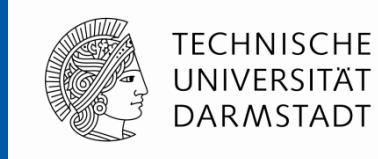


# Programmierpraktikum C und C++



```
#include <iostream>

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



ES Real-Time Systems Lab  
Prof. Dr. rer. nat. Andy Schürr

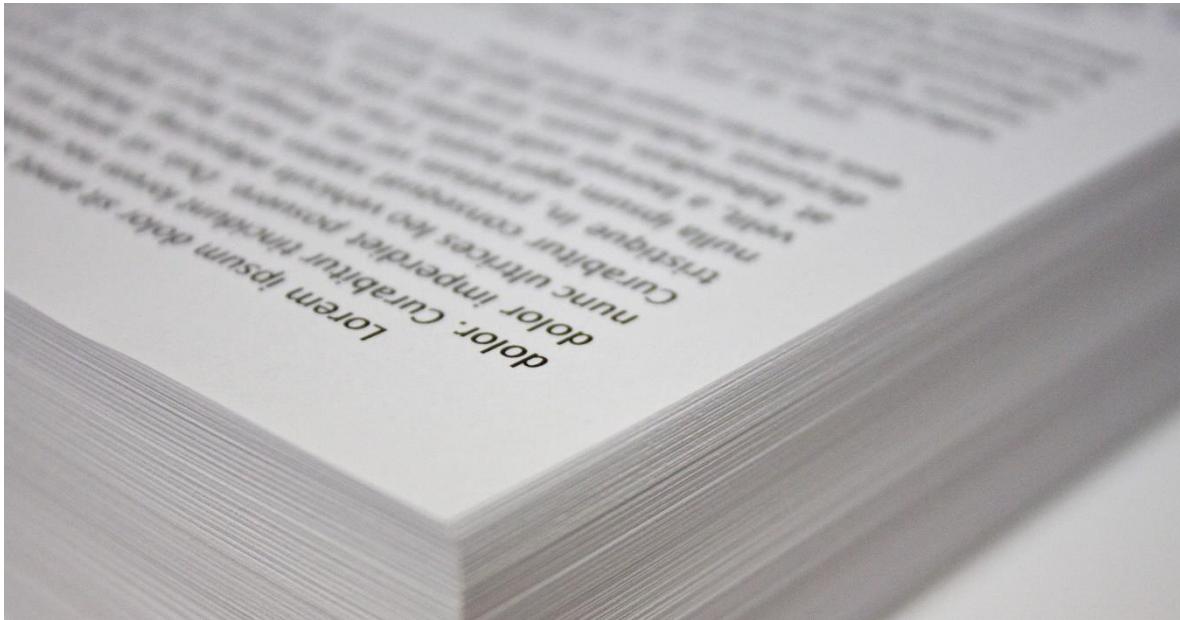
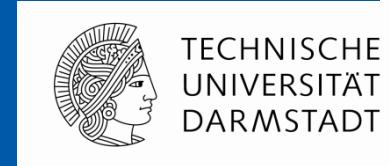
Dept. of Electrical Engineering and Information Technology  
Dept. of Computer Science (adjunct Professor)  
[www.es.tu-darmstadt.de](http://www.es.tu-darmstadt.de)

Roland Kluge

[roland.kluge@es.tu-darmstadt.de](mailto: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](http://www.es.tu-darmstadt.de)

**Roland Kluge**

[roland.kluge@es.tu-darmstadt.de](mailto:roland.kluge@es.tu-darmstadt.de)



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

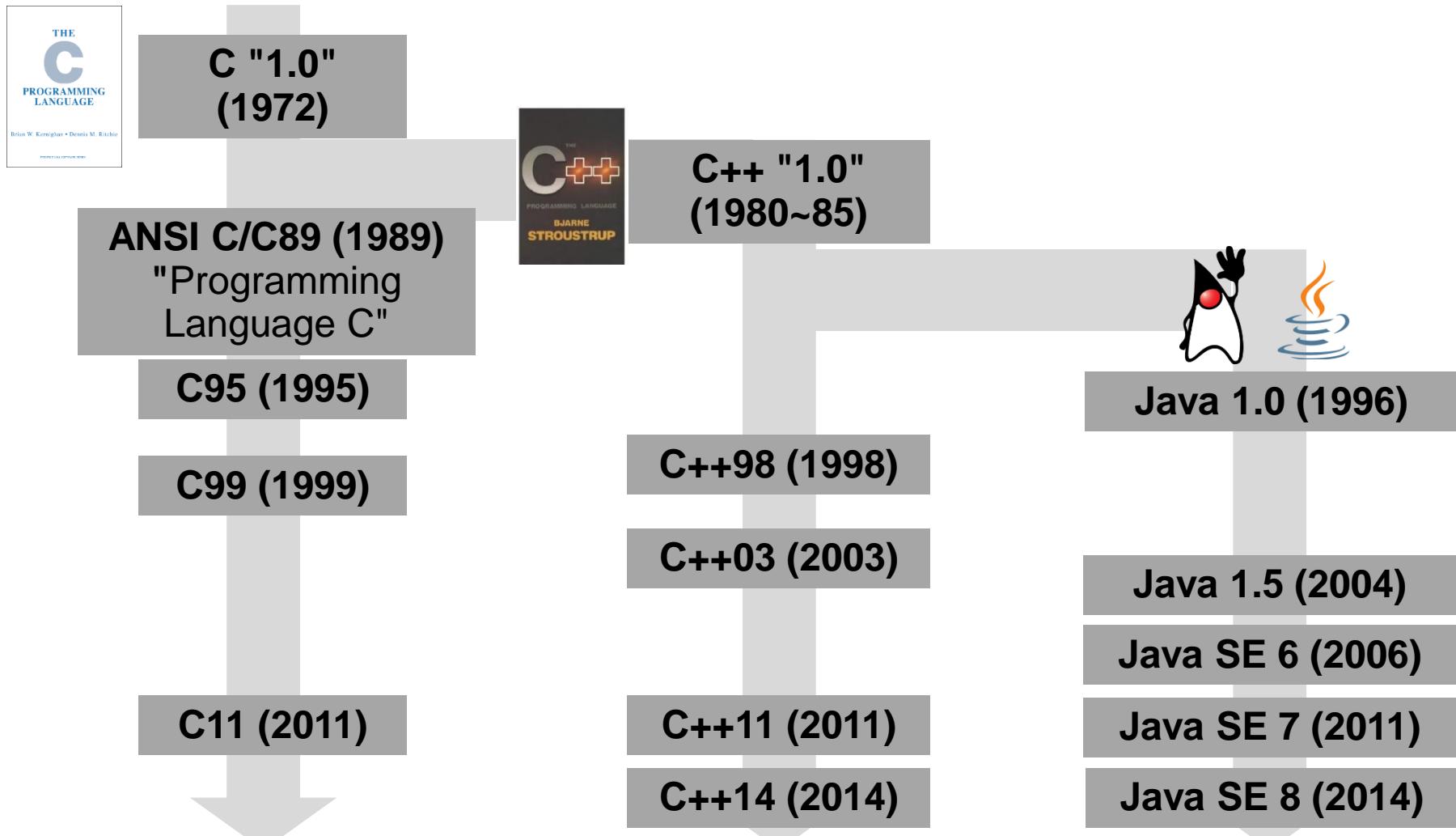
## Idee des Praktikums

- Vorlesung vermittelt Konzepte
- Übung vermittelt praktische Kenntnisse

## Basisvoraussetzungen

- Allgemeine Programmiererfahrung
- Kenntnisse in Java

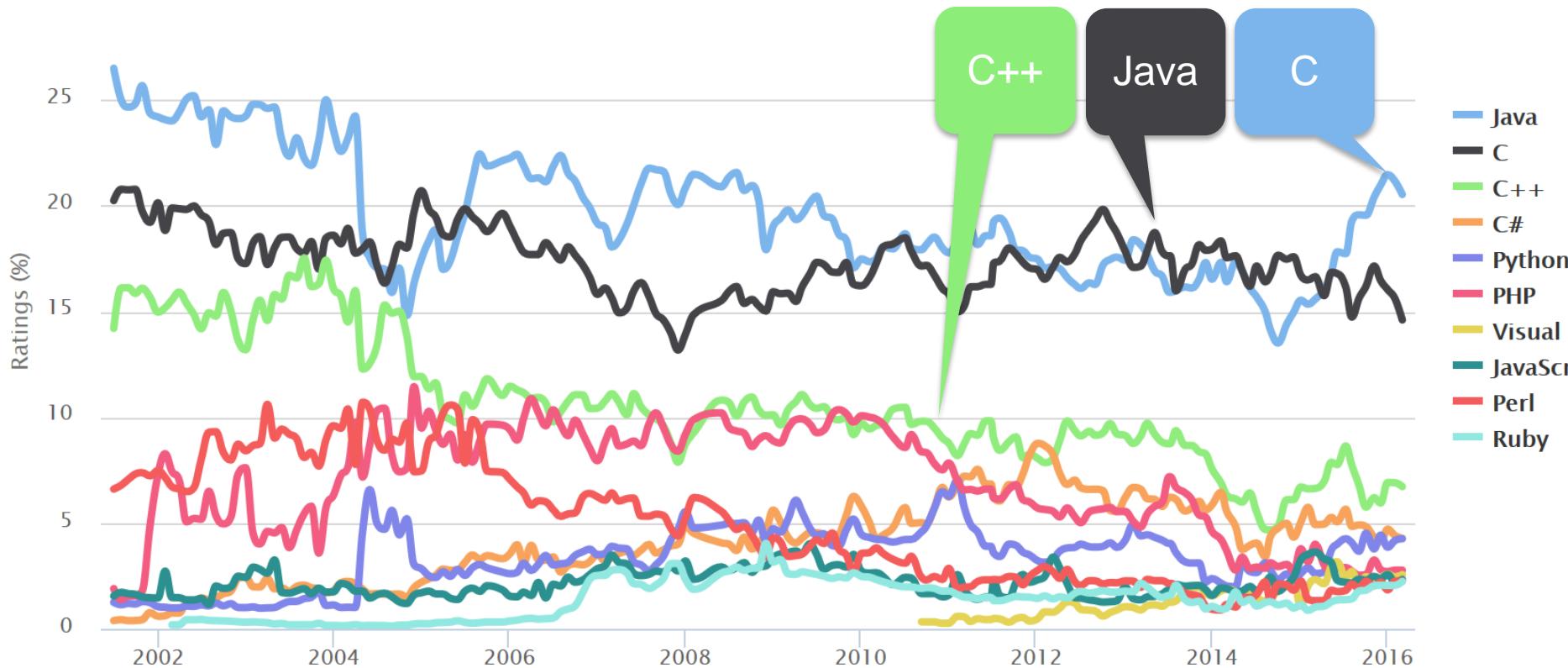
# Zusammenhang zwischen C, C++ und Java



# Wie wichtig sind C und C++? Der TIOBE-Index.



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



Rank	Rank Change	Language	Ratings	Change
1	2	Java	20.528%	+4.95%
2	1	C	14.600%	-2.04%
3	4	C++	6.721%	+0.09%
4	5	C#	4.271%	-0.65%

[http://www.tiobe.com/tiobe\\_index?page=index](http://www.tiobe.com/tiobe_index?page=index)

ES – Real-Time Systems Lab





## Grundlagen

- Projektstruktur, Kompiliervorgang, allgemeine Konzepte

## Speicherverwaltung

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

## Objektorientierung

- Besonderheiten von C++

## Fortgeschrittene Themen

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

## Embedded C

- Besonderheiten einer Hardwareplattform



## Jeden Tag

09:00 – ca. 16:00 im Electronic Classroom (S3|21 1)

## Anwesenheitspflicht

- Ausnahmen **persönlich genehmigen lassen** (Klausur, Krankheit)
- Wer **mehr als 2 Kontrollen** fehlt (**egal wieso**), darf leider nicht an der Klausur teilnehmen!
- Anwesenheitsbescheinigung **kann** in folgende Jahre "**mitgenommen**" werden.

## Ansprechpartner

Roland Kluge	(Vorlesung, Übung, Moodle)
Laurenz Kamp	(Übung, Moodle)
Philipp Jonczyk	(Übung, Moodle)

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

# Regeln für den Electronic Classroom

**Die Hardware ist brandneu (die Tische leider nicht ☹).**

**Es gelten von der Pooladministration klare Regeln.**

**Bitte...**

1. Kein Essen
2. Kein Trinken
3. Keine Kabel abmontieren / umstecken

**Bei wiederholtem Verstoß kann ein Teilnehmer des Praktikums verwiesen werden**

# Klausur



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Termin

Datum: 11. Oktober 2016

Uhrzeit: 16:15 - 17:45 (90min Bearbeitungszeit)

Raum: S1|01 A03+A04

Einsicht am  
24.+25.10.2016  
geplant.

## Inhalt

Tag 1 – Tag 4: C++-Programmierung

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!)

# Übungsmaterial



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Virtuelle Maschine:

- Downloadbereich: <http://www.es.tu-darmstadt.de/studentftp/cppt/>
- User: ccppp, PW: C++Praktikum2016@ES

## 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/>

Siehe auch *cheatsheet.pdf*

## Fachliche Fragen bitte IMMER im Moodle:

<https://moodle.tu-darmstadt.de/course/view.php?id=6546>

# Übungsmaterial (II)



## Übungsblätter (in <https://github.com/Echtzeitsysteme/tud-cpp-exercises>)

- *day1.pdf* – C++-Grundlagen
- *day2.pdf* – Speichermanagement in C++
- *day3.pdf* – Objektorientierung
- *day4.pdf* – Fortgeschrittene Themen
- *day5.pdf* – Mit embedded C warm werden
- *day6.pdf* – Ideen für eigene Projekte mit dem µC-Board
- *elevator.pdf* – Aufzugszenario aus der Vorlesung selber implementieren
  - Geht über alle Tage hinweg
  - Gute Vorbereitung für die Klausur
- *cheatsheet.pdf*
  - CodeLite
  - git
  - VirtualBox

# Demo: Virtuelle Maschine



1. Herunterladen der VM (URL, User, PW: siehe vorige Folie)

2. Importieren der Appliance  
*praktikum2016\_v4.ova*

## WICHTIG (f. Pool):

- Beim Importieren muss der Pfad für das **Virtuelle Plattenabbild** angepasst werden, sodass die VM in **C:\vms** liegt – 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.

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"/>
Netzwerkkardapter	<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

Zuweisen neuer MAC-Adressen für alle Netzwerkkarten

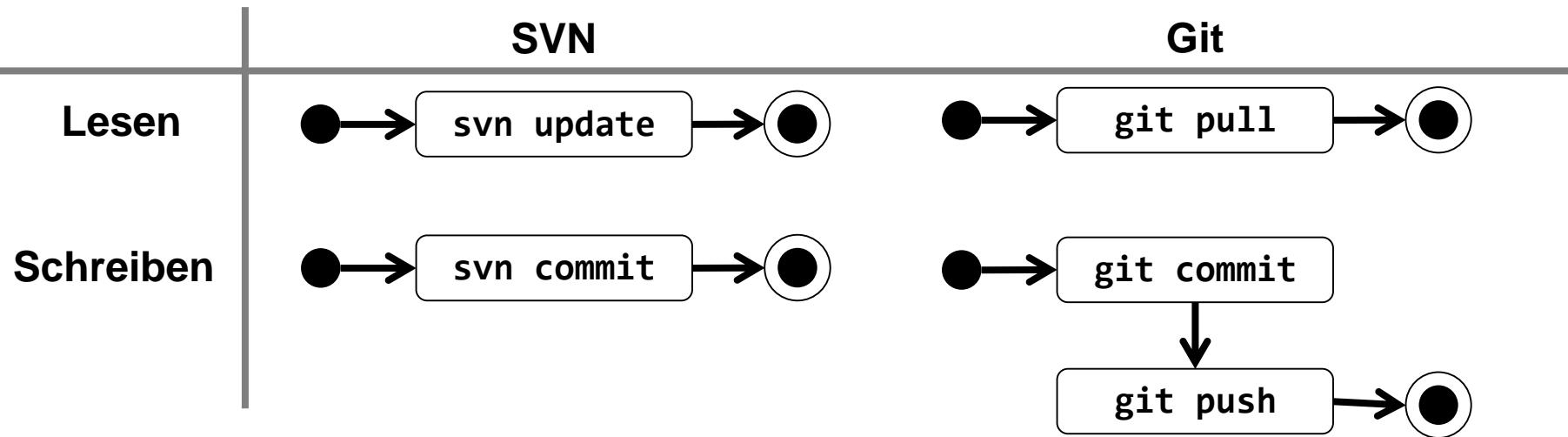
Standardeinstellungen  Importieren

# Ein paar Worte zu Git



## Bereitstellung der Vorlesungs- und Übungsunterlagen

- Bereits auf der VM ausgecheckt, aber **regelmäßiges Pullen** sinnvoll!
- **Vorlesung** <https://github.com/Echtzeitsysteme/tud-cpp-lecture>
- **Übung** <https://github.com/Echtzeitsysteme/tud-cpp-exercises>
- **Wichtig:** git kann nur "pullen", wenn keine versionierten Dateien verändert sind. → Separates Repo für eigenen Code erstellen.





# ERGÄNZENDE RESSOURCEN

# Literaturvorschläge



*Bruce Eckel:* Thinking in C++ (frei verfügbar <http://mindview.net/Books/TICPP/ThinkingInCPP2e.html>)

*Mike Banahan:* The C Book (frei verfügbar: [http://publications.gbdirect.co.uk/c\\_book/](http://publications.gbdirect.co.uk/c_book/))

*Scott Meyers:* Effective C++ & More Effective C++

*Helmut Schellong:* Moderne C Programmierung [Springer]

*Ralf Schneeweiß:* Moderne C++ Programmierung [Springer]

*Jürgen Wolf:* Grundkurs C [Galileo] & Grundkurs C++ [Galileo]

*Bjarne Stroustrup:* Einführung in die Programmierung mit C++

Kompakt und  
günstig

*TU München:* Grundkurs C/C++

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

*FH Regensburg:* Programmieren 1

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

*Heinz Tschabitscher:* Einführung in C++

[http://ladedu.com/cpp/zum\\_mitnehmen/cpp\\_einf.pdf](http://ladedu.com/cpp/zum_mitnehmen/cpp_einf.pdf)

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

CProgramming.com <http://www.cprogramming.com/>

# Alternative Veranstaltungen an der TU Darmstadt



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- 20-00-0747-pr **Modern C++ Programming (Blockpraktikum)**  
<https://www.informatik.tu-darmstadt.de/de/fachbereich/lehrbeauftragte-und-gastdozenten/modern-c-programming/>
  - zuletzt 2014
- 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](http://www.rmr.tu-darmstadt.de/lehre_rmr/vorlesungen_rmr/wintersemester/programmierung_aut/index.de.jsp)
  - starker Fokus auf Grundlagen

# Online C++-Referenzen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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

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

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

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

The screenshot shows the en.cppreference.com website. The top navigation bar has tabs for 'C++ reference', 'C++98', 'C++03', 'C++11', and 'C++14'. The main content area is titled 'C++ reference'. It contains several sections: 'Language' (ASCII chart, Compiler support), 'Preprocessor', 'Keywords', 'Operator precedence', 'Escape sequences', 'Fundamental types', 'Headers', 'Concepts', 'Utilities library' (Type support, Dynamic memory management, Error handling, Program utilities, Date and time, bitset, Function objects, pair – tuple (C++11), integer\_sequence (C++14)), 'Containers library' (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), stack – queue – priority\_queue), 'Algorithms library', 'Iterators library', and 'Numerics library' (Common mathematical functions, Complex numbers, Pseudo-random number generation).

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



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## C++ FAQ Sections

### Overview Topics

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

### Starting From Another Language

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

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

### General Topics

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

Const Correctness,  
Referenzen,...

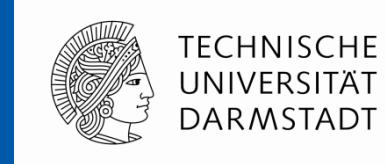


# Fragen?



# 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](http://www.es.tu-darmstadt.de)

**Roland Kluge**

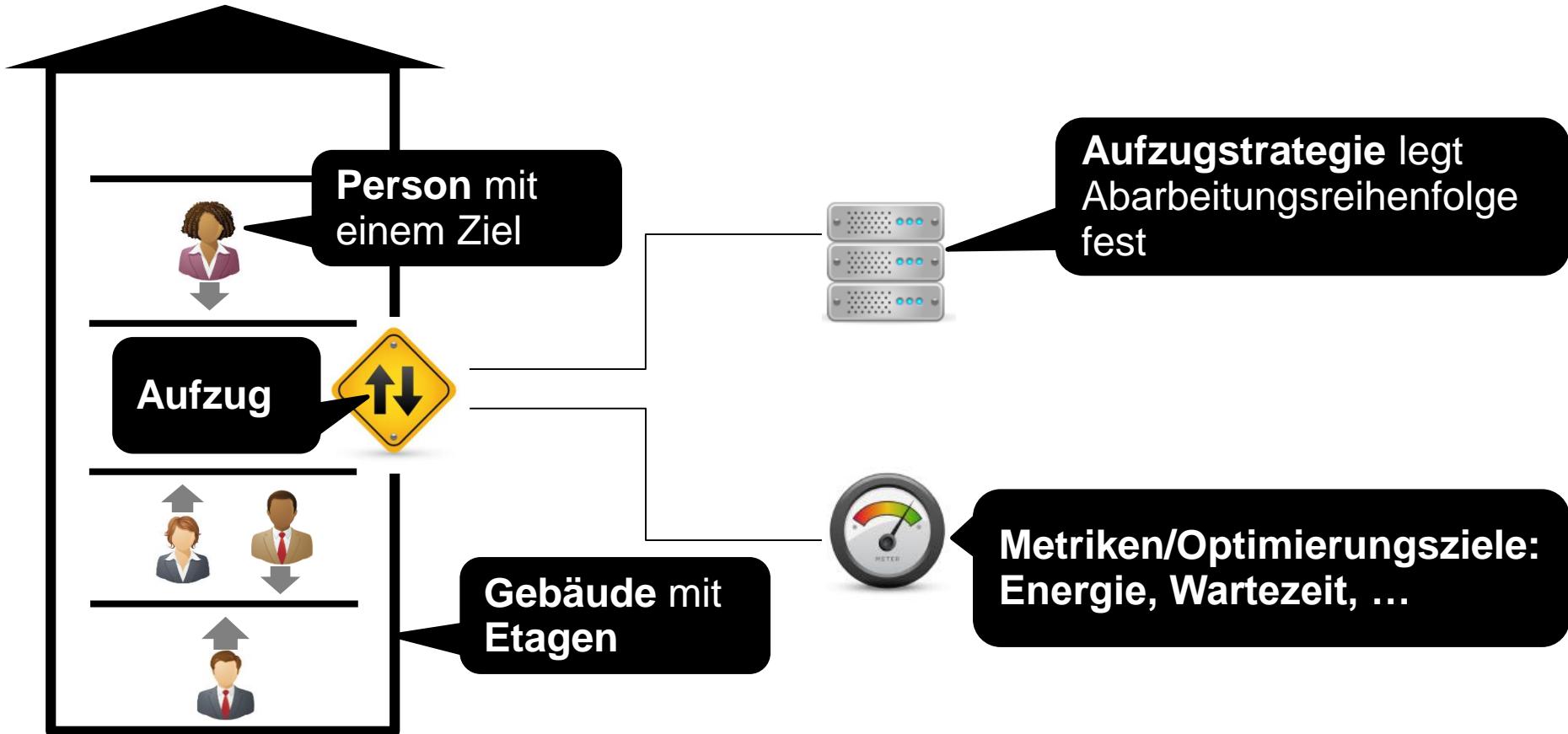
[roland.kluge@es.tu-darmstadt.de](mailto:roland.kluge@es.tu-darmstadt.de)



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# LAUFENDES BEISPIEL

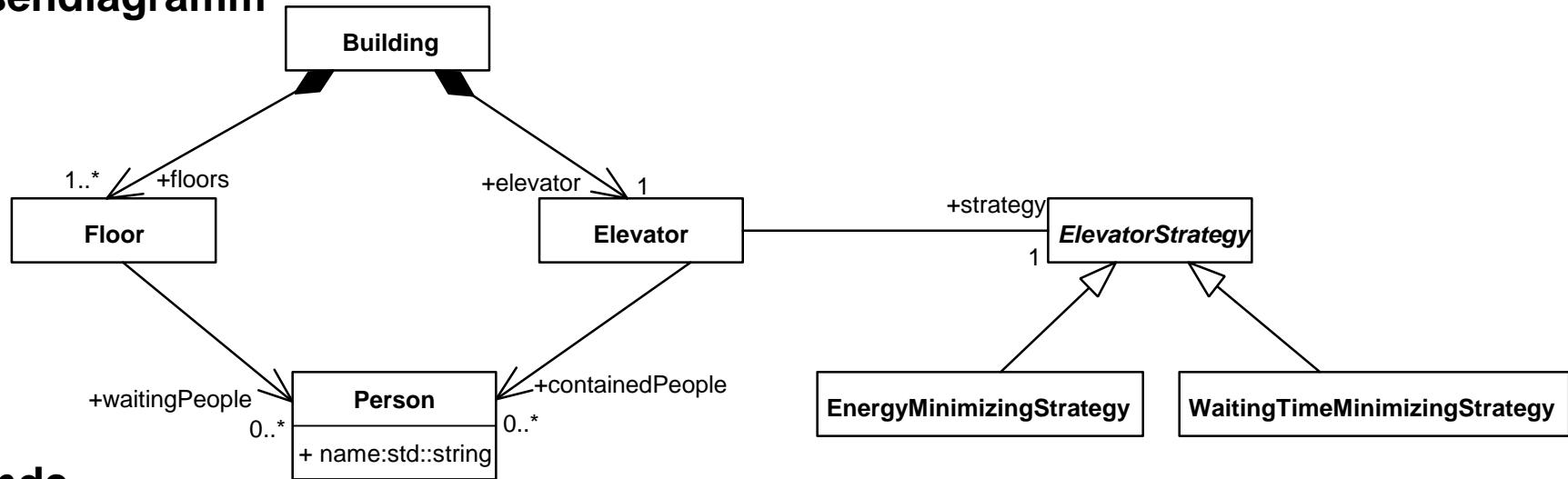
# Laufendes Beispiel: Aufzugsimulation



# Laufendes Beispiel: Klassendiagramm



## Klassendiagramm



## Legende

- Building** **ElevatorStrategy** Klasse und abstrakte Klasse
- Floor** **Person** **EnergyMinimizingS.** Assoziation mit Rollenname und Multiplizität und **std::string**-Attribut von Person
- ElevatorStrategy** **EnergyMinimizingS.** Vererbung
- Building** **Floor** Aggregation



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

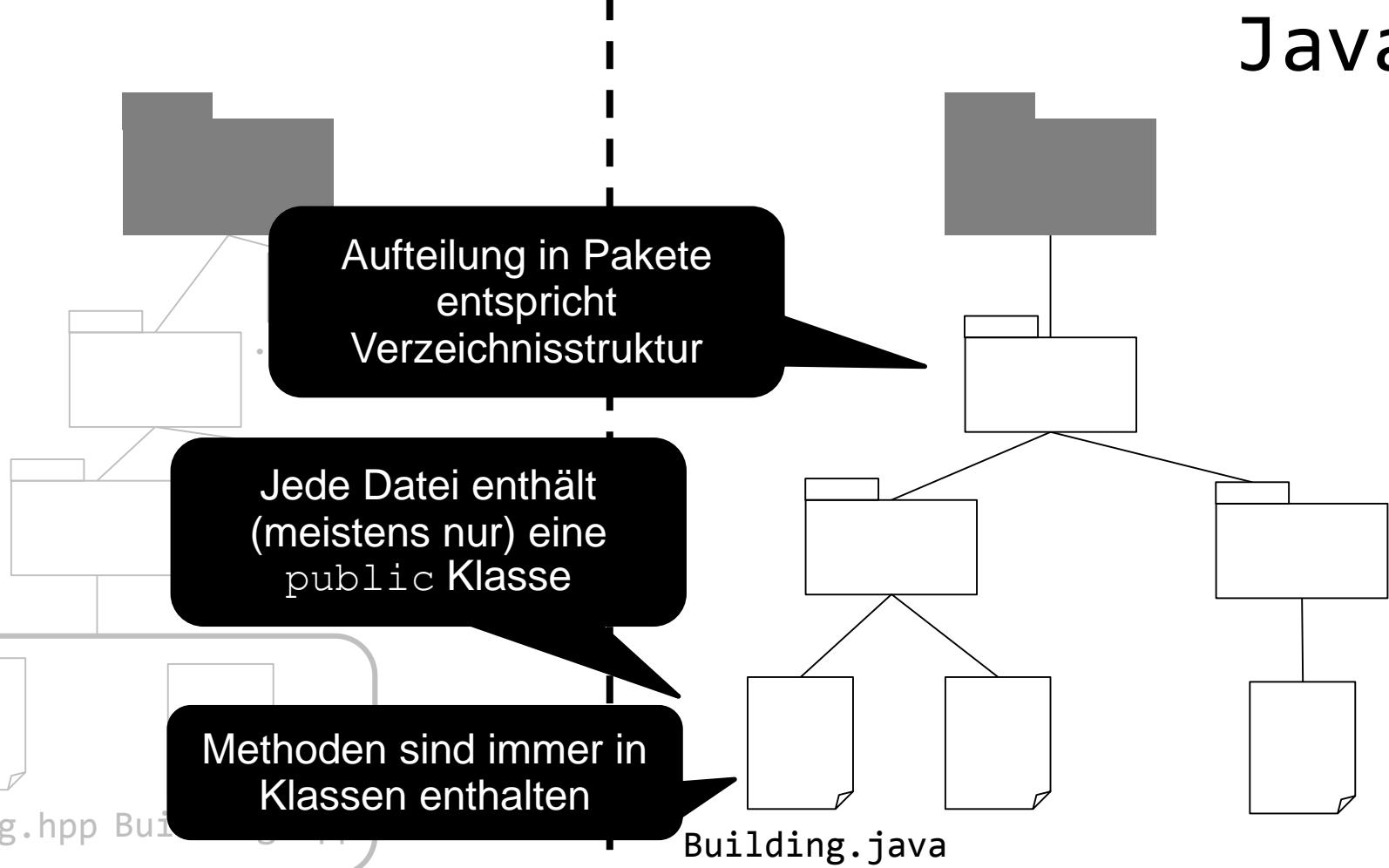
# PROJEKTSTRUKTUR

# 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! ☺

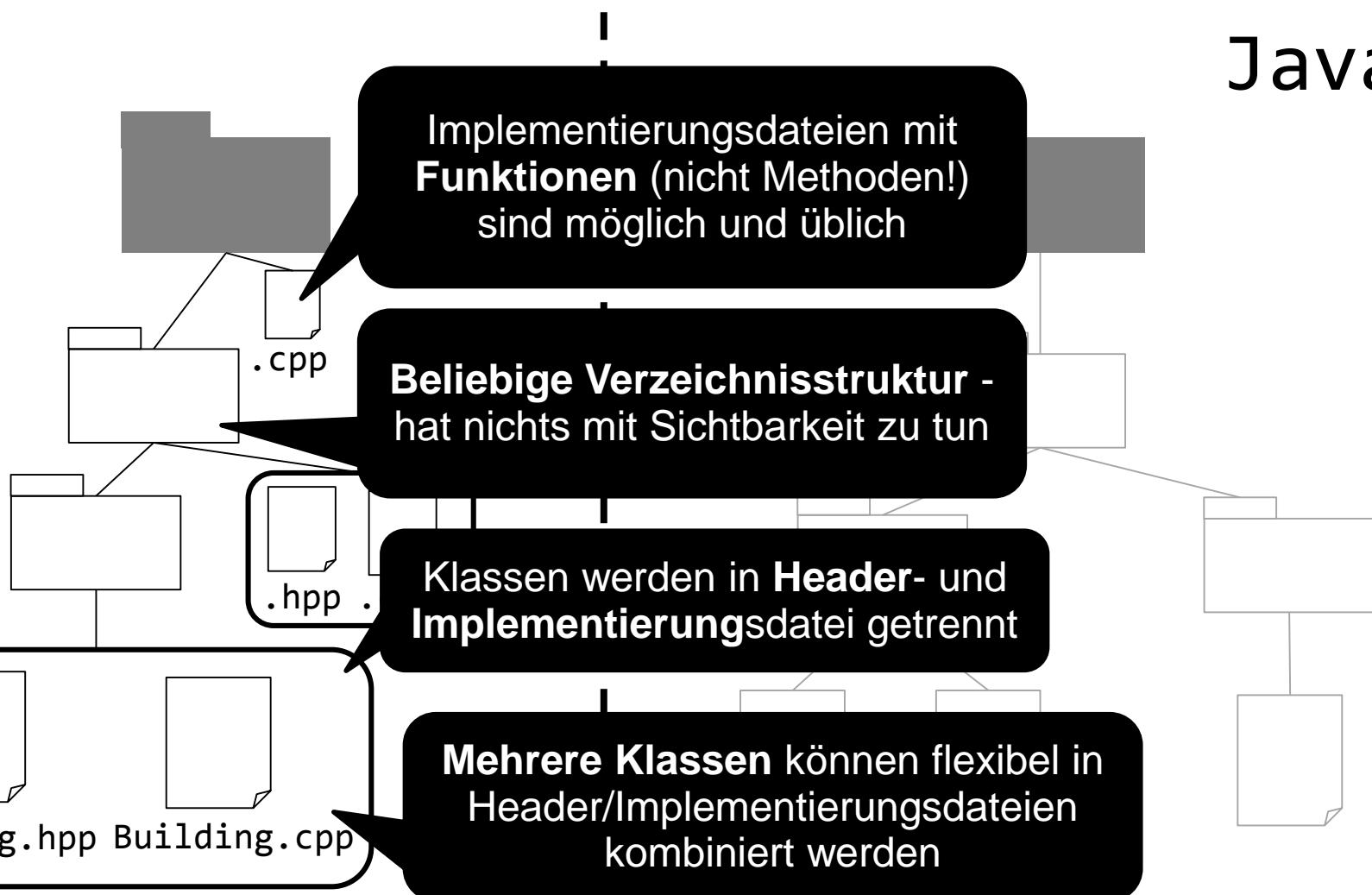


# Projektstruktur



C++

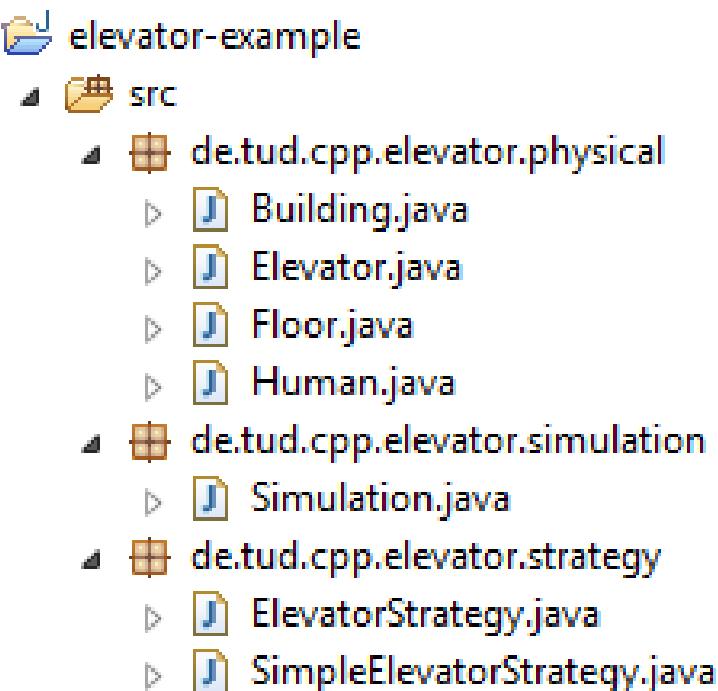
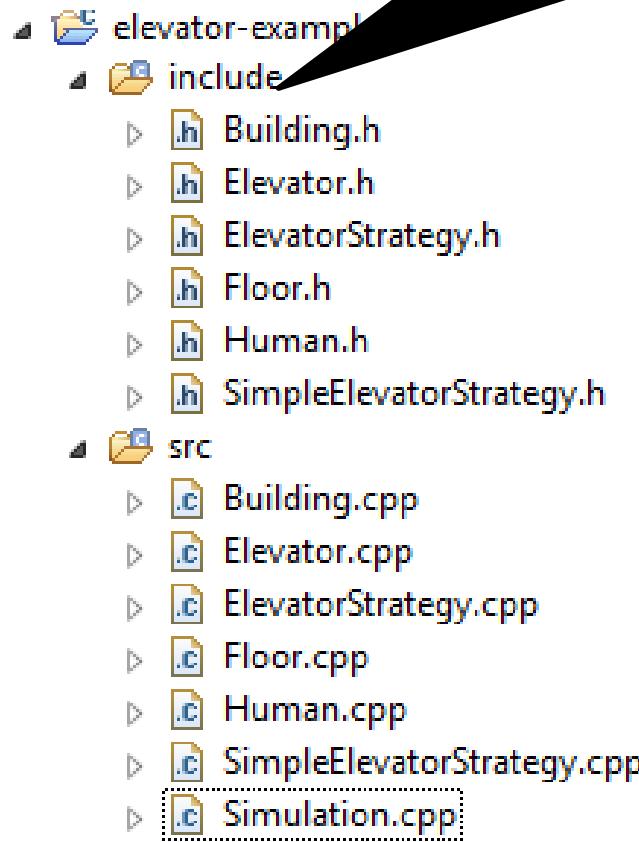
Java



C++

Best Practice: 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 numberOffFloors);
    ~Building();

    void runSimulation();

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

#endif /* BUILDING_HPP_ */
```

