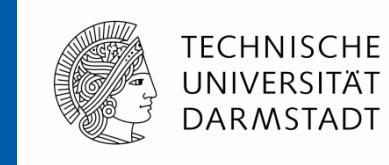


# Programmierpraktikum C und C++

## 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)

# TODO – Neue Inhalte



- **Iterierungskonzepte:**
  - std::for\_each
  - range-basierte For-Schleife
- **Initialisierungslisten**
- **auto**
- **Lambdas**
  - callable entity
  - Funktionale Programmierung: Was heißt das?
- **Delegating constructor**
- **default/delete**
  - X() = default;
  - operator=() = delete;



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

## Das wird nicht behandelt

- programmiertechnische Grundlagen (Schleifen, Rekursion, ...)
- grundlegende Datenstrukturen und Algorithmen
- Grundlagen der Objektorientierung

## Basisvoraussetzung:

Allgemeine Programmiererfahrung und Kenntnisse in Java werden vorausgesetzt!

# Organisatorisches



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Jeden Tag

09:00 – 11:30: Frontalunterricht im Hörsaal

13:00 – 16:00: praktische Übungen im Pool

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

## Anwesenheitspflicht

Ausnahmen durch E-Mail genehmigen lassen (Klausur, Krankheit)

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

## Ansprechpartner

Roland Kluge ([roland.kluge@es.tu...](mailto:roland.kluge@es.tu...)),  
Eugen Lutz, Matthias Gazzari



## Termin

Datum: Dienstag, 13.10.2014  
Uhrzeit: 16:15 – 18:15 (Bearbeitungszeit: 90 Minuten)  
Raum: S1|01 A03 (+ evtl. A04)

## Inhalt

Tag 1 – Tag 4: C++-Programmierung mit Eclipse CDT  
Tag 5 – Tag 6: C-Programmierung für Microcontroller

## Vorbereitung

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

NICHT klausurrelevant

## Zur Teilnahme erforderlich

1. amtlicher Lichtbildausweis
2. Klausuranmeldung (TUCaN!)

# Vorlesungs- und Übungsbetrieb



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Übung (nachmittags) im Raum 67

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

(User: cppp, PW: >Cppp2015< )

## Material

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

## Eigenes Projekt erstellen mit Git:

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

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

## Fachliche Fragen bitte IMMER im Moodle:

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

# Literaturvorschläge – Bücher



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

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

*Scott Meyers: Effective C++*

*Scott Meyers: More Effective C++*

*Helmut Schellong: Moderne C Programmierung [Springer]*

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

*Jürgen Wolf: Grundkurs C [Galileo]*

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

*Bjarne Stroustrup: Einführung in die Progr*

Kompakt und günstig

# Literaturvorschläge – Skripte



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Grundkurs C/C++ @ TU München

