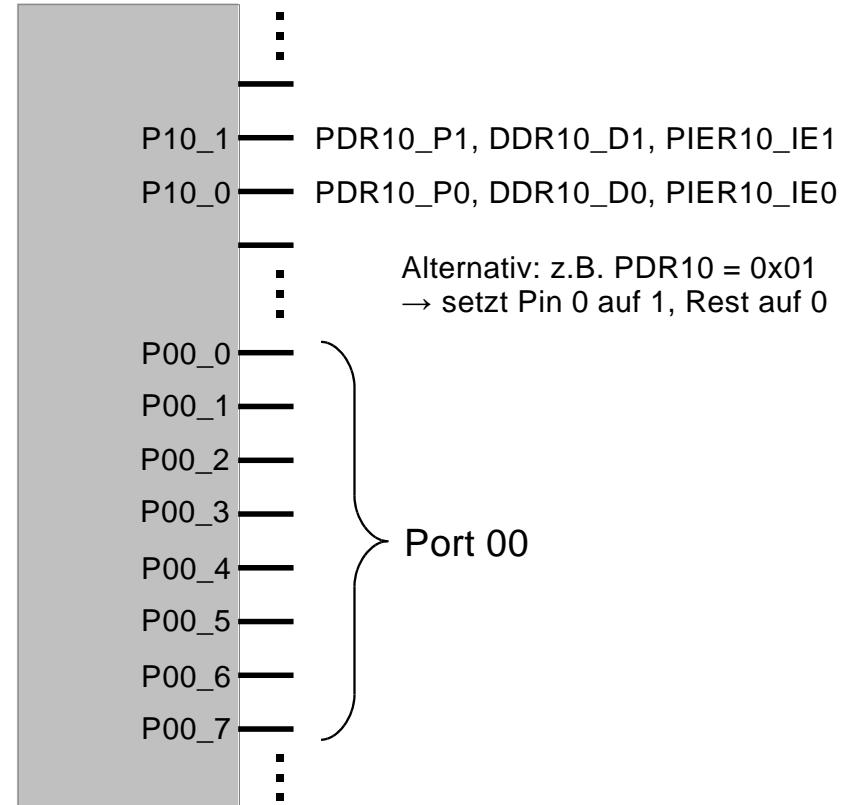
- als *Eingang*: Abfrage des Zustandes
- als *Ausgang*: Setzen des Pegels
- z.B. PDR07_P0

- **Data-Direction-Register (DDR)**

- Setzen auf Eingang oder Ausgang
- 0 → Eingang, 1 → Ausgang
- z.B. DDR07_D0

- **Port-Input-Enable-Register (PIER)**

- Bei Eingangspin den Eingang aktiv schalten
- z.B. PIER07_IE0



Beispielcode: Pins abfragen



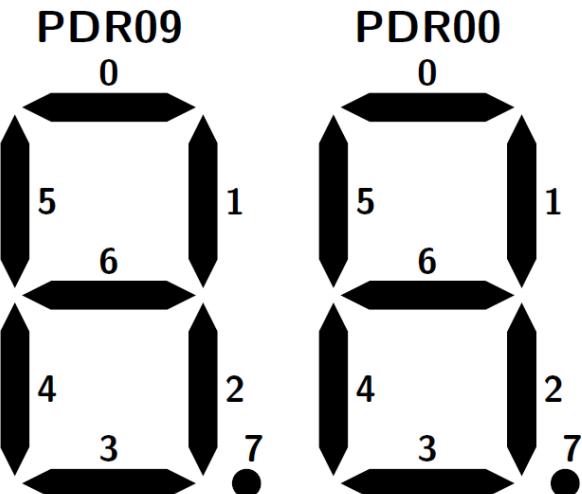
```
/* Beispiel: Pins als Eingang */
char status;
DDR07_D0 = 0;           // Pin 0 von Port 07 als Input
PIER07_IE0 = 1;         // Pin 0 von Port 07 als Eingang
aktiv

status = PDR07_P0;      // Pegel an Pin 0 von Port 07 abfragen
// -> Status des linken Tasters
```

Beispielcode: 7-Segment-Anzeige



```
/* Beispiel: 7-Segment-Anzeige */  
DDR00 = 0xff;      // Alle Pins von Port 00 als Output  
PDR00 = 0xff;      // Alle Pins von Port 00 auf High-Pegel  
                   // -> Rechte 7-Segment-Anzeige komplett aus  
  
PDR00_P7 = 0;      // Pin 7 von Port 00 auf Low-Pegel  
                   // -> Punkt der rechten 7-Segment-Anzeige an
```



Beispielcode: Analog/Digital-Wandler



8 Bit oder 10 Bit Genauigkeit (wir verwenden 8 Bit)

Wandlungsmodi (z.B. mehrere Eingänge sequentiell wandeln)