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

**Include-Anweisungen** wie Import-Befehle in Java:  
<...> für Bibliotheken,  
"..." für eigenen Code

**Deklaration der Klasse** ist  
wie ein Interface in Java

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

# Header und Implementierungs-Dateien



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

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

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

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

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

**Header-Datei** wird eingebunden  
("exakte" Einfügung)

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

**VORSICHT:** Sollte stets hinter den `#includes` auftreten.

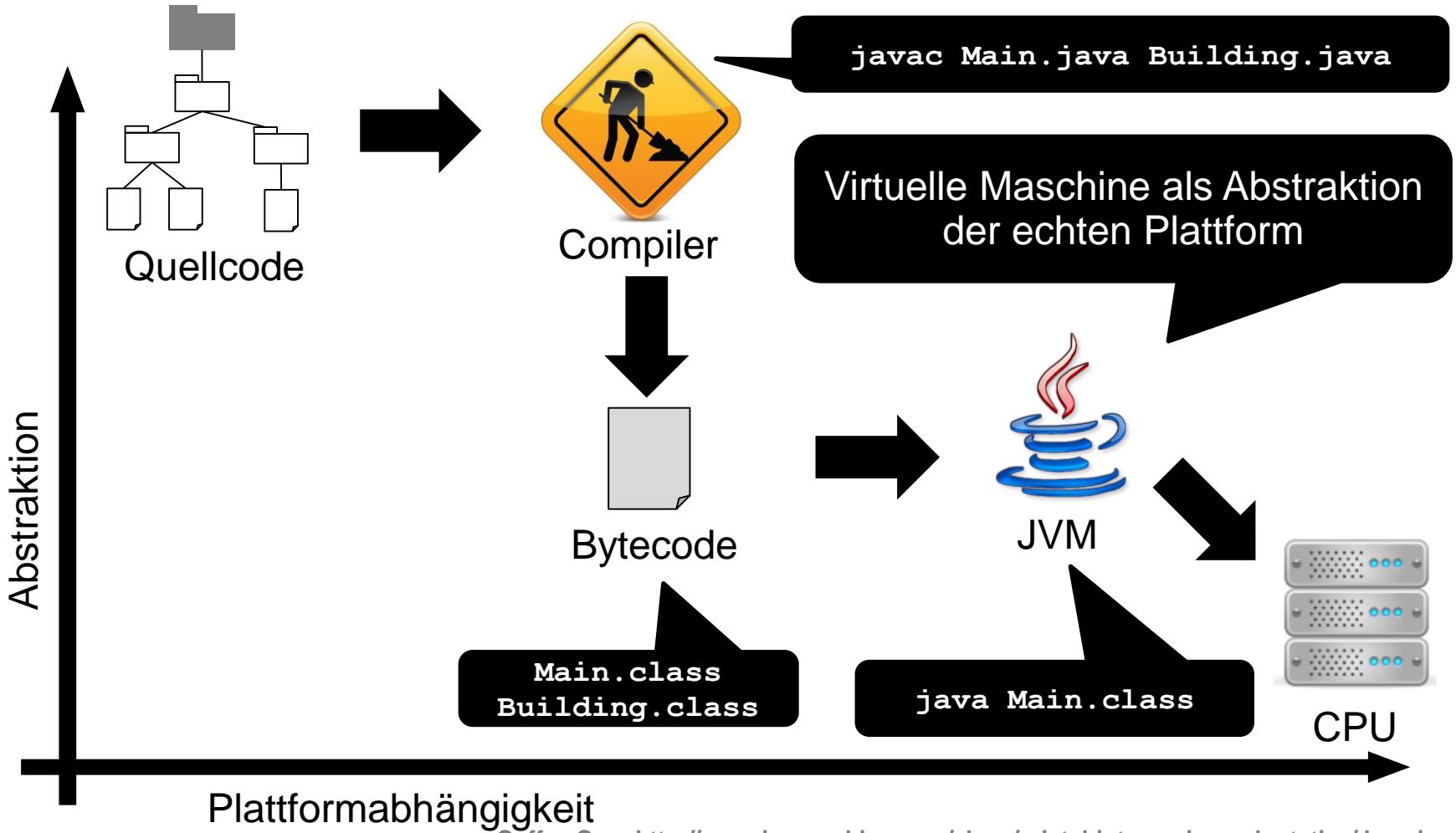
**Methoden** werden implementiert  
(Details später)