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

Sehr umfangreiches Material

## Programmieren 1 @ FH Regensburg

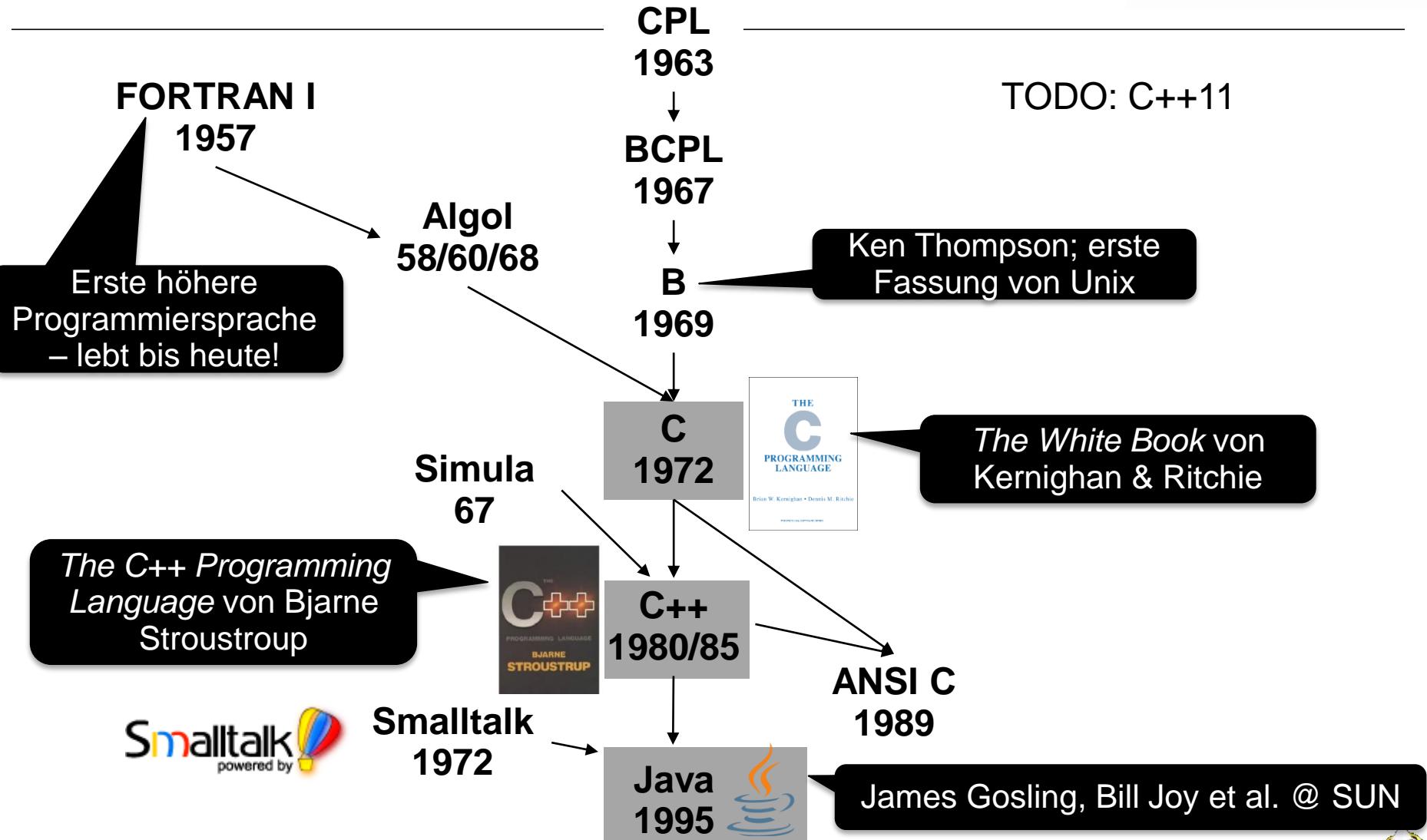
<http://fbim.fh-regensburg.de/~sce39014/pg1/pg1-skript.pdf>