- Wir verwenden **Stop Mode**: ein Kanal wird einmal pro Startsignal gewandelt
- Start- und Endkanal erhalten bei jeder Wandlung einen identischen Wert

```
unsigned char result;
```

```
// Initialisierung des AD-Wandlers
ADCS_MD    = 3;      // ADC Stop Modus
ADCS_S10   = 1;      // 8 Bit Genauigkeit
ADER0_ADE2 = 1;      // Analoge Eingänge aktivieren: AN2 + AN3
ADER0_ADE3 = 1;      // (ADER0: Eingänge AN0 bis AN7)

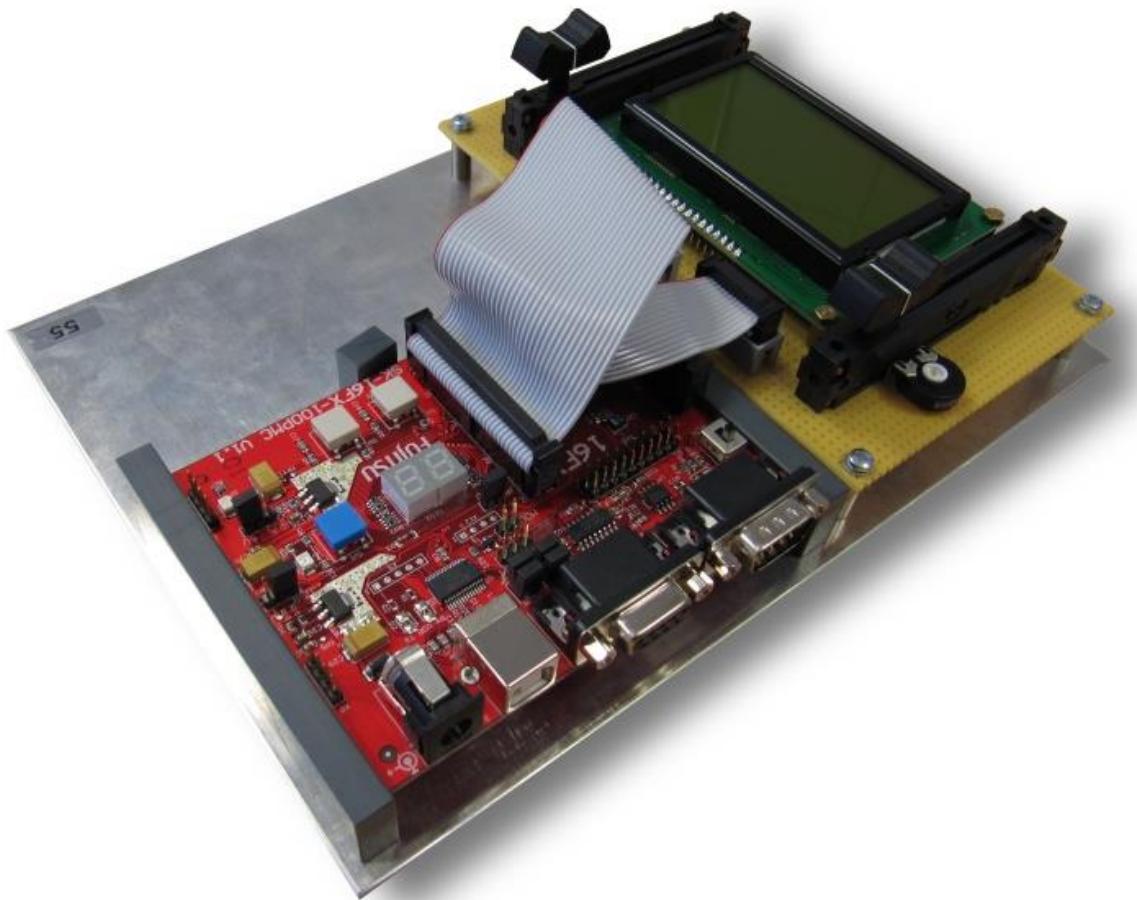
// A/D-Wandlung durchführen
ADSR = 0x6C00 + (3 << 5) + 3;          // Start- und End-Kanal 3

ADCS_STRT = 1;           // A/D-Wandler starten
while (ADCS_INT == 0) { } // Warten bis A/D-Wandlung beendet
result = ADCR1;          // Ergebnis speichern
ADCS_INT = 0;             // Bit auf 0 für nächste Wandlung
```

Viel Spaß!



TECHNISCHE
UNIVERSITÄT
DARMSTADT





TECHNISCHE
UNIVERSITÄT
DARMSTADT

FOLIEN NÄCHSTES JAHR

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. kann nach dieser Zeile nicht mehr verwendet werden – daher ist nur eine Übergabe als **const &name** sinnvoll.

Implizite Typkonvertierung unterbinden



```
#include <string>

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

private:
    std::string name;
};

void useStudent(const Student &student) {
}

int main() {
    Student mike("Mike");
    useStudent(mike);
    useStudent(std::string("Sarah"));
}
```

Ohne **explicit** kann man
useStudent auch so aufrufen wegen
impliziter Typkonvertierung.