# KOMPILEIERUNG

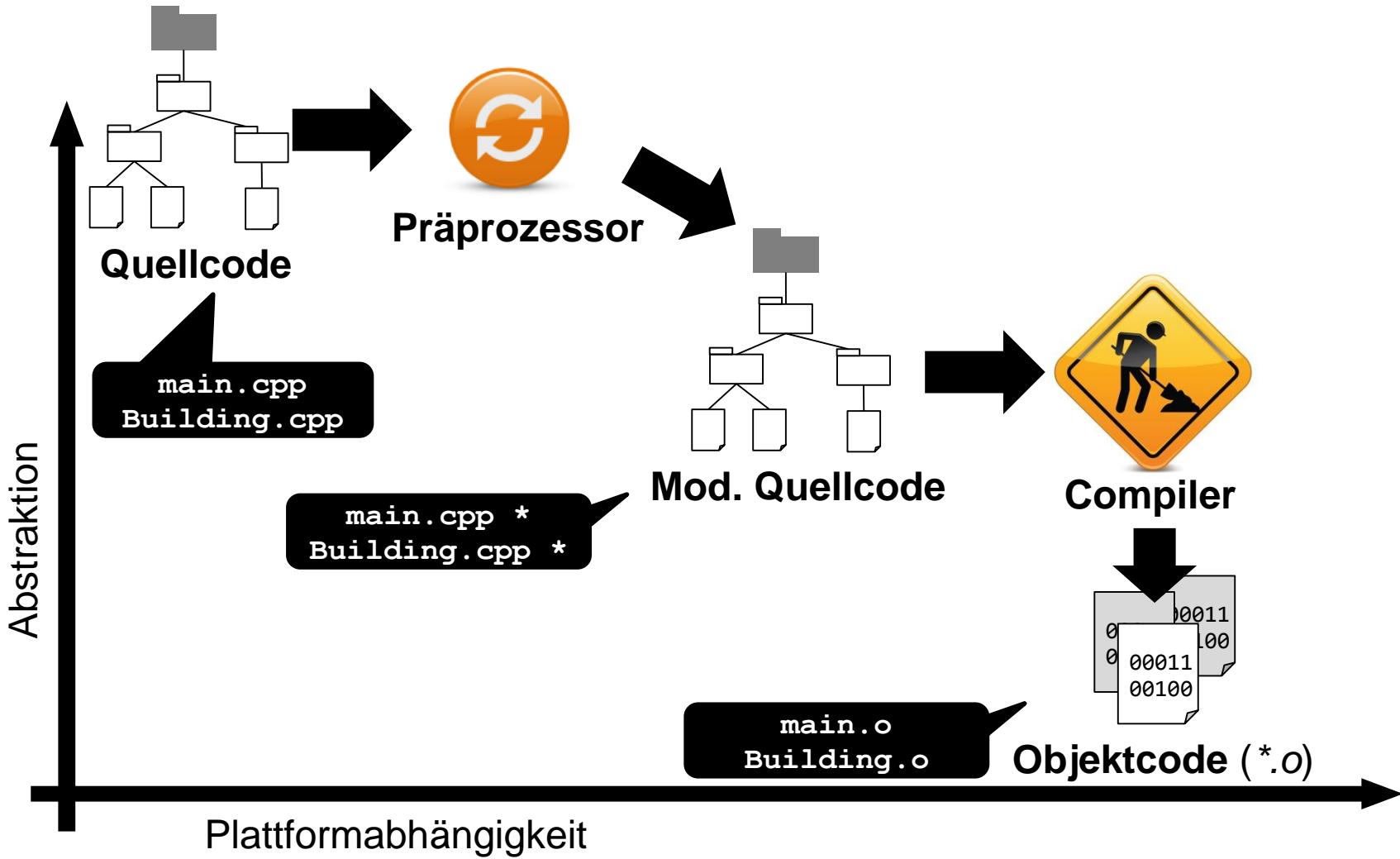


# Kompilierung in Java

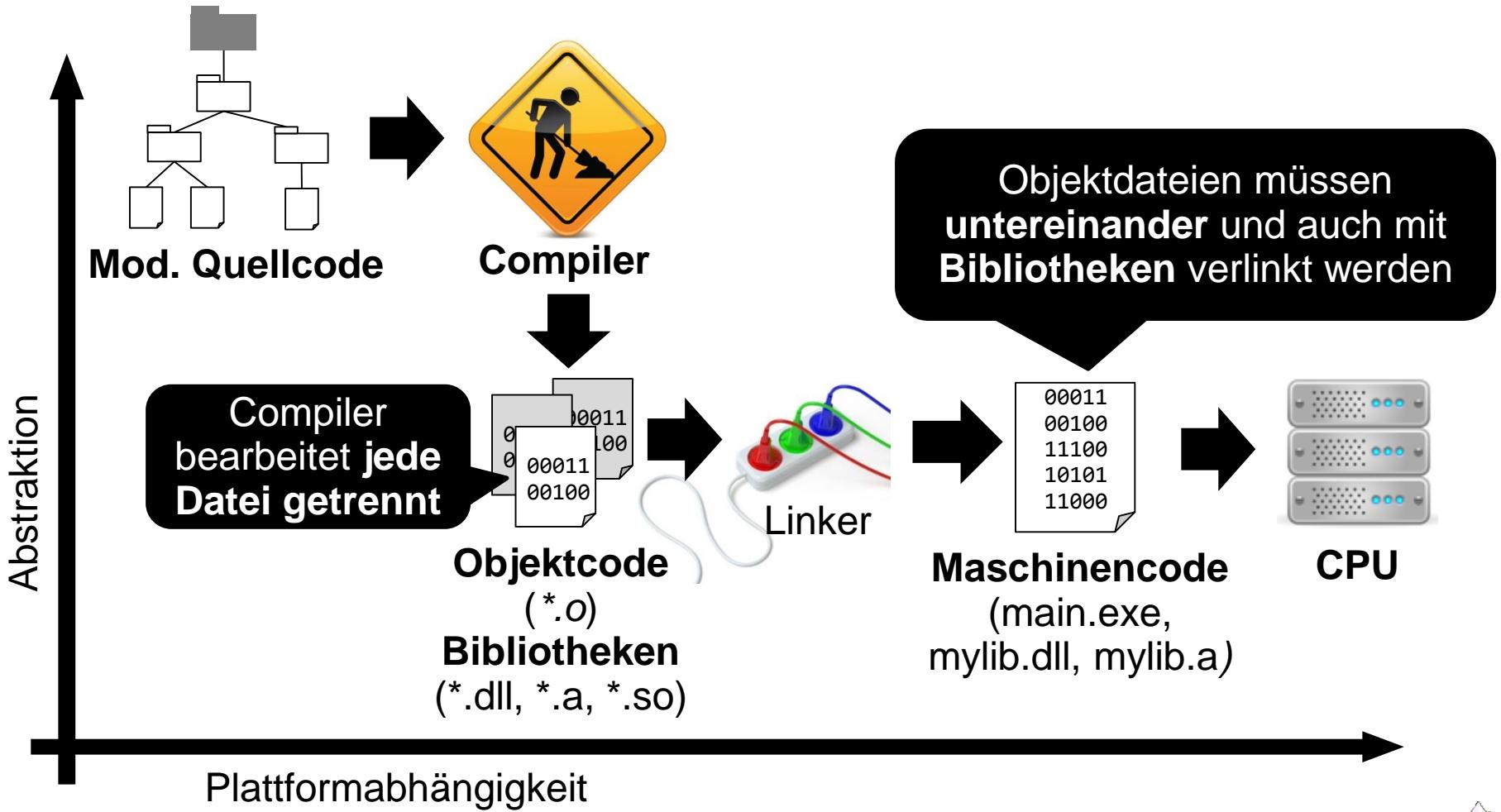


Coffee Cup: <http://www.iconarchive.com/show/cristal-intense-icons-by-tatice/Java-icon.html>

# Kompilierung für C/C++ I



# Kompilierung für C/C++ II



# Statisches und dynamisches Linken



## Statisches Linken

(Static Libraries und Shared Archives)

- Bibliothek muss zur **Linkzeit** vorhanden sein.
- "Kopie" der Bibliothek wird im Compilat (*main.exe*) abgelegt.
- Unterschied zwischen SL und SA eher klein

→ Compilat ist "**standalone**", aber (oft wesentlich) **größer** als beim dynamischen Linken

## Dynamisches Linken

(Shared Objects und DLLs)

- *Shared Objects* müssen zur **Linkzeit** und zur **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 vor mehrmaligem Einbinden von *Building.h* (Alternative: `#pragma once`)

Dadurch können wir **alle benötigten Header überall einbinden.**

## Funktionsweise:

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

## Weitere Anwendungsfälle des Präprozessors:

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

[https://en.wikipedia.org/wiki/Include\\_guard](https://en.wikipedia.org/wiki/Include_guard)

# Was passiert ohne Include Guards?



## Vor dem Präprozessor

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

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

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

## Nach dem Präprozessor

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

### Konzept

**One Definition Rule:**  
Jede Klasse/Methode/...  
durf höchstens einmal  
definiert werden

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

[http://en.cppreference.com/w/cpp/language/definition#One\\_Definition\\_Rule](http://en.cppreference.com/w/cpp/language/definition#One_Definition_Rule)

# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Warum ist die Trennung in Header- und Implementierungsdateien **hilfreich**?

Warum ist die Trennung in Header- und Implementierungsdateien **eine Fehlerquelle**?





- **Grundlegendes Konzept** in den meisten Programmiersprachen!
- **Deklaration**
  - ... gibt an, dass ein Element (z.B. Variable, Funktion, Klasse) **existiert** (→ Größe im Speicher etc.) **ohne** ihm dabei einen **konkreten Wert** zuzuweisen.
  - Beispiele: `int x; void myFunction(); class MyClass;`
- **Definition**
  - ...belegt ein Element mit einem **konkreten Wert**
  - Je nach Element ist eine Redefinition möglich
  - Beispiele: `x=3; void myFunction() /* function def. */;`  
`class MyClass /* class def. */;`
- Deklaration und Definition können **gleichzeitig geschehen**
  - Wenn möglich, vorzuziehen!
  - Trennung erlaubt es aber, **zyklische Abhängigkeiten aufzubrechen**.
  - z.B. `int x = 3;` (Rest gleich wie bei Definition)

Praktisches Beispiel: [http://www.cprogramming.com/declare\\_vs\\_define.html](http://www.cprogramming.com/declare_vs_define.html)

# 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.
- **Heutzutage:** Nur ein **Hinweis** an den Compiler – nicht "verpflichtend".
- Heute **nicht mehr notwendig**, da der Compiler automatisch über Optimierungen entscheidet (Flags -O1, -O2, -O3, ...)

[https://en.wikipedia.org/wiki/Inline\\_function](https://en.wikipedia.org/wiki/Inline_function)

# class vs. struct vs. union



- In C++ gibt es (mind.) drei Wege zur Impl. "komplexer Datentypen".
- **struct**
  - **Geerbt von C** ( $\rightarrow$  µC-Teil), u.a. für Binärkompatibilität (z.B. `<ctime>`)
  - In C++: **Konstruktor, Methoden und Vererbung** möglich; Unterschied zu `class`: standardmäßig sind alle Member **public**
- **class**
  - Standardmittel in C++
- **union** [eher exotisch]
  - **Spezialdatentyp**, zur Speicherung "alternativer" Member; Belegung ist klar vom Kontext.
  - Höhere **Effizienz** durch gemeinsame Speichernutzung

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

```
union ResultValue {  
    int exitCode;  
    bool flag;  
};  
  
ResultValue result1;  
result1.exitCode = 3;  
result1.flag = true;
```

<https://blogs.mentor.com/colinwalls/blog/2014/06/02/struct-vs-class-in-c/>  
<http://en.cppreference.com/w/cpp/language/union>

# 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 .cpp-Datei?





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**

# In C(++) ist die Reihenfolge wichtig!



- Der C++-Compiler analysiert ganze dumm (dafür effizient) jede Datei von **vorne nach hinten**. (In Java: beliebige Reihenfolge)

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

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

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

Konzept

Funktionsprototyp



---

Eine lose Übersicht...

# WEITERE KONZEPTE IN C++

# Namenskonflikte vermeiden mit Namespaces



- **In Java:** Packages/Ordnerstruktur
  - import und import static in Java
- **In C++:** namespace{}, class{}, struct{};
  - using-Direktive zum Importieren

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

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

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

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

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

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

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

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



- Nur innerhalb einer class-Definition, für Attribute und Methoden (A+M)
  - bereichsweise, nicht attribute-/methodenweise
  - **public**: alle folgenden A+M sind unbegrenzt nutzbar
  - **protected**: alle folgende A+M sind nur in dieser und Unterklassen nutzbar
  - **private**: alle folgenden A+M sind nur in dieser Klasse nutzbar
  - **friend** f() erlaubt Funktion/Methode f() auf private A+M dieser Klasse
- Anders als in Java: keine package-/default-Sichtbarkeit
  - via ::-Operator oder **using** können alle Funktionen und public-Methoden genutzt werden
- N.B.: Auch in "Plain C" gibt es Sichtbarkeitsmodifikatoren (z.B. **static** für Funktionen)



- **In Java:**
  - Quote-Literale werden zu **java.lang.Strings**
  - Beispiel:
    - `System.out.println("Hello World".getClass());  
// java.lang.String`
- **In C++:**
  - Quote-Literale werden zu **C-Strings = char-Arrays**
  - Beispiele:
    - `const char *myString = "Hello World.;"`
    - `std::string myString2 = std::string("Hello World.");`
    - `std::string myString3 = "Hello World";  
// implicit constructor invocation`

# Standard-Bibliotheken in C++



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

flexibel, erweiterbar, `std::cout/std::cin`

Mehr Details später.

# Boost:

## "Brutschränk" für C++-Standardkomponenten



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



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

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

Herb Sutter, Andrei Alexandrescu, C++ Coding Standards

Array

Filesystem

Lambda

Odeint

Chrono

Function(al)

Math  
(advanced)

Smart Ptr

Date Time

Graph

MPI

System



- **In Java:**
  - Operatoren in **Sonderrolle, fest belegt** ("Lehre aus Erfahrung mit C++)
  - fixe Präzedenz
  - Beispiel: `++`, `--` (Postfix) vor `++`, `--`, `+`, `-`, `~, !` vor `*`, `/`, `%` vor ...
- **In C++:**
  - Operatoren als **Syntactic Sugar** und beliebig überschreibbar
  - fixe Präzedenz
  - `a + b` gleichwertig zu `operator+(a,b)` oder `a.operator+(b)`
  - Extrem wichtig: Zuweisungsoperator `operator=` (siehe später)
  - `::` vor `++`, `--` (Postfix), `()`, `[ ]`, `., ->` vor `++`, `--`, ...

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

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

[http://en.cppreference.com/w/cpp/language/operator\\_precedence](http://en.cppreference.com/w/cpp/language/operator_precedence)



- **In Java:**
  - Casts in Sonderrolle (Sprachfeature)
  - Nur Typecast: SpecialBuilding sb = (SpecialBuilding)b;
  - Laufzeitfehler bei Fehlschlag: java.lang.ClassCastException
- **In C++:**
  - Casts als reguläre Funktionen, große Vielfalt und durch Bibliotheken erweiterbar
  - `int i = (int) 3.4;` C-Stil; beliebige Umwandlung ist möglich.
  - `static_cast<int>(3.0)` Umwandlung ohne Laufzeitcheck
  - `dynamic_cast<SC*>()` Umwandlung mit Laufzeitcheck
  - `reinterpret_cast<C>(x)` beliebige Umwandlung
  - `const_cast<char*>()` Constness entfernen

# Iterierungskonzepte in C++



---

## In Java:

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

## In C++:

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

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

# Konzepte und Konventionen sind in C++ wesentlich



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- C++ "vertraut" dem Programmierer → **Alles, wirklich alles**, ist möglich.
- **Konventionen** sind in C++ wesentlich, werden tw. mittels Schlüsselwörtern spezifiziert und vom Compiler überprüft:
  - `void f() noexcept` garantiert, das f keine Exceptions wirf.
- **Konzepte:**
  - **One Definition Rule**
    - Methoden/Klassen dürfen nur einmal def. werden.
  - **Undefined Behavior (UB)**
    - UB tritt ein, wenn Code auf eine nicht-spezifizierte Weise aufgerufen wird; z.B. `x/0`, Konstanten nach `const_cast` manipulieren, `int f /*no return stmt*/`
  - **Const Correctness**
    - Schutz vor ungewollten Zustandsänderungen, vgl. **final** Variablen neuzuweisen in Java

[https://en.wikipedia.org/wiki/Undefined\\_behavior](https://en.wikipedia.org/wiki/Undefined_behavior)

[https://en.wikipedia.org/wiki/One\\_Definition\\_Rule](https://en.wikipedia.org/wiki/One_Definition_Rule)

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

Fortgeschritten: <http://en.cppreference.com/w/cpp/concept>

# 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](http://www.es.tu-darmstadt.de)

**Roland Kluge**

[roland.kluge@es.tu-darmstadt.de](mailto: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

- Typischerweise wesentlich größer als Stack
- Speicherverwaltung:  
**manuell, 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?