## Heinz Tschabitscher, Einführung in C++

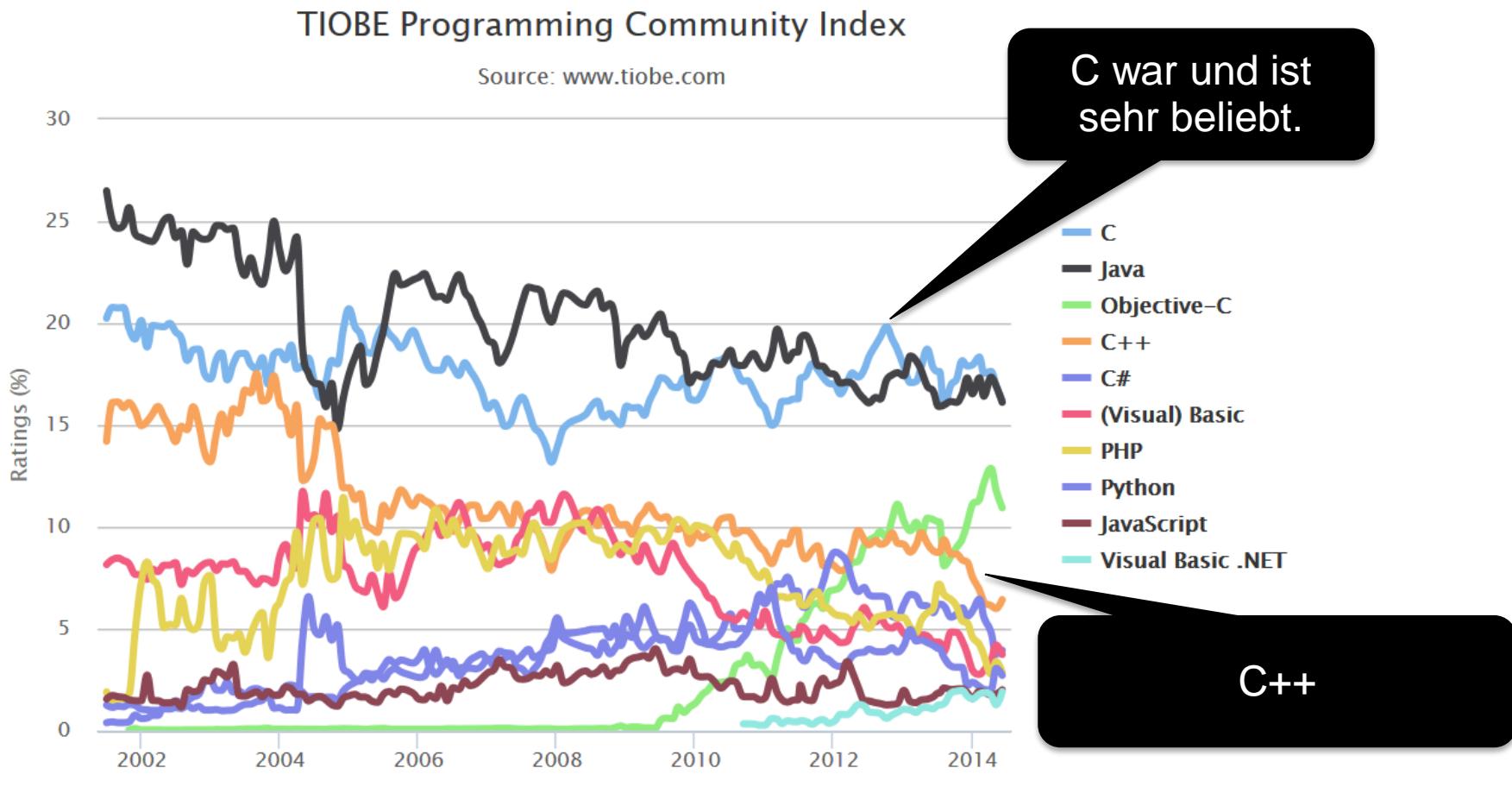
[http://ladedu.com/cpp/zum\\_mitnehmen/cpp\\_einf.pdf](http://ladedu.com/cpp/zum_mitnehmen/cpp_einf.pdf)

Grundlagen (Schleifen, etc.)