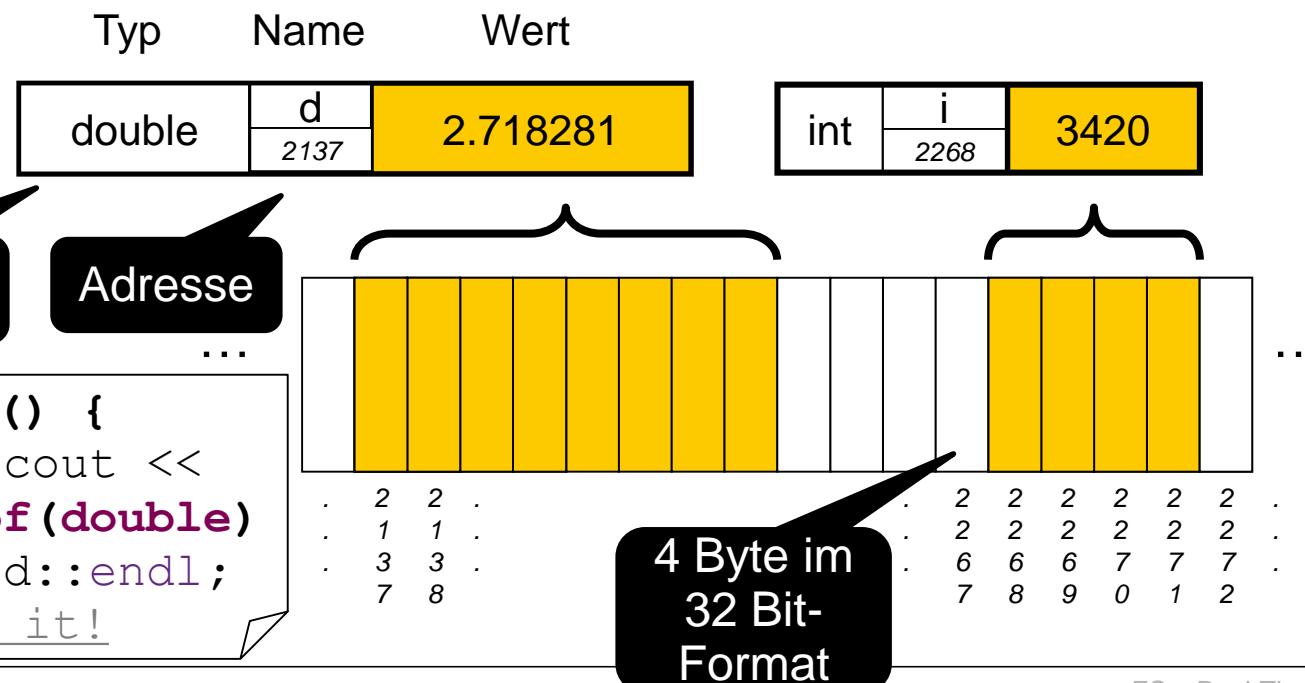


# Variablen und Zeiger: Was ist eine Variable?



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

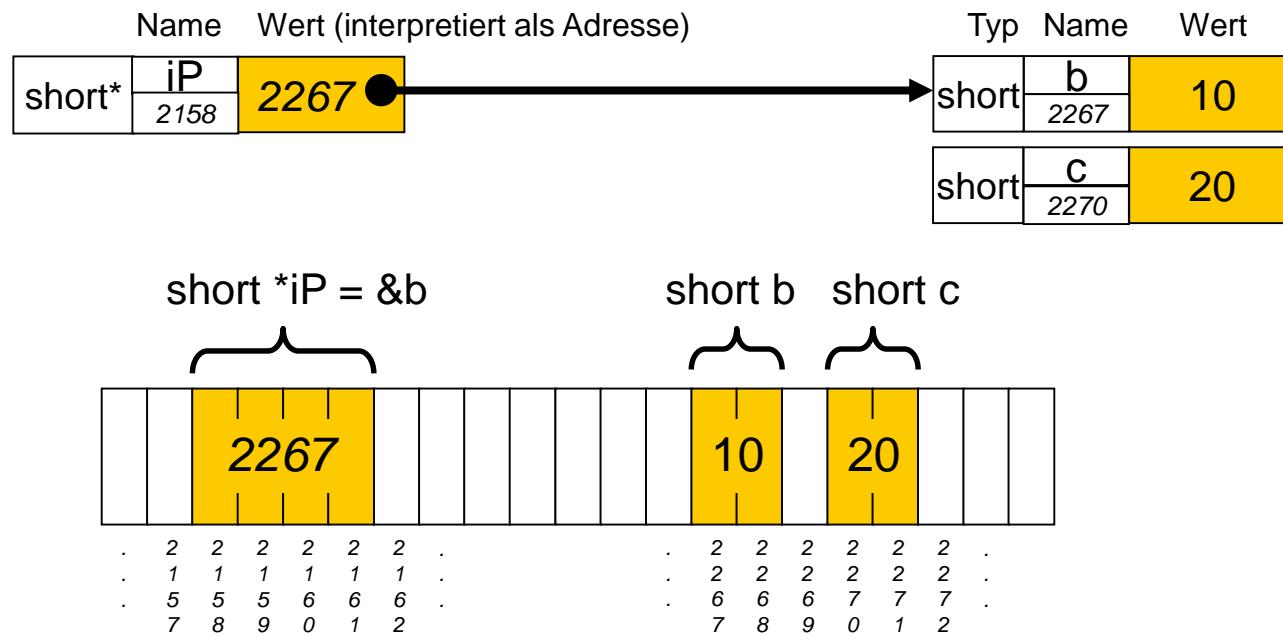
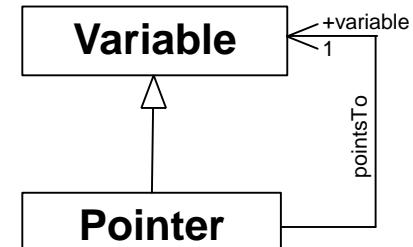


# Variablen und Zeiger: Was ist ein Zeiger?



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;
```

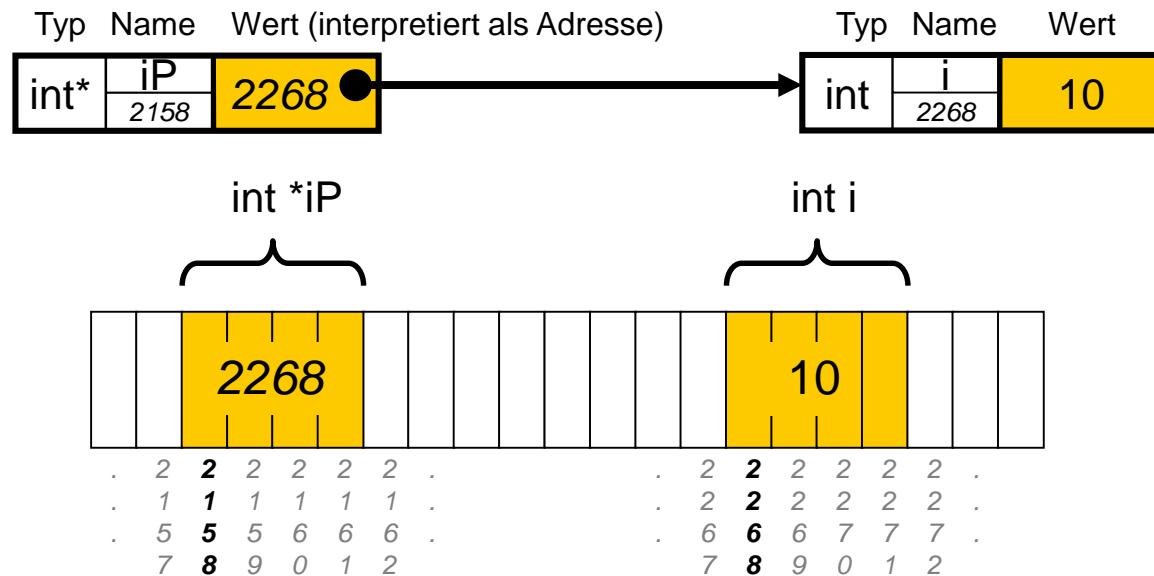
**Dereferenzierung** eines Zeigers, um den Inhalt zu erhalten

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

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

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

# 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 anzuzeigen, 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;
```

0x0

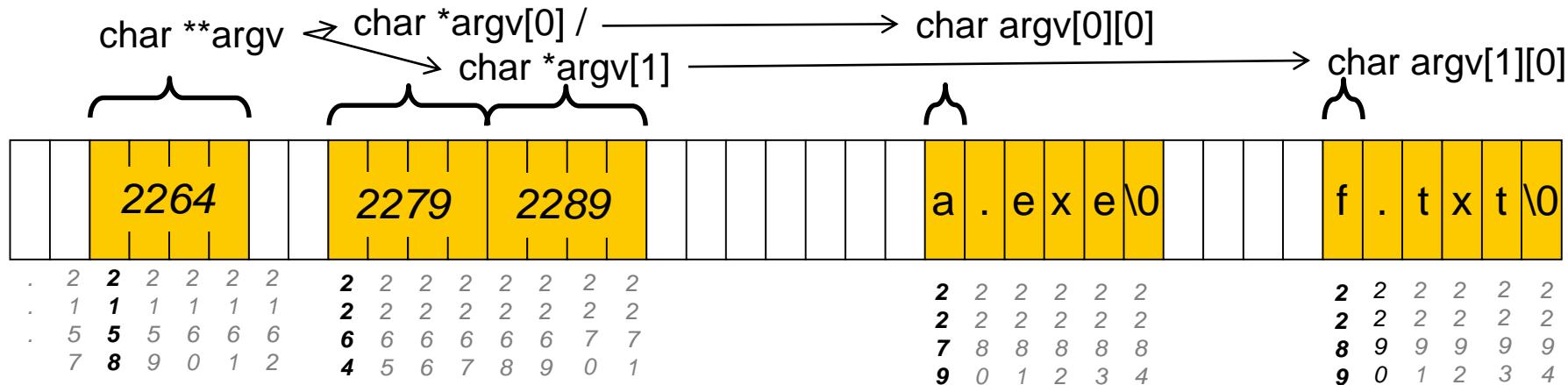
NULL  
nullptr

Wie <stddef.h>, aber mit Namespaces

# int main(int argc, char\*\* argv)



- Was passiert beim Aufruf `a.exe f.txt`?
- Strings (in C) sind Folgen von char (mit "\0" abgeschlossen)



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



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

Spezieller operator<<  
für char\*

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

2279

void\* =  
"Generischer" Pointer



## In Java:

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

## In C++:

- Syntactic Sugar: Array = Pointer auf zusammenhängenden Speicherbereich
- Problem: Längeninformation werden nicht explizit gespeichert ( $\rightarrow$  s. Übung)
- Beispiel: `int* myArray = {1, 1, 2, 3, 5, 8}; int x2 = myArray[2];`

# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Wann braucht man wirklich Zeiger?

Wieso kann man nicht einfach nur normale Variablen verwenden?





# CONST CORRECTNESS

Konzept



- Ein Programm ist const-korrekt, wenn unveränderliche (d.h. als unverändlich gemeinte) Objekte durch das Programm auch nicht verändert werden.
- Wird in C++ durch das **Schlüsselwort const** (für Typen und Funktionen) sichergestellt.
- `const int` und `int` entsprechen **zur Compile-Zeit** verschiedenen Typen, **zur Laufzeit** jedoch wird kein Unterschied gemacht

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

# 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?



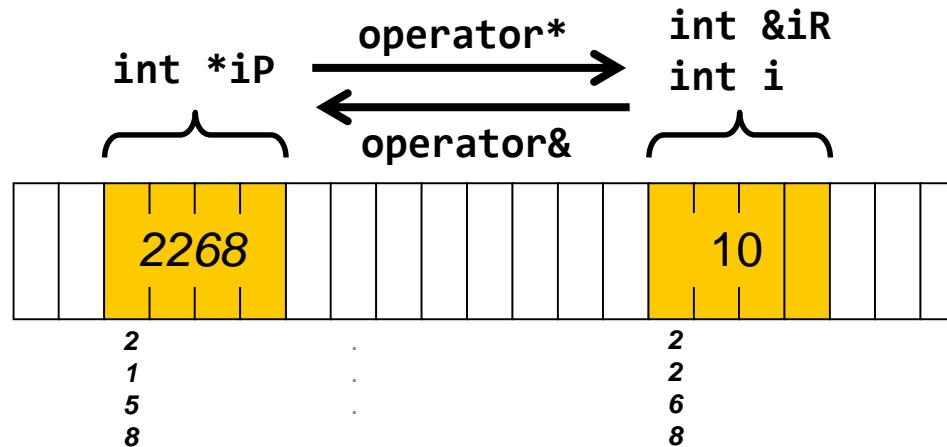
- Eine **Referenz** ist ein **Alias auf eine Variable** (braucht also keinen eigenen Speicher, verhält sich wie(!) ein const-Pointer).
- C++-Referenzen entsprechen final-Variablen in Java

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

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

Syntax wie  
für Variablen

# Beispiel: Asterisk und Ampersand



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

# **const bei Objekten**



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

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

building darf  
nicht verändert  
werden

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

# const Overloading



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

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

Auch die Elemente des Vektors sind const!

# 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*).



# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Wieso soll ich konsequent **const** verwenden?

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

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

Was ist der Unterschied zwischen

`int* iP`

und

`int *iP`

und

`int * iP`

?



# Zusammenfassung: Vorteile von `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: \* und &



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Weil es **so wichtig** ist, noch einmal: Asterisk (\*) und Ampersand (&) können je nach Auftrittsort unterschiedliche Bedeutungen haben.

Welche Bedeutung kann der **Asterisk (\*)** im Code annehmen?

Welche Bedeutung kann das **Ampersand (&)** im Code annehmen?





(Kopier-)Konstruktor, Zuweisung und Destruktor

# AUF- UND ABBAUEN VON OBJEKTEN

# Konstruktor, Destruktor und Copy-Konstruktor



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

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

**Copy-Konstruktor**

**Destruktor**

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

# Initialisierungslisten



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

impliziter Konstruktoraufruf

Klassisch: [http://en.cppreference.com/w/cpp/language/initializer\\_list](http://en.cppreference.com/w/cpp/language/initializer_list)  
`std::initializer_list`: [http://en.cppreference.com/w/cpp/utility/initializer\\_list](http://en.cppreference.com/w/cpp/utility/initializer_list)

# Delegating Constructors (C++11)



## - In Java:

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

## - In C++:

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

[http://en.cppreference.com/w/cpp/language/initializer\\_list#Delegating\\_constructor](http://en.cppreference.com/w/cpp/language/initializer_list#Delegating_constructor)  
Praktisch: <http://www.learncpp.com/cpp-tutorial/b-5-delegating-constructors/>

# 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?



# Assignment-Operator (I)



---

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

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

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

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

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

# Assignment-Operator (II)



## Vergleich mit Java

- Assignment-Operator kann in Java **nicht überschrieben/angepasst werden.**
- **Java-Primitive** (int, double,...): **Wertzuweisung**
  - int x = 1;
  - int y = x;
  - ++y // Only y is modified
- **Java-Objekte:** **Referenzzuweisung/Aliasing**
  - Floor x = new Floor();
  - Floor y = x; // Aliasing
  - y.setLevel(3); // x and y are modified



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
    // No assignment operator
    ~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.

# Compiler-generierte Methoden (I)



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

- Default-Konstruktor                    `MyClass()`                    (←wie in Java!)
- Copy-Konstruktor                        `MyClass(const MyClass &a)`
- Assignment-Operator                    `void operator=(const MyClass &a)`
- Destruktor                              `~MyClass()`
- Initialisierungsliste                    → Default-Konstruktoren für Felder

Man kann auch die **Generierung unterbinden**

- vor C++11:
  - `void operator=(Elevator &); // WITHOUT implementation`
- seit C++11:
  - `void operator=(Elevator &) = delete;`

# Compiler-generierte Methoden (II)



## In Java (Beispiel)

- Jede Klasse erbt (indirekt) von `java.lang.Object`

```
public class Elevator {}
```

wird zu durch Compiler zu

```
public class Elevator extends Object {}
```

- **Namensraum** `java.lang.*` wird automatisch eingebunden.



Hängende Zeiger und Speicherlecks

# STOLPERFALLEN BEI DER SPEICHERVERWALTUNG

[http://static.tvtropes.org/pmwiki/pub/images/Bear\\_Trap\\_7423.jpg](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

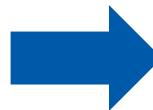


```
class EnergyMinimizingStrategy {
public:
    EnergyMinimizingStrategy() {
        cout << "Constructor called" << endl;
    }

    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();
}
```



### Ausgabe:

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

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

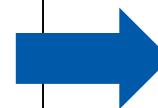
Zu erwarten ist, dass bei (5.) zunächst ein Objekt mittels Default-Konstruktor angelegt und dann mittels *operator=* überschrieben wird – C++ ist da schlauer 😊.

[https://en.wikipedia.org/wiki/Copy\\_elision](https://en.wikipedia.org/wiki/Copy_elision)

# Rückgabe von Objekten auf dem Heap



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



Creating floor [0]  
Copying floor [0]  
Made next floor [1]  
  
Next floor is floor [1]  
Destroying floor [0]



Dieses Programm enthält einen  
Fehler! 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]



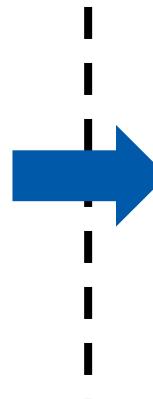
Extrem gefährlich!

# 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 = nullptr;  
  
    delete floor;  
}
```



Creating floor [0]  
Destroying floor [1]

Nach dem Löschen  
immer auf nullptr  
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



:Person

Alice

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

# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

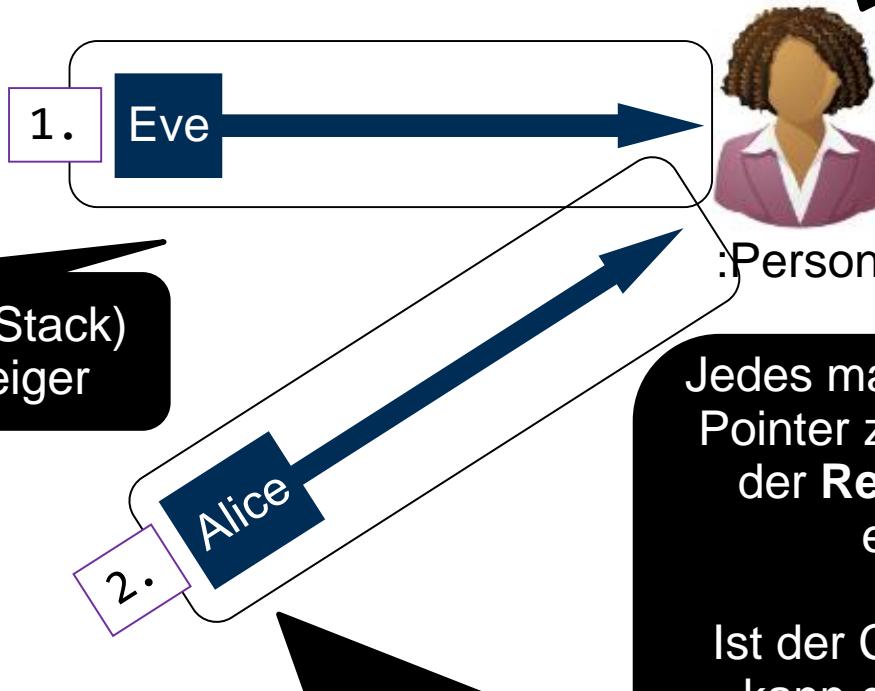
Wie könnte man das Problem lösen? Wir müssen ja irgendwie entscheiden wann ein Objekt gelöscht werden darf ...



# Mit std::shared\_ptr

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

Objekt auf dem Heap



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

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

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

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

# Person – ohne std::shared\_ptr



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

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

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

private:
    const string name;
};
```