# C, C++ und Java

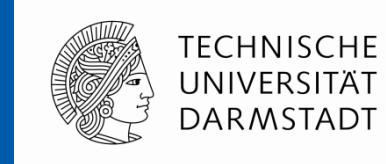


# Wie wichtig sind C/C++?



# Programmierpraktikum C und C++

## Einführung



ES Real-Time Systems Lab

Prof. Dr. rer. nat. Andy Schürr

Dept. of Electrical Engineering and Information Technology

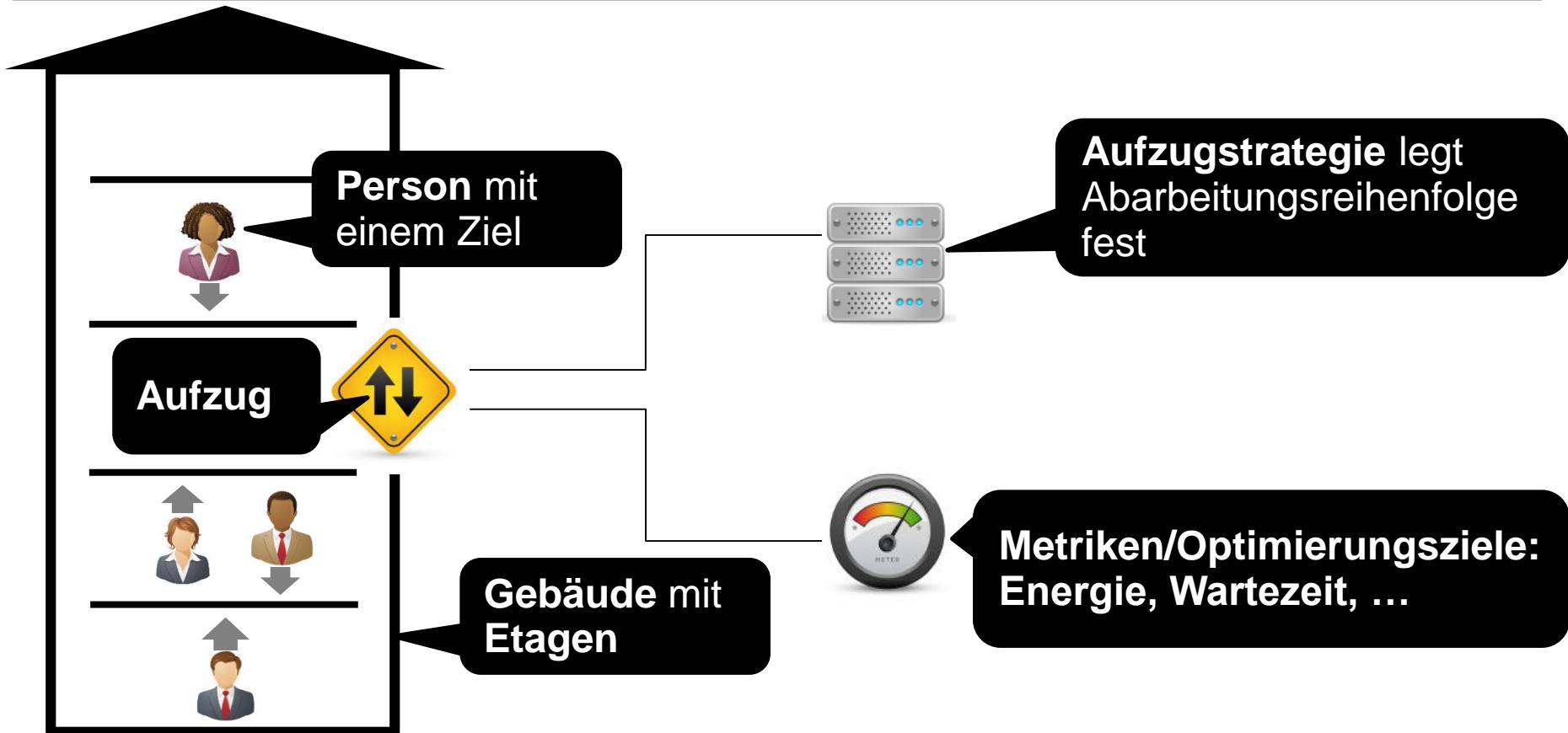
Dept. of Computer Science (adjunct Professor)

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

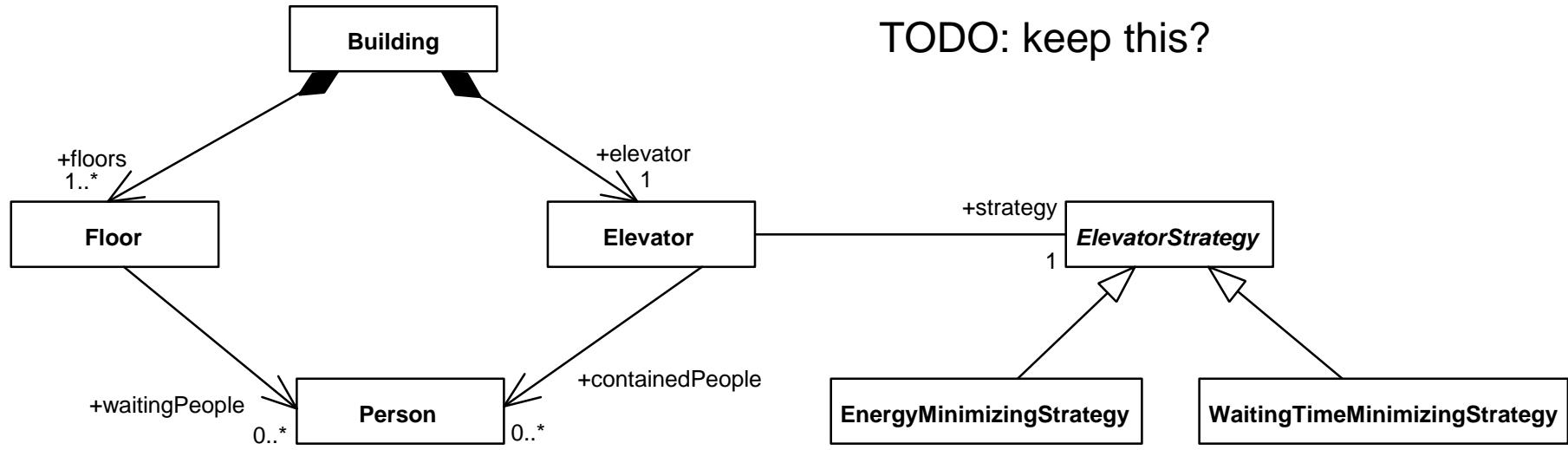
**Roland Kluge**

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

# Laufendes Beispiel: Implementierung einer Aufzugsimulation



# Statische Struktur des Systems (Klassendiagramm / Metamodell)





# Vergleich zwischen Java und C++

# PROJEKTSTRUKTUR

# Projektstruktur



C++



Java



Aufteilung in Pakete  
entspricht  
Verzeichnisstruktur

Jede Datei enthält  
(meistens nur) eine  
„public“ Klasse

Methoden sind immer in  
Klassen enthalten

Building.java

# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Ist es sinnvoll, zu verlangen, dass jede „Funktion“ in einer Klasse sein MUSS?

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

Darf man in Java mehrere Klassen in einer Datei implementieren?