Person.hpp

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

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

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

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

Person.cpp

# Person – mit std::shared\_ptr



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

Person.hpp

```
class Person {

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

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

private:
    const string name;
};

typedef std::shared_ptr<Person>
PersonPtr;

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

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

Person.cpp

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

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

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

# Beispiel: Klassische und smarte Zeiger



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

using namespace std;

void makeSmallTalkWith(const Person &person){
    cout << "Isn't the weather 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

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

using namespace std;

void makeSmallTalkWith(ConstPersonPtr person){
    cout << "Isn't the weather 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

# std::make\_shared



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

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

int main() {
    Person *leila = new Person("Leila");
    delete leila;

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

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

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

Die Utility-Funktion **std::make\_shared** ist vorteilhaft:

- 1) Exceptions führen nicht zu Speicherfehlern
- 2) Die Speicherallokation ist schneller

[http://en.cppreference.com/w/cpp/memory/shared\\_ptr/make\\_shared](http://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared)

# Weak SmartPointer: Motivation

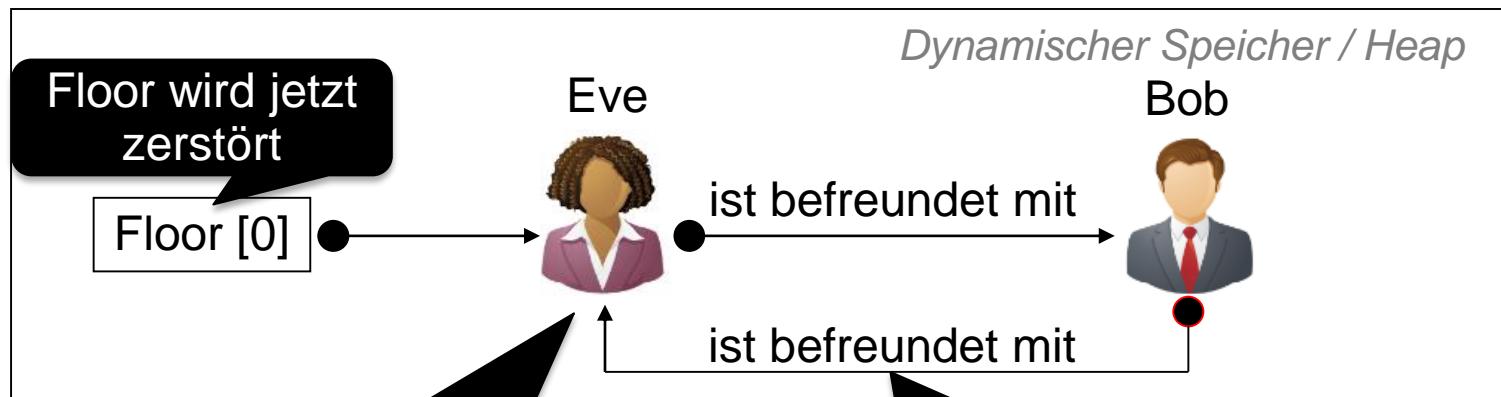


`std::shared_ptr<>` ist nicht perfekt:

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

Ablauf:

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



# Weak Pointer (`std::weak_ptr`)



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

# 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?



# Lösung: Verzicht auf Zeiger (I)



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

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

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



Welches neue Problem handeln wir uns damit ein?



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

# Lösung: Verzicht auf Zeiger (II)



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

## Ausgabe:

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

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

[https://en.wikipedia.org/wiki/Immutable\\_object](https://en.wikipedia.org/wiki/Immutable_object)

# Zusammenfassung: Übergabe und Rückgabe



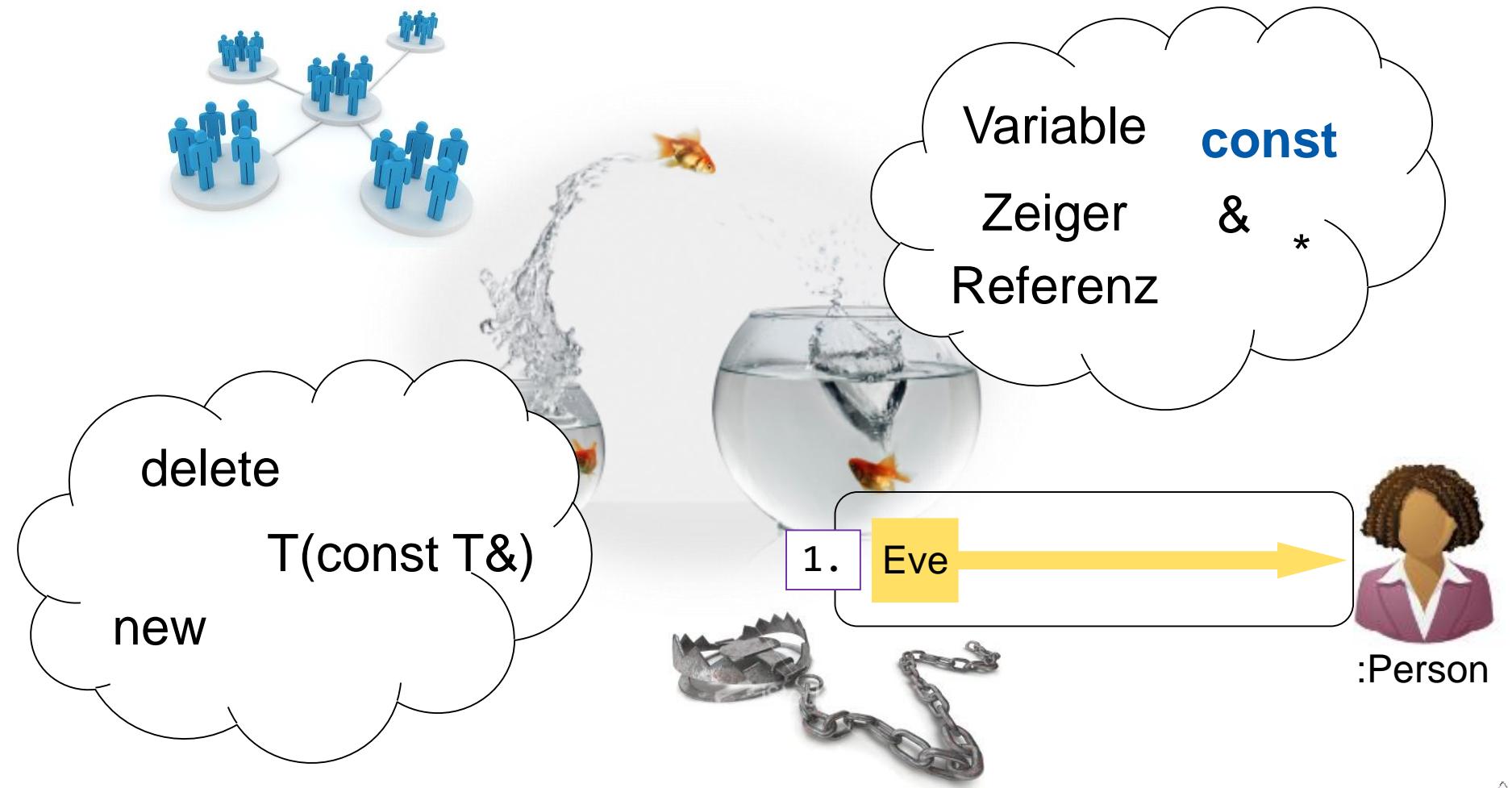
## In Java

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

## In C++

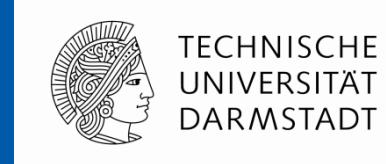
- Maximal konfigurierbar, aber anspruchsvoll.
- *Übergabe* unabhängig ob primitiver oder komplexer Datentyp
  - "pass by value"
  - "pass by reference (to const)"
  - "pass by pointer (to const)"
- *Rückgabe*:
  - "return by value" (sicher, aber Zusatzaufwand durch Kopie, evtl. Copy Elision)
  - "return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

# 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](http://www.es.tu-darmstadt.de)

**Roland Kluge**

[roland.kluge@es.tu-darmstadt.de](mailto:roland.kluge@es.tu-darmstadt.de)

# Was ist (Untertyp-)Polymorphie?



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

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

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

[https://en.wikipedia.org/wiki/Polymorphism\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Polymorphism_(computer_science))

# Ein einfaches Beispiel für Polymorphie in C++

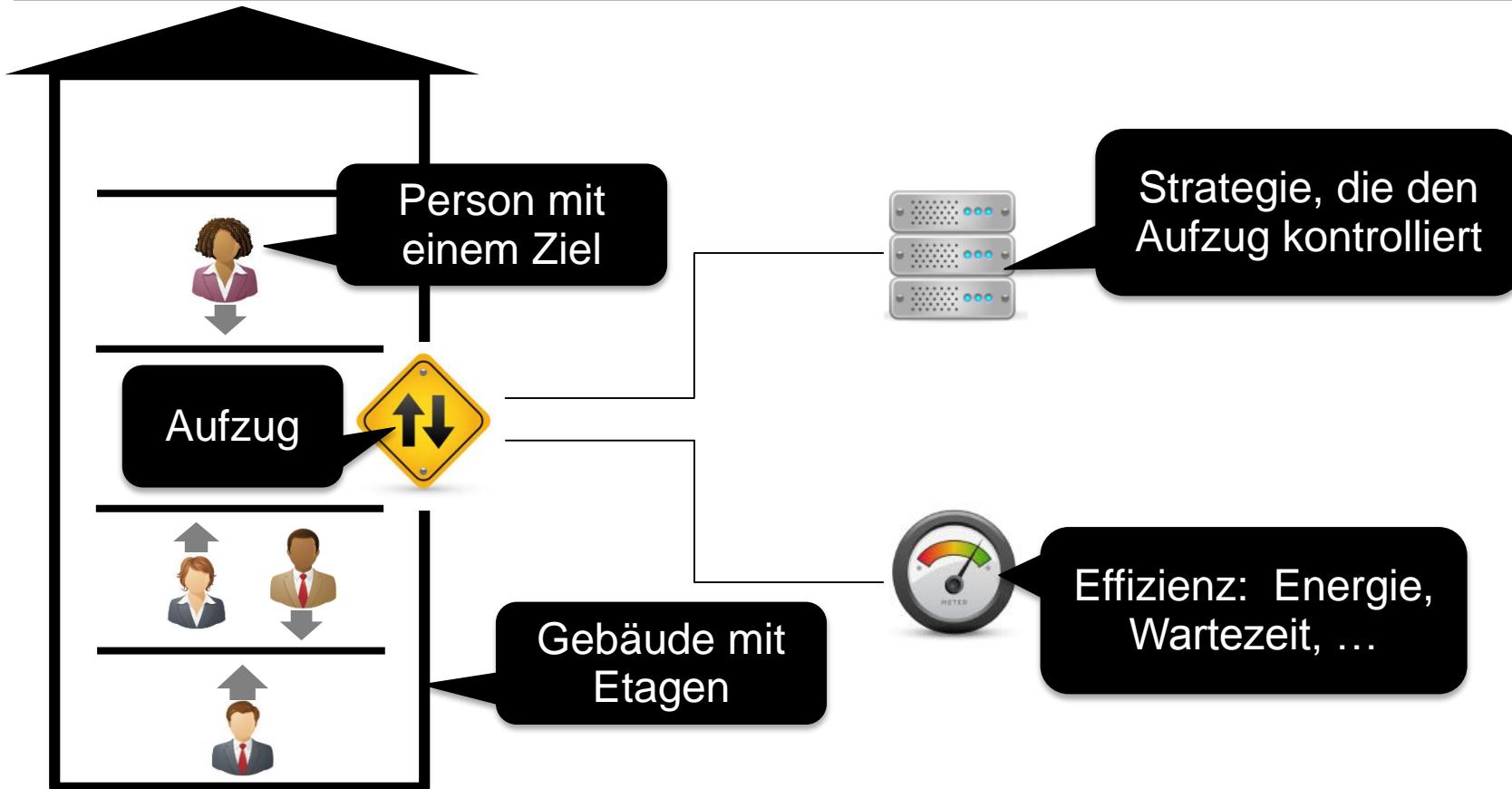


```
class Base {  
public:  
    virtual void print() {  
        std::cout << "B" << std::endl;  
    } };  
  
class Child : public Base{  
public:  
    virtual void print() override {  
        std::cout << "C" << std::endl;  
    } };  
  
void doPrint(Base b) { b.print(); }  
  
void doPrintRef(Base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

Polymorphie funktioniert in C++ nur mit Pointern und Referenzen

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

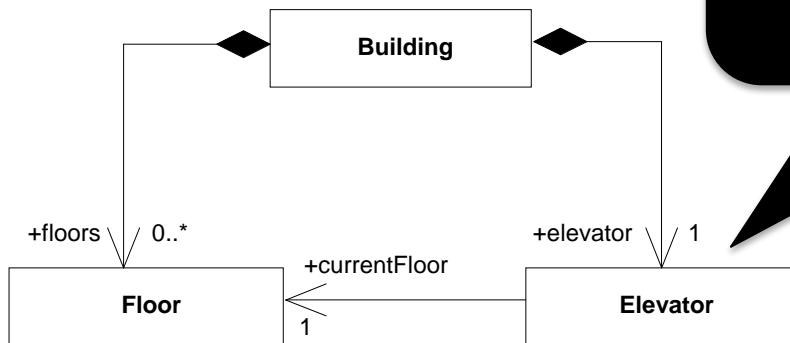
# Wozu Polymorphie?



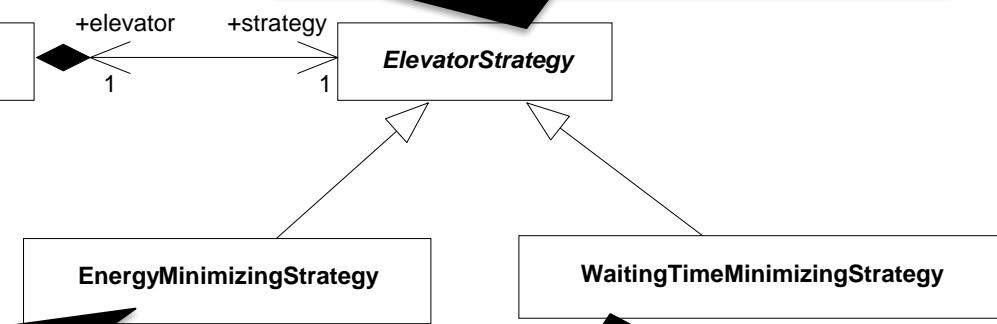
# Verschiedene Strategien als Unterklassen



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



**(Abstrakte) Oberklasse** kann nicht instatiert werden.



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

Konkrete Unterklassen

# 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 ...  
    }  
}
```

## "Dispatch" von Hand

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

Fehleranfällig, schlecht wartbar  
**(Fluch des switch-case)**

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

## 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**.

# 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?



# Ein Blick auf die Klassen ElevatorStrategy

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

```
#include <memory>           ElevatorStrategy.hpp
#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 .cpp-Datei ist dies aber kein Problem!



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
#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();
}
```

ElevatorStrategy.cpp

**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

Parameter ohne  
Namen möglich

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

Elevator.cpp

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

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

# Sichtbarkeits-Modifier bei Vererbung



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() {

    // ...
}

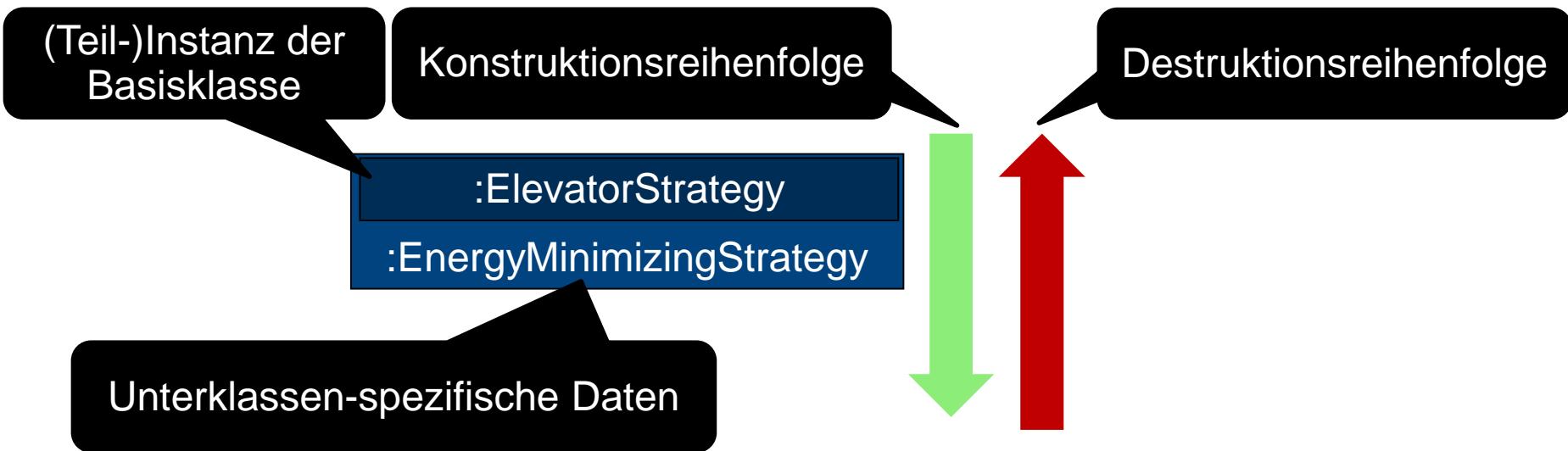
EnergyMinimizingStrategy::~EnergyMinimizingStrategy() {
    cout << "~EnergyMinimizingStrategy(): "
        << "Destroying energy minimizing strategy"
        << endl;
}

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 Destruktion bei Vererbung



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

# Probelauf unserer Simulation



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;  
};
```

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

# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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

Wo sind **virtuelle Konstruktoren** nützlich?



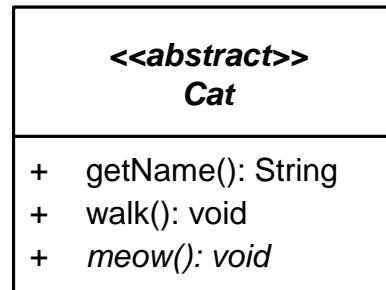
# Exkurs: Virtual Method Table

## Der Mechanismus der dynamischen Bindung



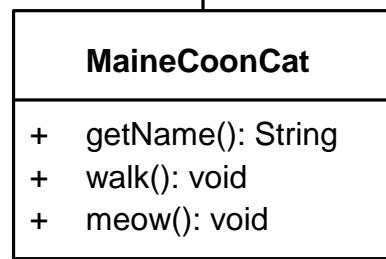
Egal, wie der Pointer auf ein Objekt deklariert ist (z.B. ElevatorStrategy\*),  
**das Objekt behält seinen Typ** (z.B. EnergyMinimizingStrategy\*).

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.

[https://en.wikipedia.org/wiki/Virtual\\_method\\_table](https://en.wikipedia.org/wiki/Virtual_method_table)

# Probelauf mit virtuellen Methoden



ElevatorStrategy(): Creating basic strategy

EnergyMinimizingStrategy(): Creating energy minimizing strategy

Floor(): Creating floor [0]

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

~Floor(): Destroying floor [0]

Elevator(): Creating elevator.

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

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

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 durch =0 nicht mehr instantiiert werden.

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

- 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. (~ Hierarchie abstrakter Klassen)

# Intermezzo



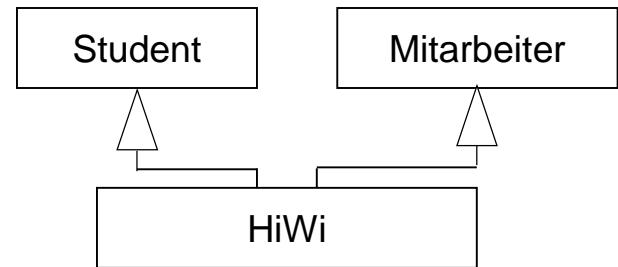
TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Wieso sind **virtuelle Methoden** "teuer"?