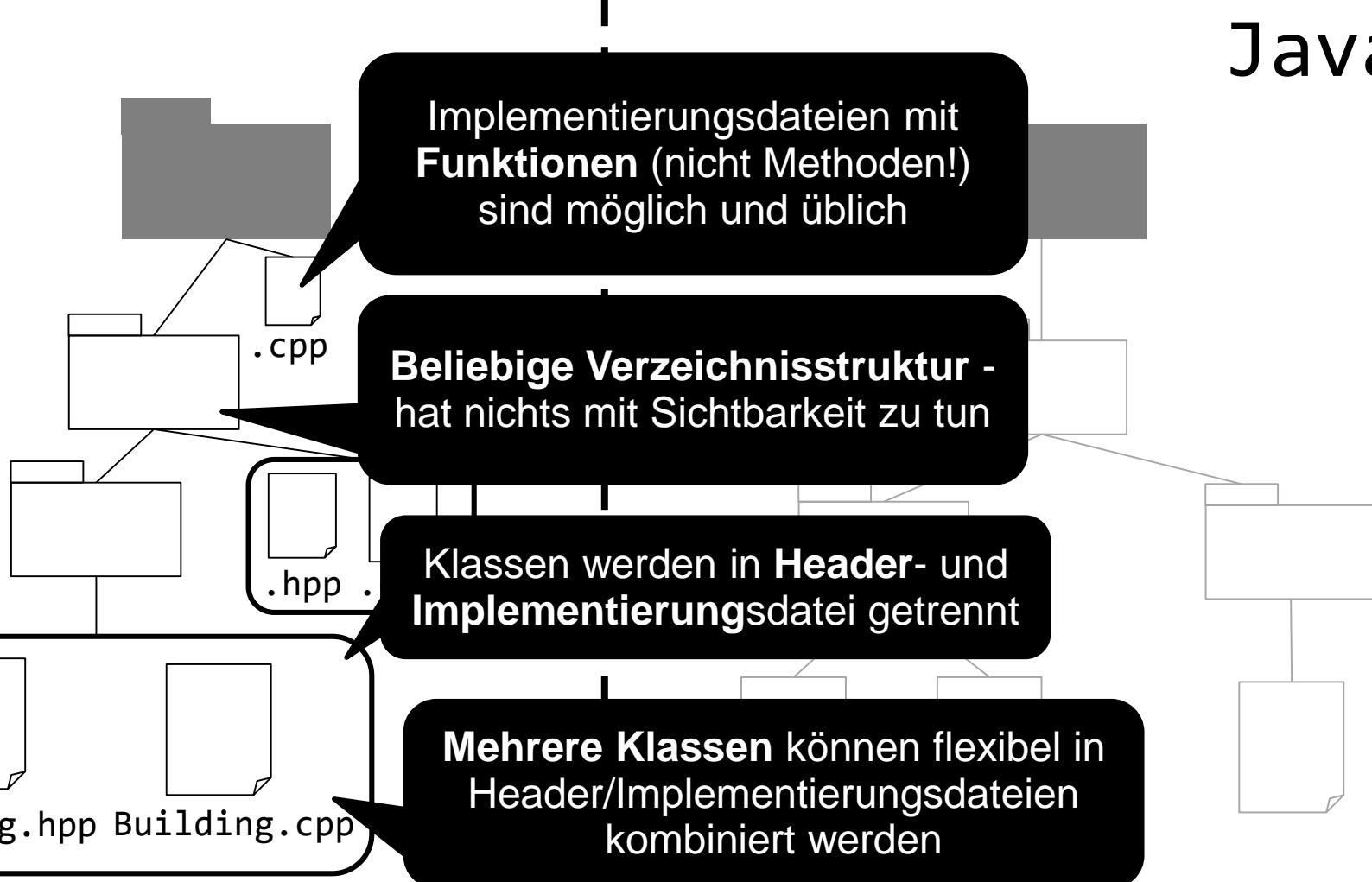
<http://cliparts.co/clipart/2613703>

# Projektstruktur



C++

Java



# Projektstruktur



## C++

- ▲ elevator-example
- ▲ include
  - ▷ Building.h
  - ▷ Elevator.h
  - ▷ ElevatorStrategy.h
  - ▷ Floor.h
  - ▷ Human.h
  - ▷ SimpleElevatorStrategy.h
- ▲ src
  - ▷ Building.cpp
  - ▷ Elevator.cpp
  - ▷ ElevatorStrategy.cpp
  - ▷ Floor.cpp
  - ▷ Human.cpp
  - ▷ SimpleElevatorStrategy.cpp
  - ▷ Simulation.cpp

## Java

- ▲ elevator-example
- ▲ src
  - ▲ de.tud.cpp.elevator.physical
    - ▷ Building.java
    - ▷ Elevator.java
    - ▷ Floor.java
    - ▷ Human.java
  - ▲ de.tud.cpp.elevator.simulation
    - ▷ Simulation.java
  - ▲ de.tud.cpp.elevator.strategy
    - ▷ ElevatorStrategy.java
    - ▷ SimpleElevatorStrategy.java

# Header und Implementierungs-Dateien



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

**Kommentare** wie in Java  
(auch // möglich)

```
#ifndef BUILDING_HPP_
#define BUILDING_HPP_

#include <vector>

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

**Include-Anweisungen** wie Import-Befehle in Java.

< ... > für Bibliotheken,  
" ... " für eigenen Code

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

    void runSimulation();

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

#endif /* BUILDING_HPP_ */
```

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

# Header und Implementierungs-Dateien



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

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

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

**Header-Datei** wird eingebunden

```
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;
}
```

**Methoden** werden implementiert  
(Details später)

# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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



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

# Exkurs: C++-Referenzen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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

The screenshot shows the cplusplus.com website's reference section for the <iostream> header. The left sidebar has sections for Information, Tutorials, Reference, Articles, and Forum. Under Reference, there are links for C library, Containers, Input/Output, and Multi-threading. The main content area shows the <iostream> header documentation, which is part of the Standard Input / Output Streams Library. It defines the standard input/output streams and includes notes about automatic inclusion of <iomanip>, <ios>, <iostream>, and <iosfwd>. A note also mentions that the iostream class is mainly declared in <iostream>. The page includes a search bar at the top.

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

The screenshot shows the en.cppreference.com website's reference section for the <iostream> header. The left sidebar lists categories like Strings library, Containers library, Algorithms library, Iterators library, and Numerics library. The main content area shows the <iostream> header documentation, which is part of the Standard Input / Output Streams Library. It defines the standard input/output streams and includes notes about automatic inclusion of <iomanip>, <ios>, <iostream>, and <iosfwd>. A note also mentions that the iostream class is mainly declared in <iostream>. The page includes a search bar at the top.