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

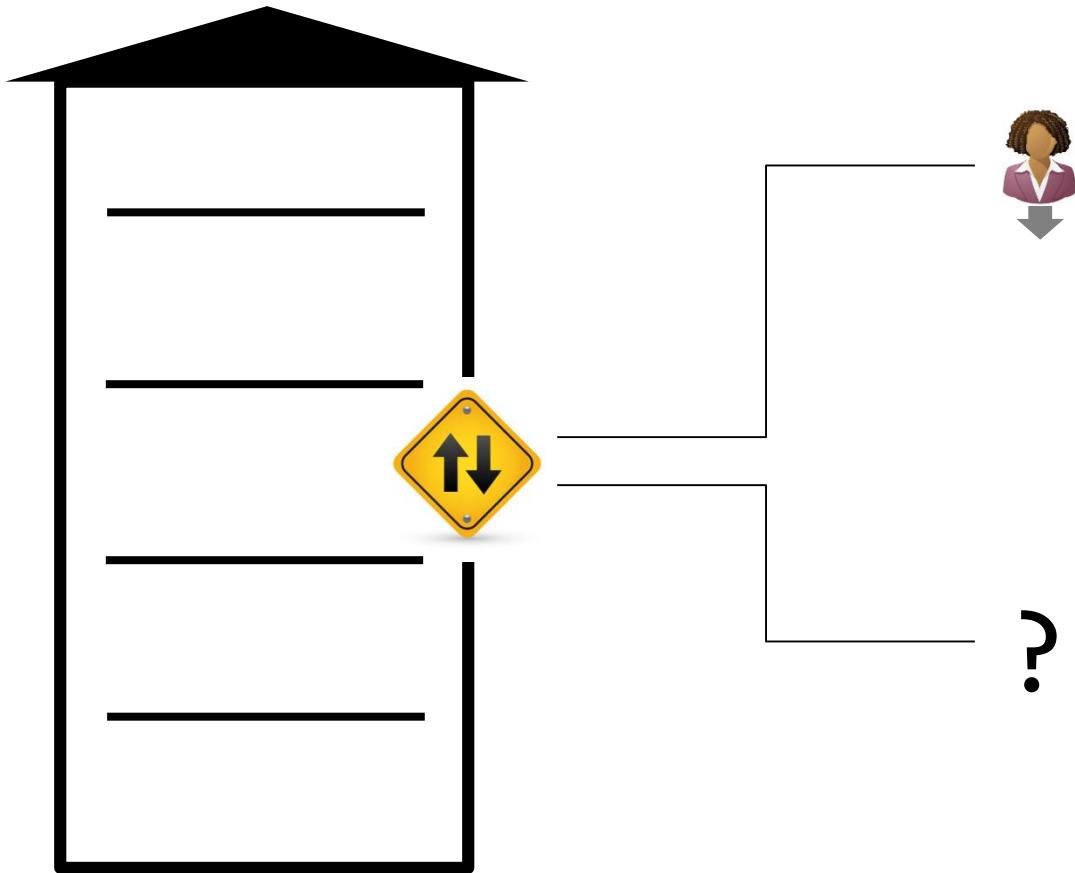
```
virtual const Floor* ElevatoryStrategy::next(const  
Elevator *elevator) const = 0;
```





# MEHRFACHVERERBUNG

# Mehrfachvererbung: Motivation



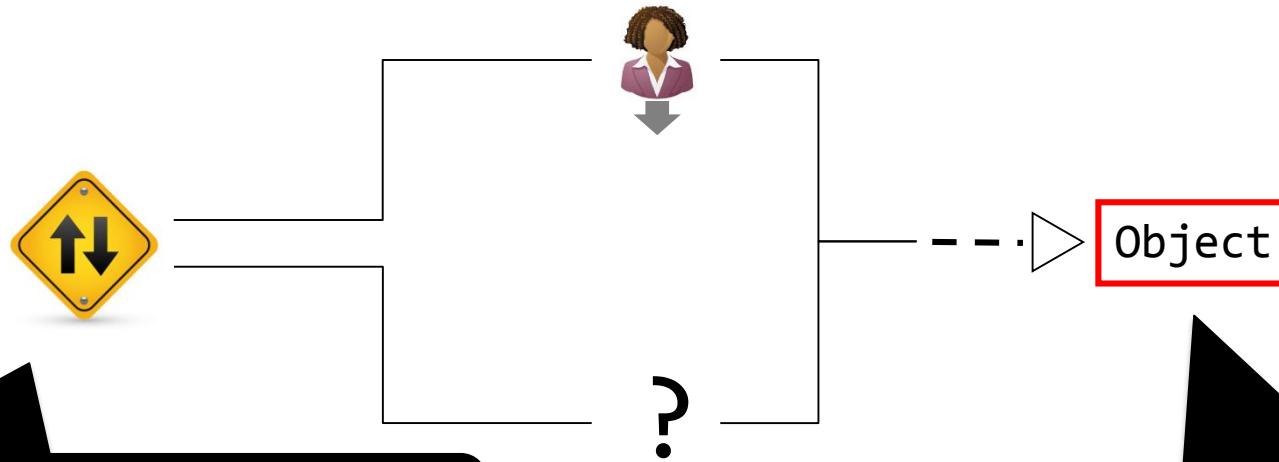
**Ziel: Aufzüge für bestimmte Zwecke**

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

# Historie: Das Containerproblem



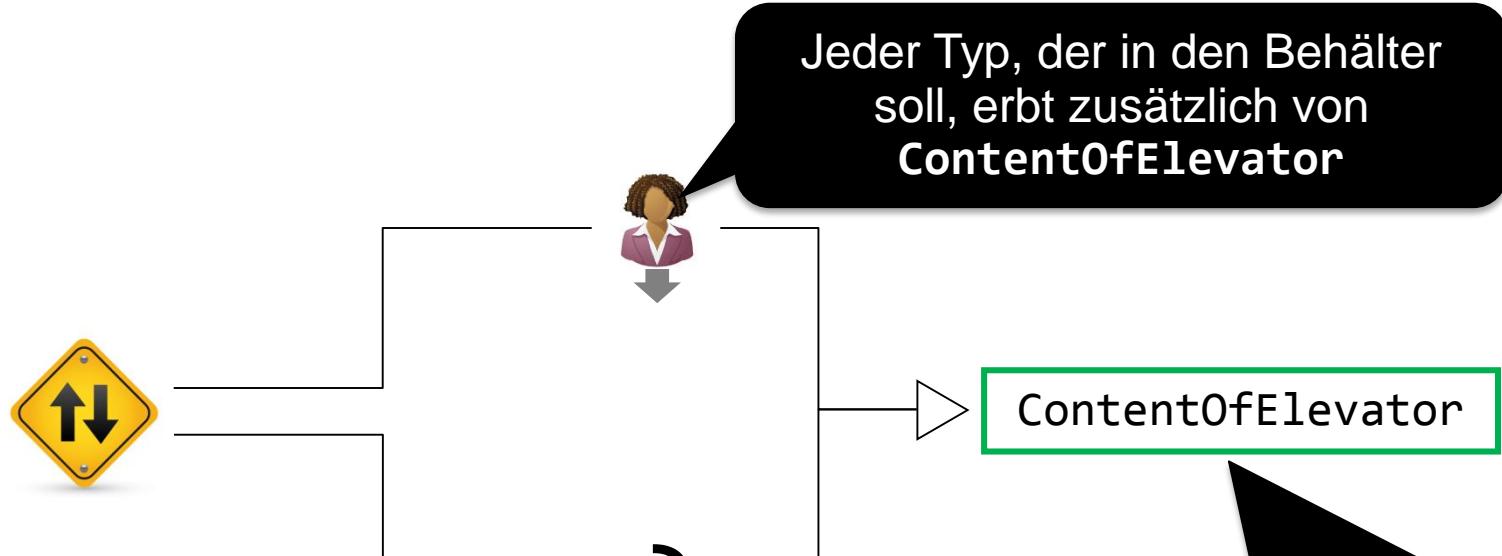
**Ursprünglich als Lösung für Containerproblem:** Wir wollen Objekte unterschiedlicher Art in den Aufzug (Container) laden.



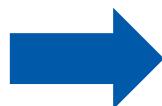
Wir können also nicht einfach **Objects** in den Aufzug laden

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

# Lösung mit Mehrfachvererbung



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



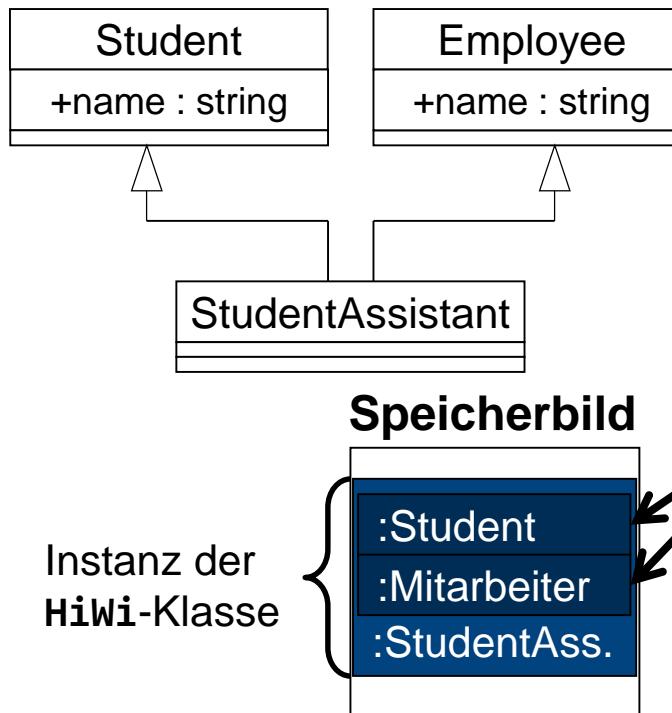
Besserer Ersatz (in  
diesem Fall):  
**Templates (Tag 4)**

# 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 Employee {public: std::string name;};

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

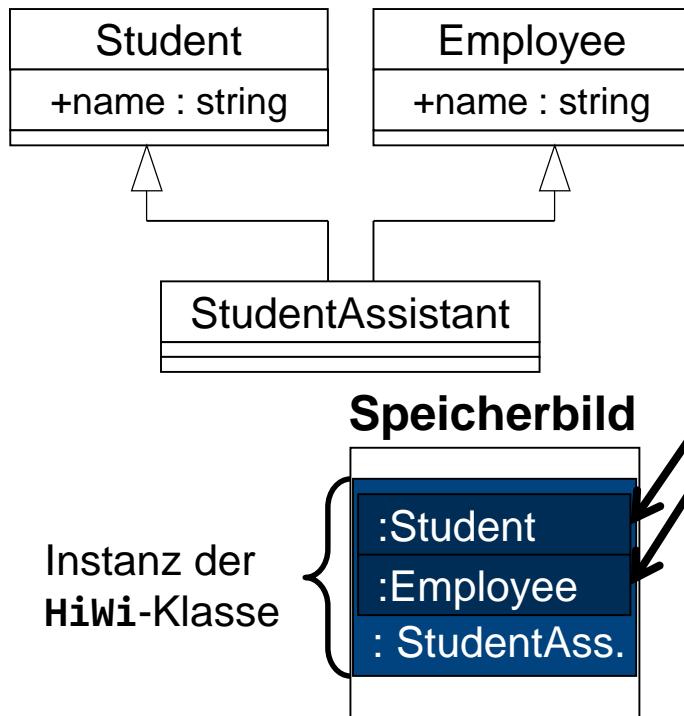
```
int main() {
    StudentAssistant *h = new StudentAssistant();
    h->name = "Christian";
    /* Error: best match for name is ambiguous */
}
```

Namenskonflikt!  
Keine eindeutige  
Zuweisung ...

# Implementierungsvererbung: Konflikte



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



```
#include <string>

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

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

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

Instanz der  
HiWi-Klasse

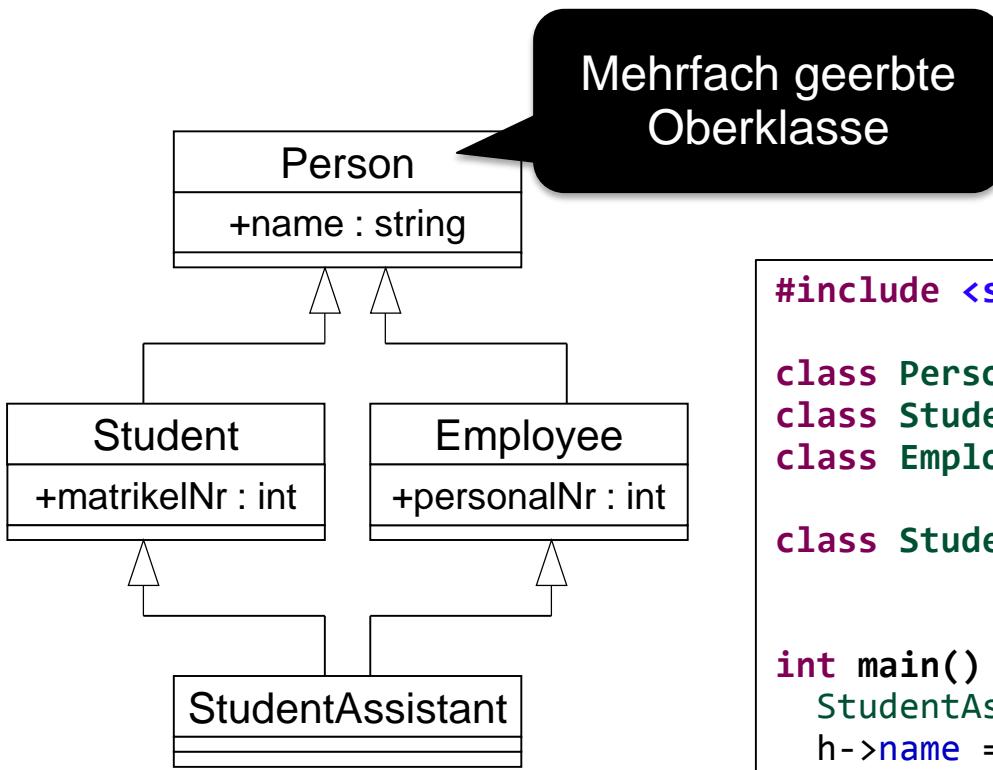
:Student  
:Employee  
: StudentAss.

Scope-Operator nötig!

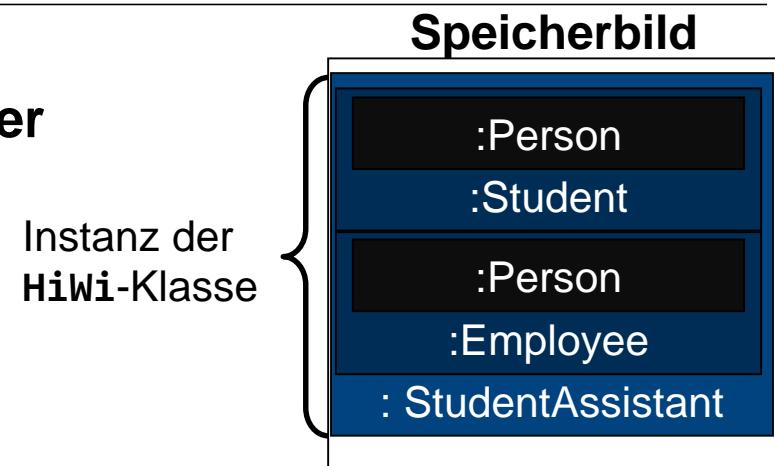
# Implementierungsvererb.: Speicherproblematik



Mehrfach geerbte Oberklassen führen  
auch zur unnötigen Bindung von Speicher



Mehrfach geerbte  
Oberklasse



```
#include <string>

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

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

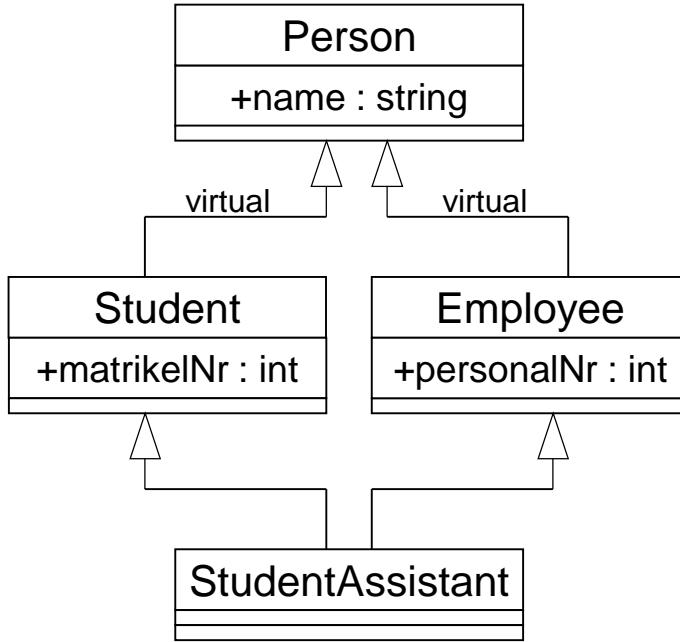
int main() {
 StudentAssistant\* h = new StudentAssistant();
 h->name = "Christian";
}

Fehler! Keine  
eindeutige  
Zuweisung ...

# Virtuelle (Mehrfach-)Vererbung (I)



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



```
#include <string>

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

class HiWi:
    public Student,
    public Employee{};

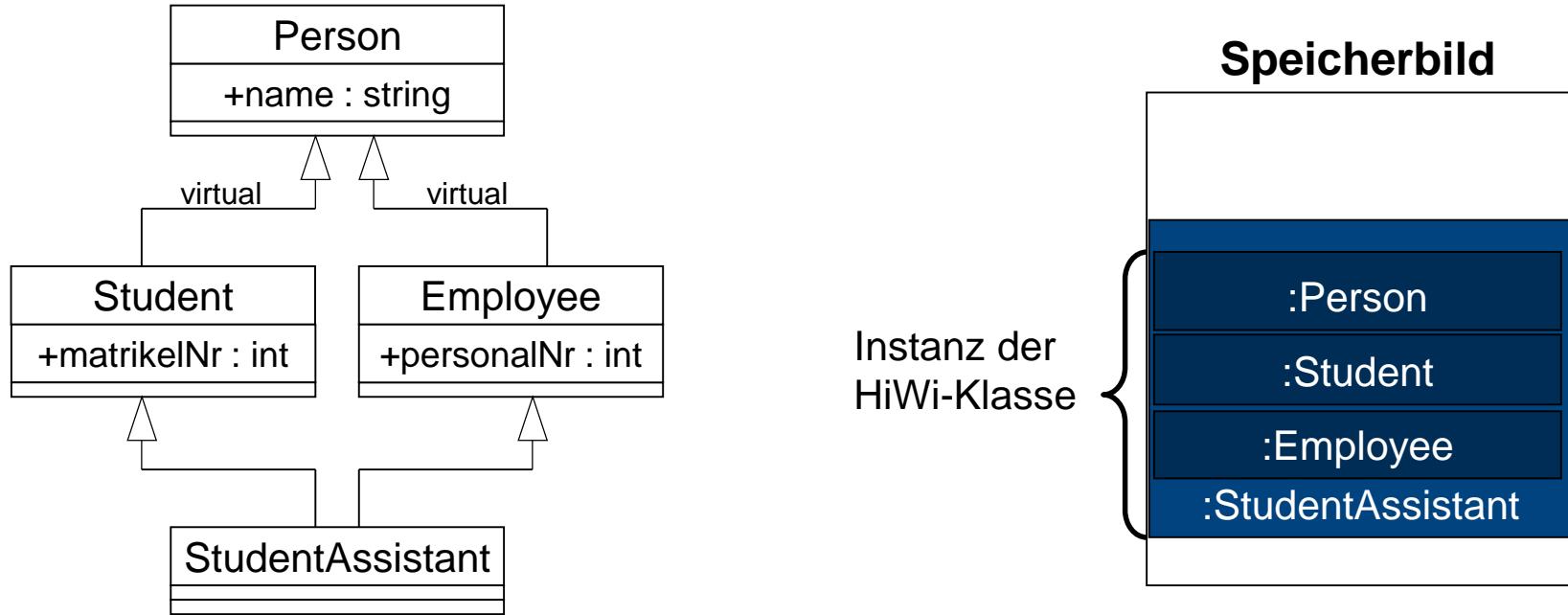
int main() {
    StudentAssistant* h = new StudentAssistant();
    h->name = "Max";
}
```

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

# Virtuelle (Mehrfach-)Vererbung (II)



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

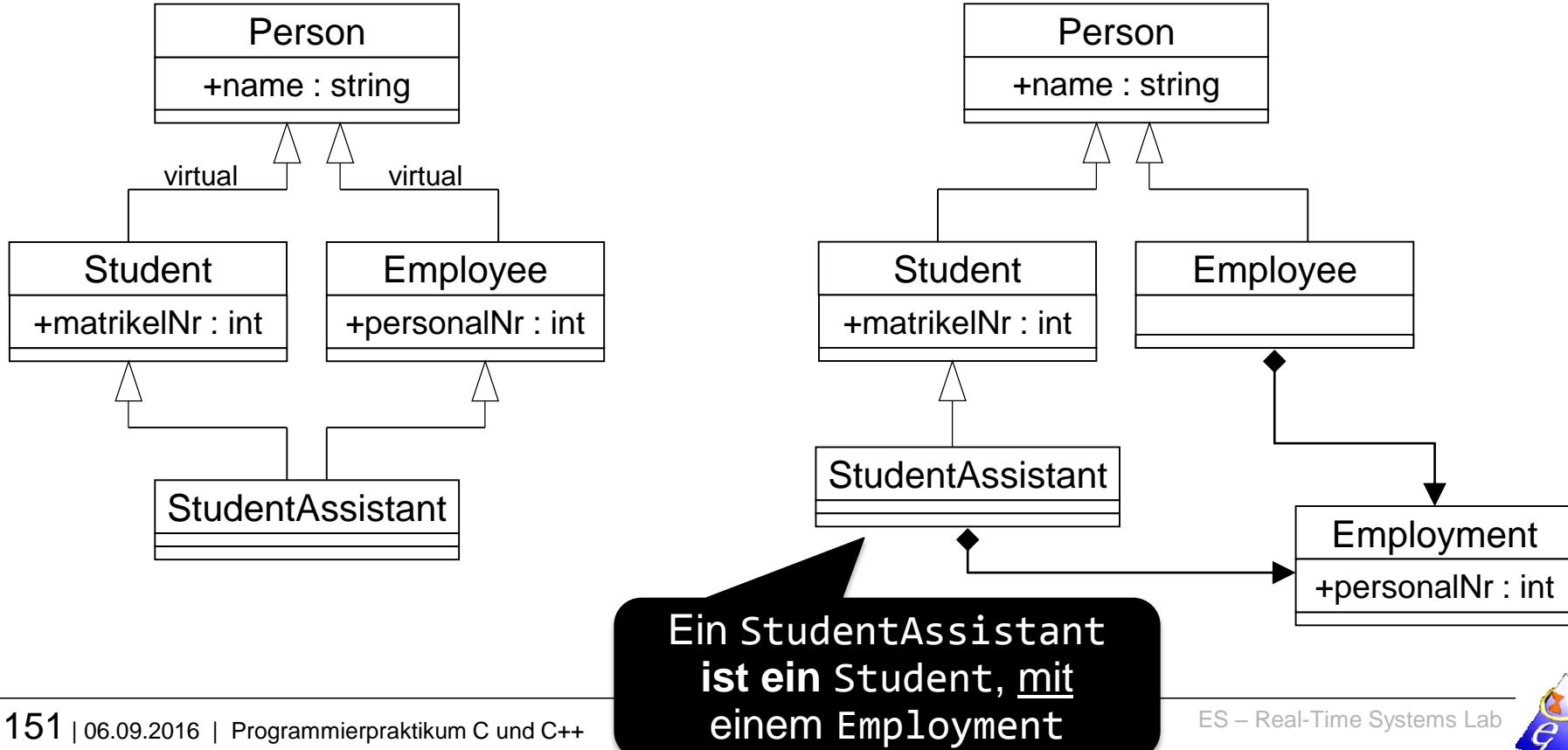


# Implementierungsvererbung: Schlechtes Design?



## Mehrfachvererbung kann auf schlechtes Design hindeuten

Gemeinsamkeiten sollen explizit extrahiert und das Design vereinfacht werden



# Schnittstellen- vs. Implementierungsvererbung



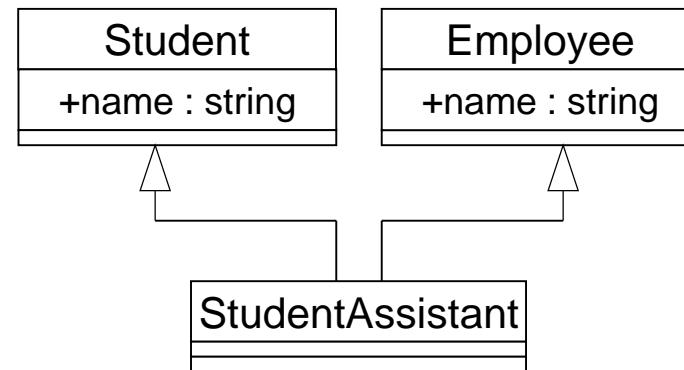
## Schnittstellenvererbung:

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

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

## Implementierungsvererbung:

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



Konzept

# Wie funktioniert Mehrfachvererbung in Java?



**Frage:** Wie wird in Java die folgende Situation gelöst?

**Antwort:** Gar nicht – darf so nicht vorkommen!

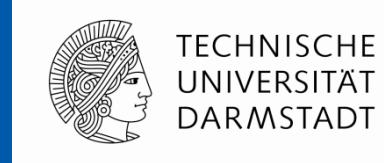
```
interface InterfaceA {      int run();      }  
interface InterfaceB {      boolean run();     }  
  
public class MyClass implements InterfaceA, InterfaceB {  
  
    @Override  
    public int run() {  
        return 0;  
    }  
}  
}
```

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](http://www.es.tu-darmstadt.de)

**Roland Kluge**

[roland.kluge@es.tu-darmstadt.de](mailto: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

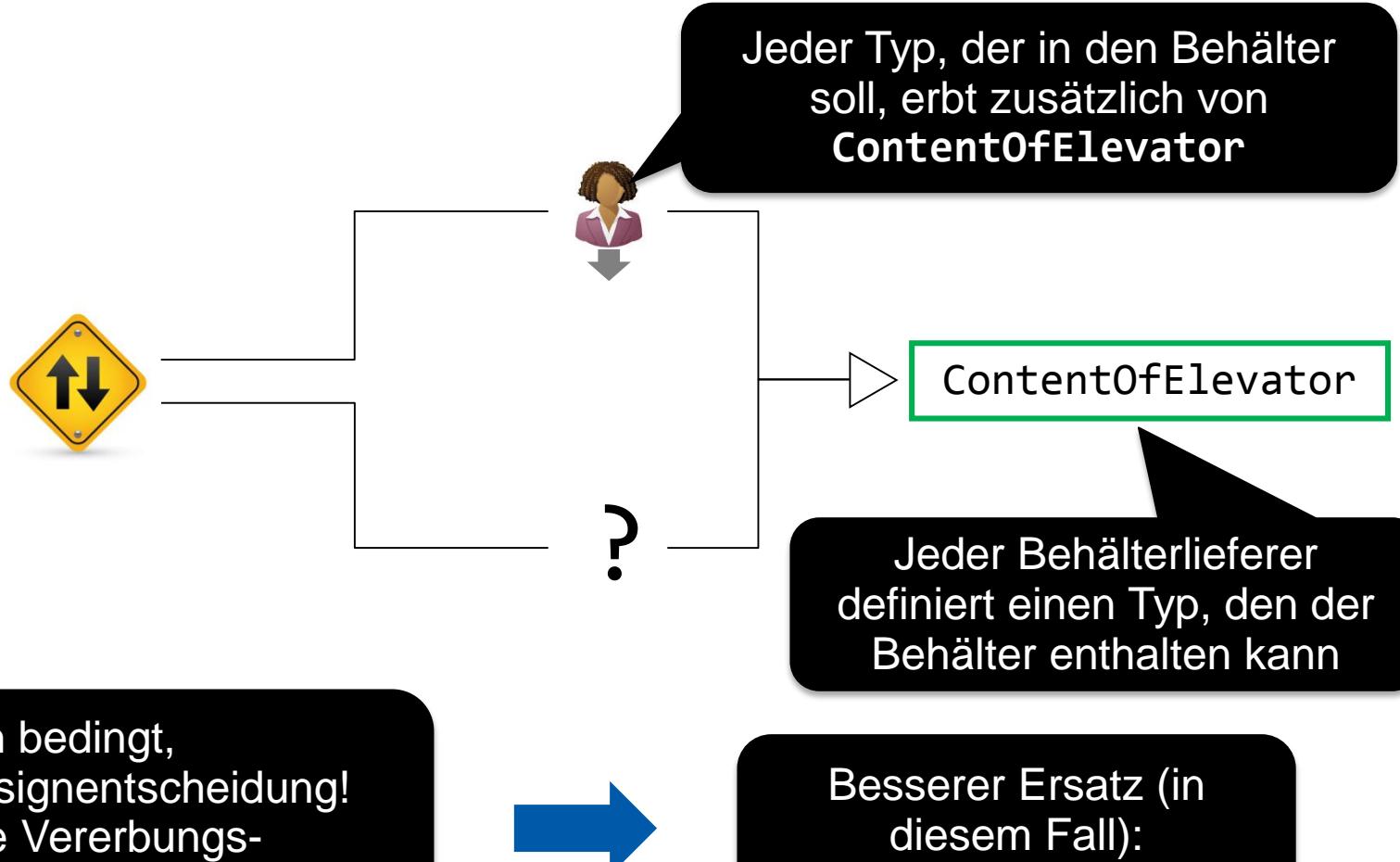
```
all: main.exe

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



# TEMPLATES

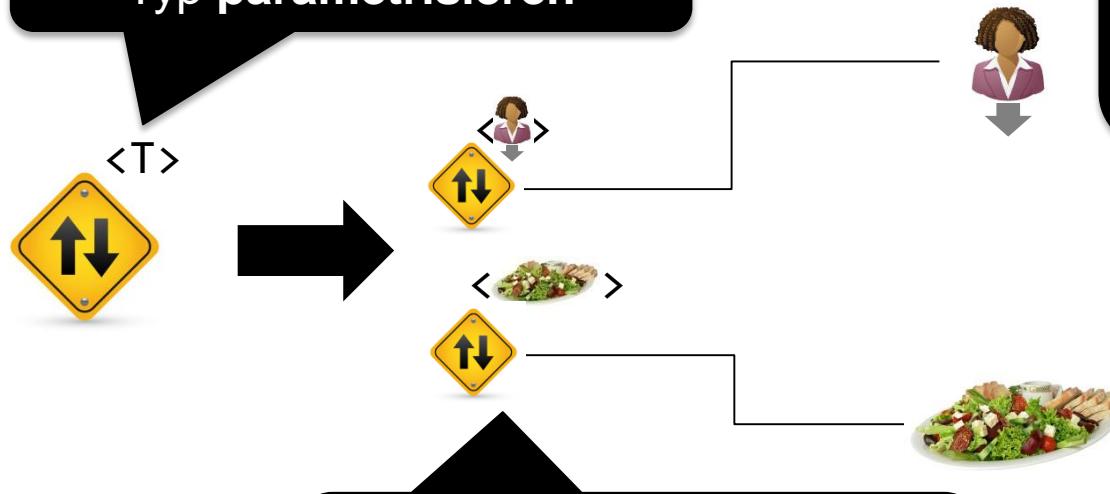
# Rückschau: Containerproblem und Mehrfachvererbung



# 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

# 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<typename 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<typename S, typename 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 Klassen-Templates)

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

Dies ist besonders für  
**generische Algorithmen** sehr nützlich

# Templates: Verwendung

Defaulttyp *Person* wird verwendet



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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

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

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

// :~>

Elevator<Dish> dumbwaiter;

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

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

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

"Primitive"  
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?



# Induzierte Schnittstelle



## Template-Code

```
template<typename S, typename 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
{
    // Nothing else needed!
};
```

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** (→ Mehrfachvererbung in Java) sinnvoll, nützlich (Design!) und zumeist unproblematisch
2. **Implementierungsvererbung** (→ Mehrfachvererbung in C++) problematischer und zu vermeiden (Komposition vorziehen, vgl. Employment-Klasse in HiWi-Beispiel)
3. **Mixins** durchaus sinnvoll - eigentlich eine Art Komposition



# FUNKTIONSZEIGER, FUNKTIONSOBJEKTE/FUNKTO REN UND METHODENZEIGER

# Funktionszeiger: Motivation (I)



```
class Timer {  
public:  
    double measureDuration(  
        unsigned long iterations,  
        void (*fp)(unsigned long)) {  
        tic();  
        fp(iterations);  
        toc();  
        return 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<typename F, typename T>
void applyToSequence(F function, T* begin, T* end){
    while (begin != end) function(*begin++);
}
```

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

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

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

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

Ermöglicht kompakte, elegante, und sehr generische Algorithmen

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

# Funktionszeiger: Beispiel II



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

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

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

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

Zeiger auf eine Funktion  
mit const string&  
Parameter

Verwendung wie ein  
normaler  
Funktionsaufruf



# 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");
```

# Funkoren aka. Funktionsobjekte



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

**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

# Exkurs: Automatische Typableitung



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

# Exkurs: Lambdas (C++11)



- **Lambda-Ausdruck** = anonyme Funktion (ohne zugewiesenen Namen)
- **C++11:**
  - Weiterer Mechanismus, um "Verhalten als Parameter zu übergeben"
  - In Kombination mit auto extrem mächtig und zugleich kompakt!
  - Beispiel:

```
auto prefix = "Info: ";
auto print = [=] (const std::string &msg)
    {std::cout << prefix << msg << std::endl;}
print("Hello World!"); // Output: Info: Hello World!
```
  - Mittels [ ] kann die Variable prefix aus dem Kontext von print "**eingefangen**" werden ([=] "by value", [&] "by reference")
- **In Java seit 1.8**
  - Beispiel: `Arrays.asList(1,2,3).stream()
 .map(x -> x*x)
 .filter(x -> x < 7).collect(Collectors.toString());`

z.B. <http://www.cprogramming.com/c++11/c++11-lambda-closures.html>  
viele Beispiele: [https://en.wikipedia.org/wiki/Anonymous\\_function](https://en.wikipedia.org/wiki/Anonymous_function)



# 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<typename S>  
    void print(const S& s) { /* ... */}  
    void validateAges(int a) { /* ... */}  
};
```

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

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

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

... entsprechend ändert sich der Aufruf.

# 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?



# 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 TEMPLATE LIBRARY (STL) VON C++**

# Generische STL-Algorithmen: std::copy



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

Parameters:  
`first, last`

## STL-weite Konvention zur Nutzung von Iteratoren

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

`result`

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

**Return Value:**

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

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

# 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?



# Generische STL-Algorithmen: std::copy



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

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

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

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

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

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

STL-Behälter bieten  
InputIteratoren an

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

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

# Generische STL-Algorithmen:

## std::remove\_copy\_if



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

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

### Parameters:

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

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

The function shall not modify its argument.

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

### Return Value:

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

[http://www.cplusplus.com/reference/algorithm/remove\\_copy\\_if/](http://www.cplusplus.com/reference/algorithm/remove_copy_if/)

# Generische STL-Algorithmen: std::remove\_copy\_if



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

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

Funktion even entscheidet  
was ausgelassen wird

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

    remove_copy_if(result.begin(), result.end(),
                  ostream_iterator<int>(cout, ", "),
                  even); // 1, 3, 5
}
```

Funktionszeiger oder  
Funktionsobjekt übergeben

[http://www.cplusplus.com/reference/algorithm/remove\\_copy\\_if/](http://www.cplusplus.com/reference/algorithm/remove_copy_if/)

# Generische Behälter: std::priority\_queue



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

Typ vom Inhalt der Warteschlange

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

Binäres Prädikat (`less` wird als Default verwendet)

Damit Compiler weiß, dass `value_type` ein Typ ist

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

[http://www.cplusplus.com/reference/queue/priority\\_queue/](http://www.cplusplus.com/reference/queue/priority_queue/)

# Generische Behälter: std::priority\_queue



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

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

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

Einfache Hilfsfunktion  
für die Ausgabe

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

[http://www.cplusplus.com/reference/queue/priority\\_queue/](http://www.cplusplus.com/reference/queue/priority_queue/)

# Intermezzo



## Schleifen-basierte Lösung vs. std::remove\_copy\_if

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

vs.

```
template<class T, typename InputIterator, typename
OutputIterator>
void myCopyRemoveIf(InputIterator first, InputIterator last,
OutputIterator result) {
    for (T iter = first; iter != last; ++iter)
    {
        if (!P(*iter)) {
            (*result) = *iter
            ++result;
        }
    }
}
```



**Was ist "schöner"? Was ist fehleranfälliger? Was ist kompakter?**

# 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 mit **hunderten von Dateien/Klassen** und **noch mehr 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

# "Make is an expert system." (nach [1])

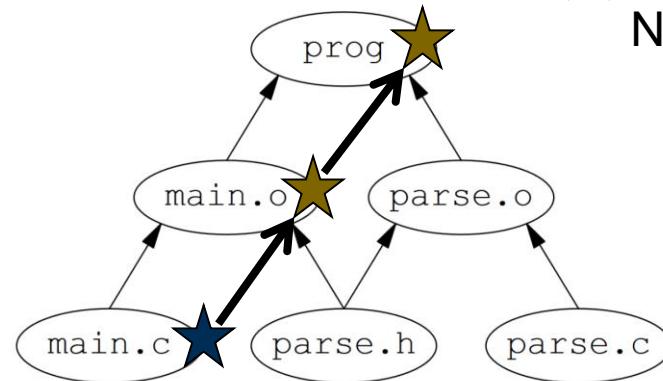
- **Eingabe:** Regelmenge (fix) + Zustand des Workspaces (variabel)
- **Ausgabe:** Notwendige Buildschritte

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

## Makefile (Regelmenge)

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

## Workspace



- ★ Veränderung
- ★★ Notwendige Neuberechnung

## Directed Acyclic Graph

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

# 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
```

```
-include $(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

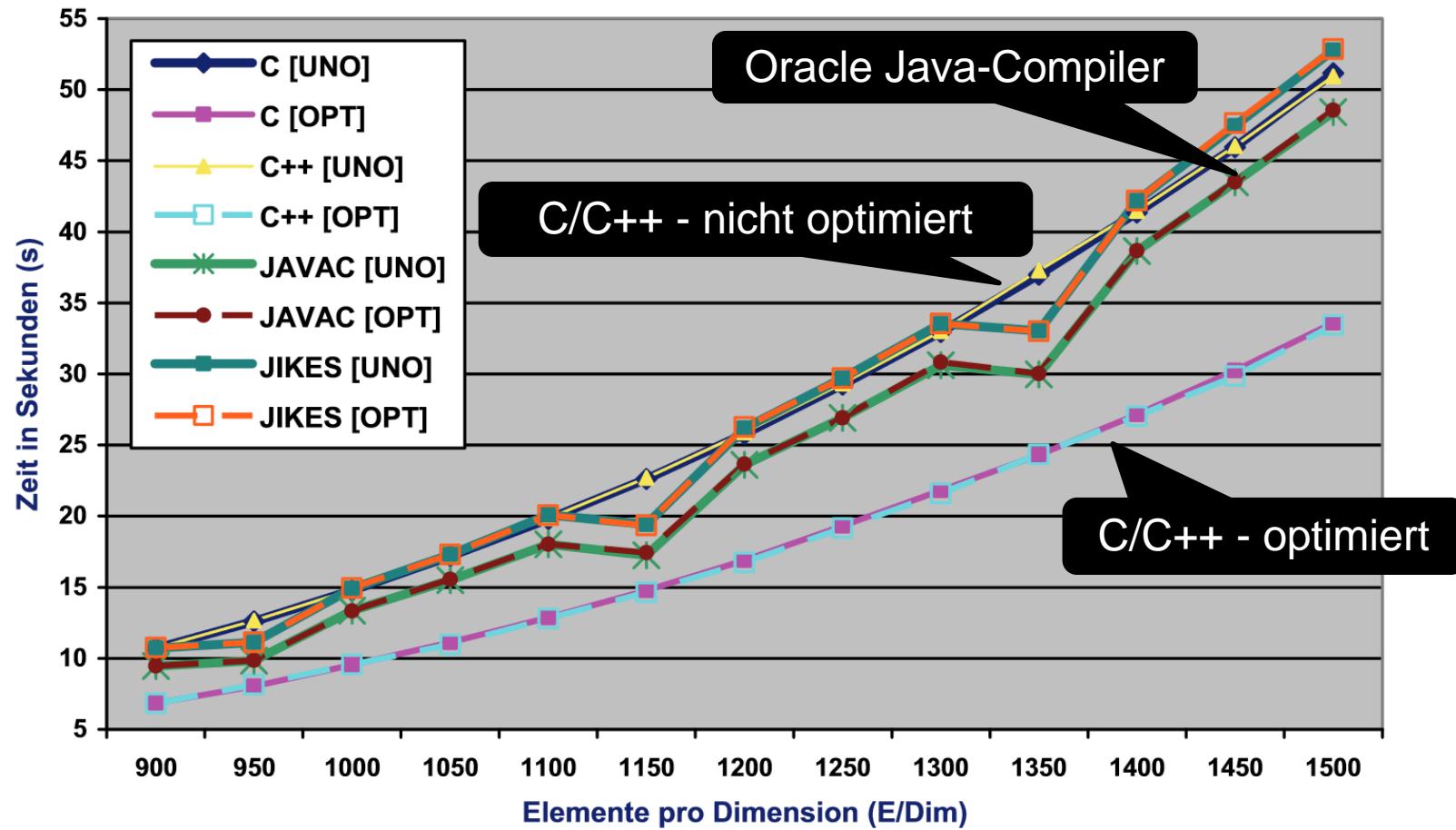


## Java vs. C++: Stärken und Schwächen?

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

# 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!

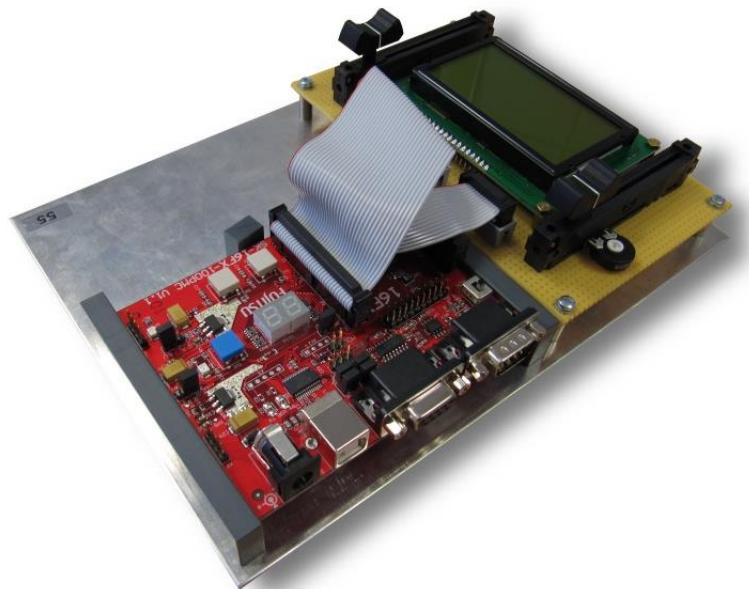
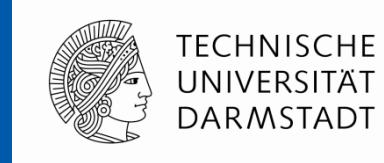


TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Nützliche Kommentare  
finden sich  
auch in den PowerPoint-Notizen!

# Programmierpraktikum C und C++

C für Microcontroller – Einführung



**Roland Kluge**

[roland.kluge@es.tu-darmstadt.de](mailto:roland.kluge@es.tu-darmstadt.de)

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](http://www.es.tu-darmstadt.de)

# Entwicklungsboard

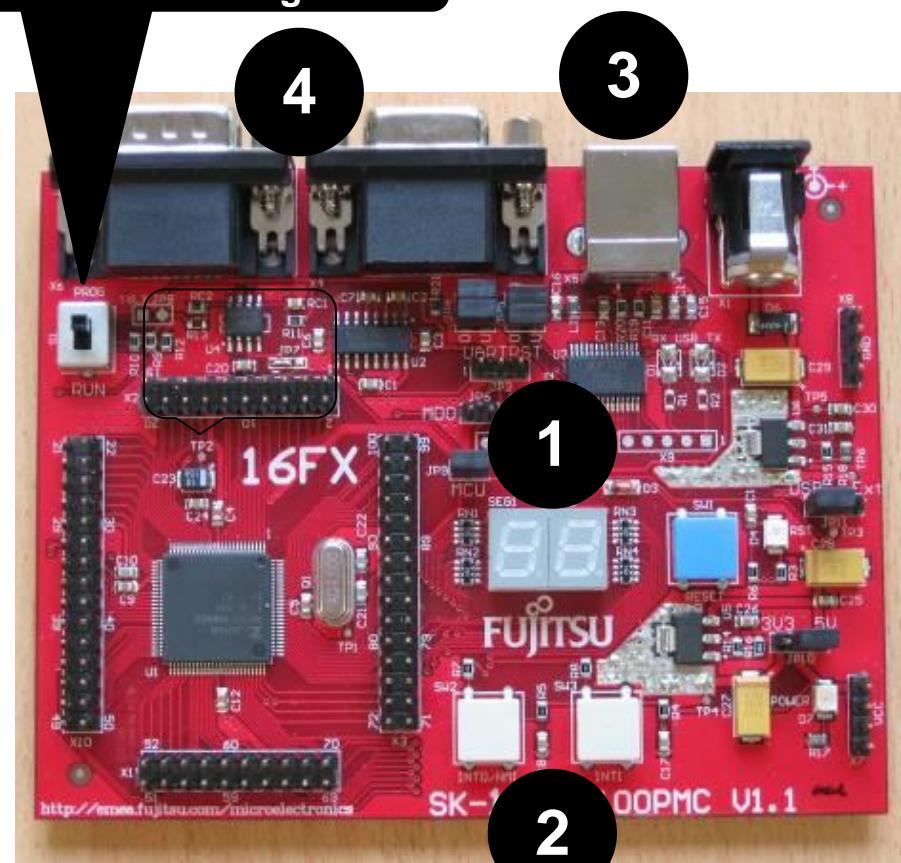


## MB96F348HSB Mikrocontroller

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

4

PROG für Programmierung  
RUN für Ausführung



## Starterkit SK-16FX-EUROscope

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

1

2

3

# Erweiterungen gegenüber der Standardausführung

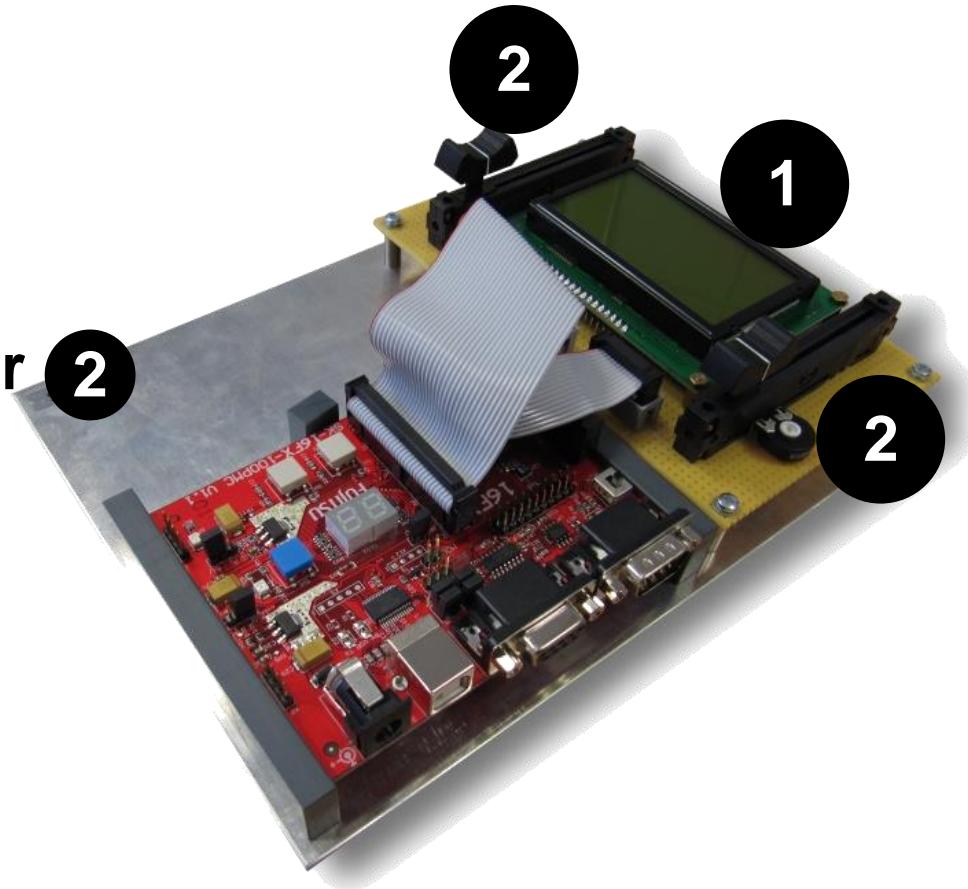


TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## 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)**

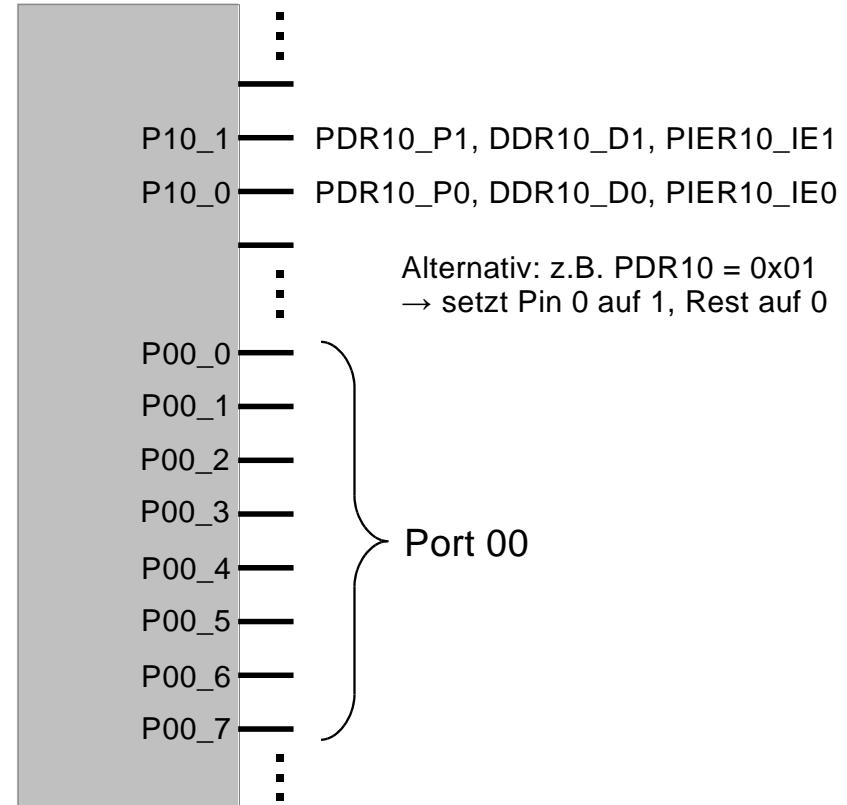
- als *Eingang*: Abfrage des Zustandes
- als *Ausgang*: Setzen des Pegels
- z.B. PDR07\_P0

- **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



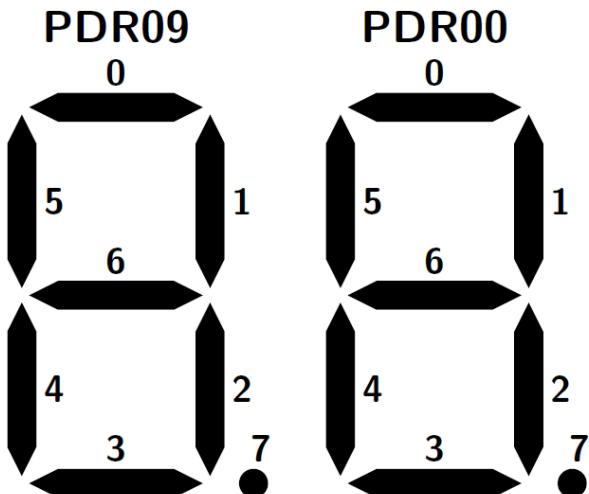
```
/* Beispiel: Pins als Eingang */
char status;
DDR07_D0 = 0;           // Pin 0 von Port 07 als Input
PIER07_IE0 = 1;         // Pin 0 von Port 07 als Eingang
aktiv

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
```



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# TECHNISCHE ANMERKUNGEN

# Projektvorlagen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Die **Projektvorlagen** (~/CPPP/Repos/ tud-cpp-exercises/projects/day5) enthalten Code, der euch beim Starten hilft.
- **Tipp:** Nicht in .../Repo arbeiten, da git sonst das Pullen verhindert.

# Anschluss des Boards an die Virtuelle Maschine

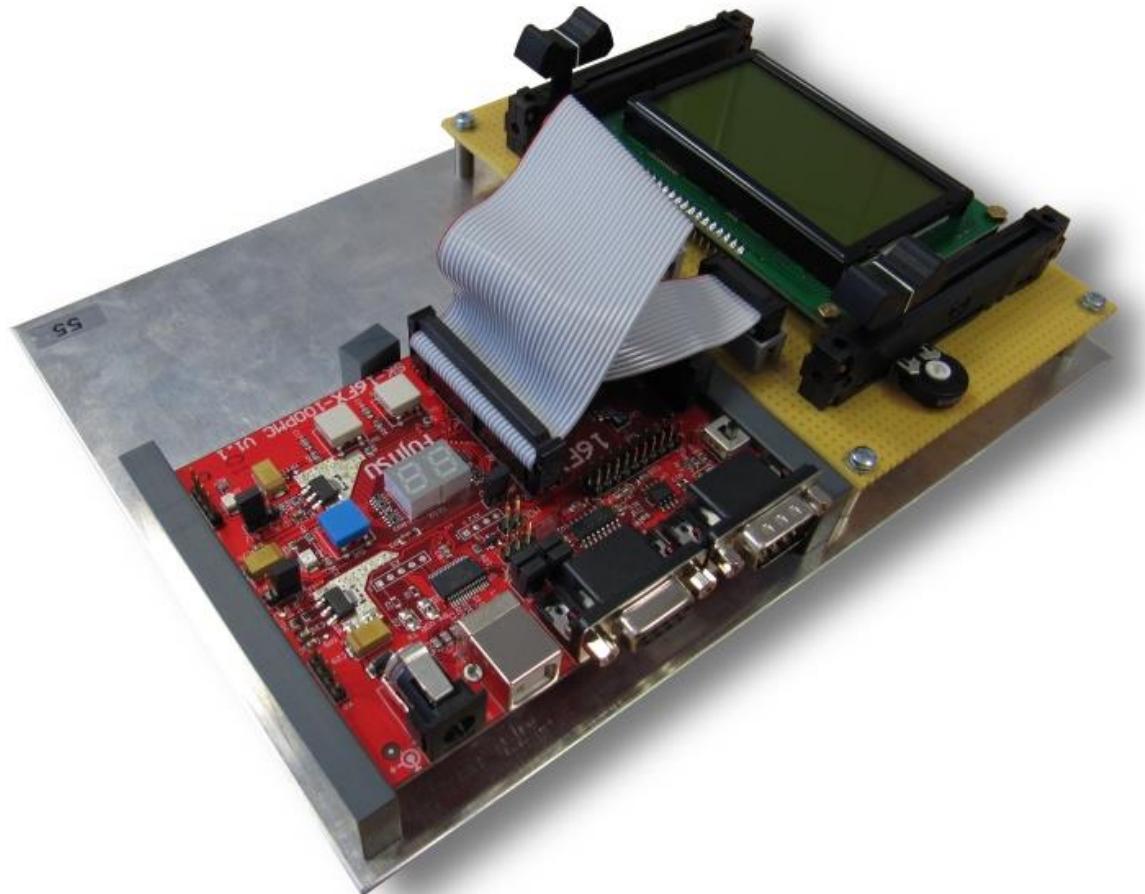


- Im **Host-System** wird eine USB-zu-Seriell-Schnittstelle erstellt
  - Windows: Im Geräte-Manager prüfen, welcher Ports erscheint, wenn das Board angesteckt wird (bspw. COM8)
  - Linux: Nach /dev/ttyUSB0 (o.ä.) Ausschau halten.
- In der **Konfiguration der VM** ("Ändern...", vor dem Start!) wird COM8 des Hosts auf den ersten COM-Port des Guest gelegt ("Host-Schnittstelle")
- In der **VM** ist dieser serielle Port als /dev/ttys0 verfügbar
  - Die Makefiles sind so aufgebaut, dass dann alles automatisch ablaufen sollte.

# Viel Spaß!



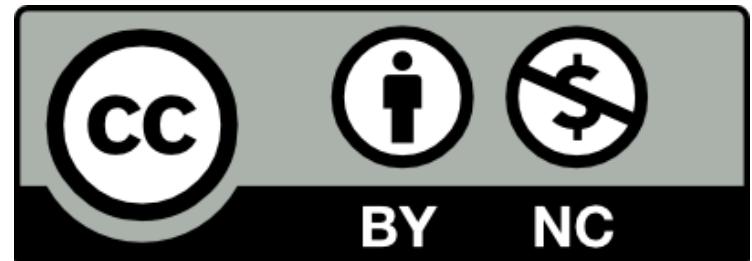
TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



Quelle: Real-Time Systems Lab

Dieser Foliensatz ist lizenziert unter

- Creative Commons "Attribution-NonCommercial 4.0 International"
- siehe <https://creativecommons.org/licenses/by-nc/4.0/legalcode>



# Bildnachweis



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

---

Titelbild "Organisatorisches" (Papierstapel): Jonathan Joseph Bondhus (Wiki Commons,  
[https://commons.wikimedia.org/wiki/Paper#/media/File:Stack\\_of\\_Copy\\_Paper.jpg](https://commons.wikimedia.org/wiki/Paper#/media/File:Stack_of_Copy_Paper.jpg), CC BY-SA 3.0)

Lächelndes Fragezeichen: katieyunholmes: smiley face clip art animated (<http://cliparts.co/clipart/2613703>, "attribution")



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# FOLIEN-BRUTKASTEN ☺

# Was passiert ohne Include Guards? Lösung.



## Vor dem Präprozessor

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

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

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

## Nach dem Präprozessor

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

# 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**.

# Exkurs: Implizite Typkonvertierung unterbinden mit `explicit`



```
#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"));
}
```

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

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

# Intermezzo



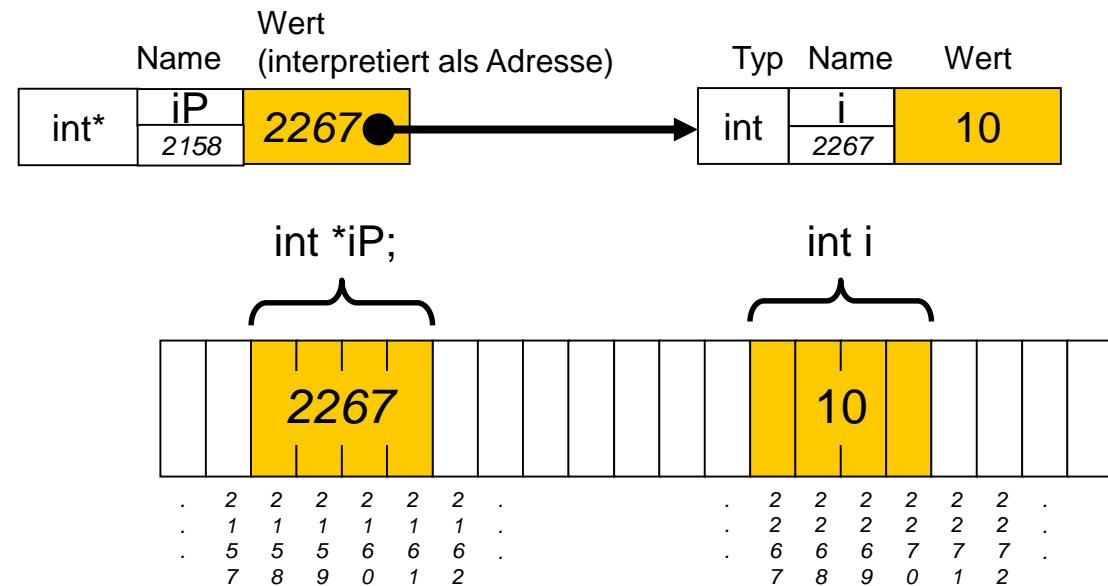
TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Welche **Speicherbereiche** gibt es in Java?

Wieso muss man sich in Java **so wenig Gedanken** um die Speicherverwaltung machen?



# Alternativer Blick auf die Bedeutung von const an verschiedenen Positionen



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

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

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

# Weak SmartPointer: Lösung

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

1. Floor[0] wird zerstört
2. 0 Smart Pointer und 1 Weak Pointer auf Eve
3. Eve wird zerstört
4. 0 Smart/Weak Pointer auf Bob
5. Bob wird zerstört