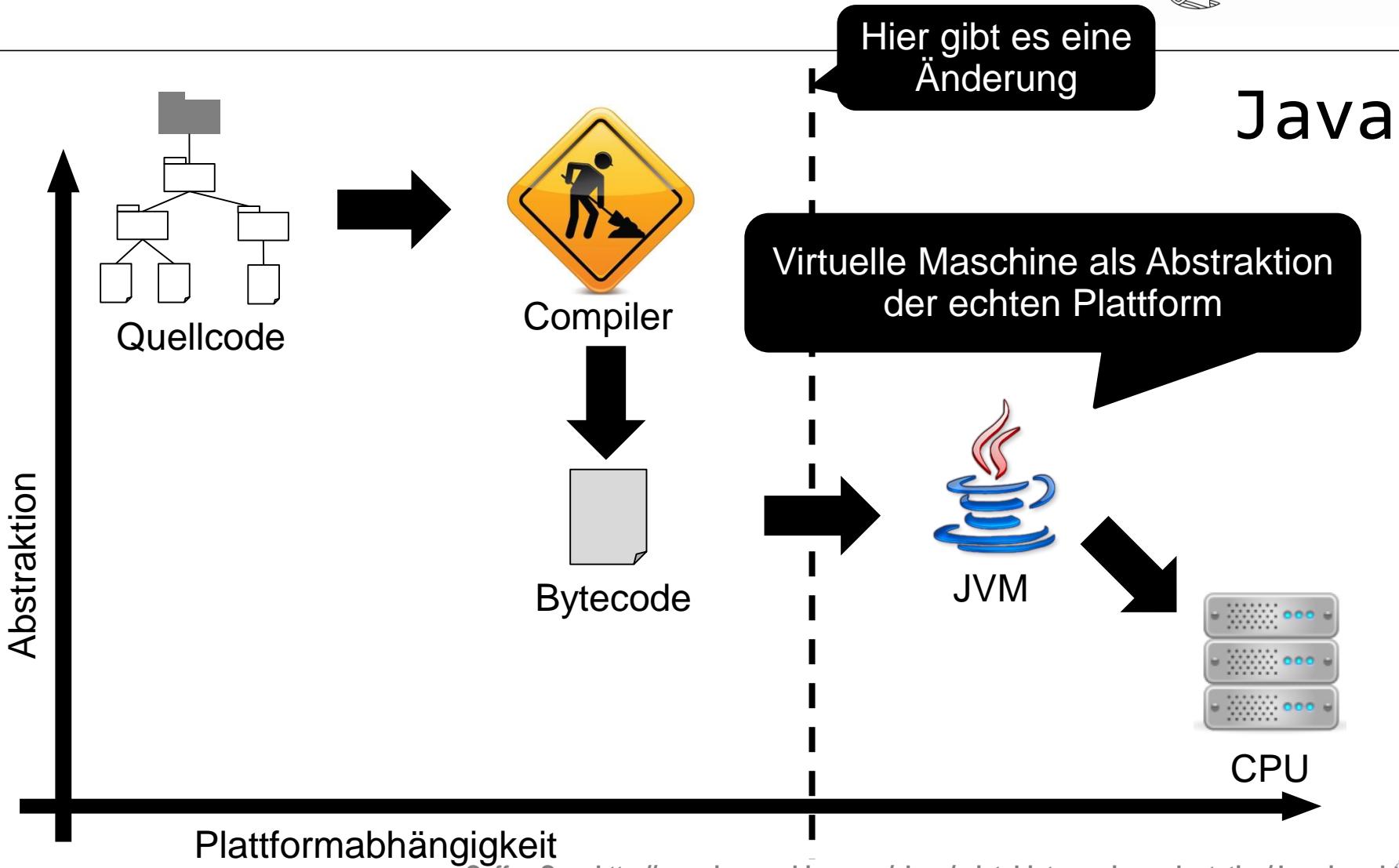


TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# KOMPILEIERUNG

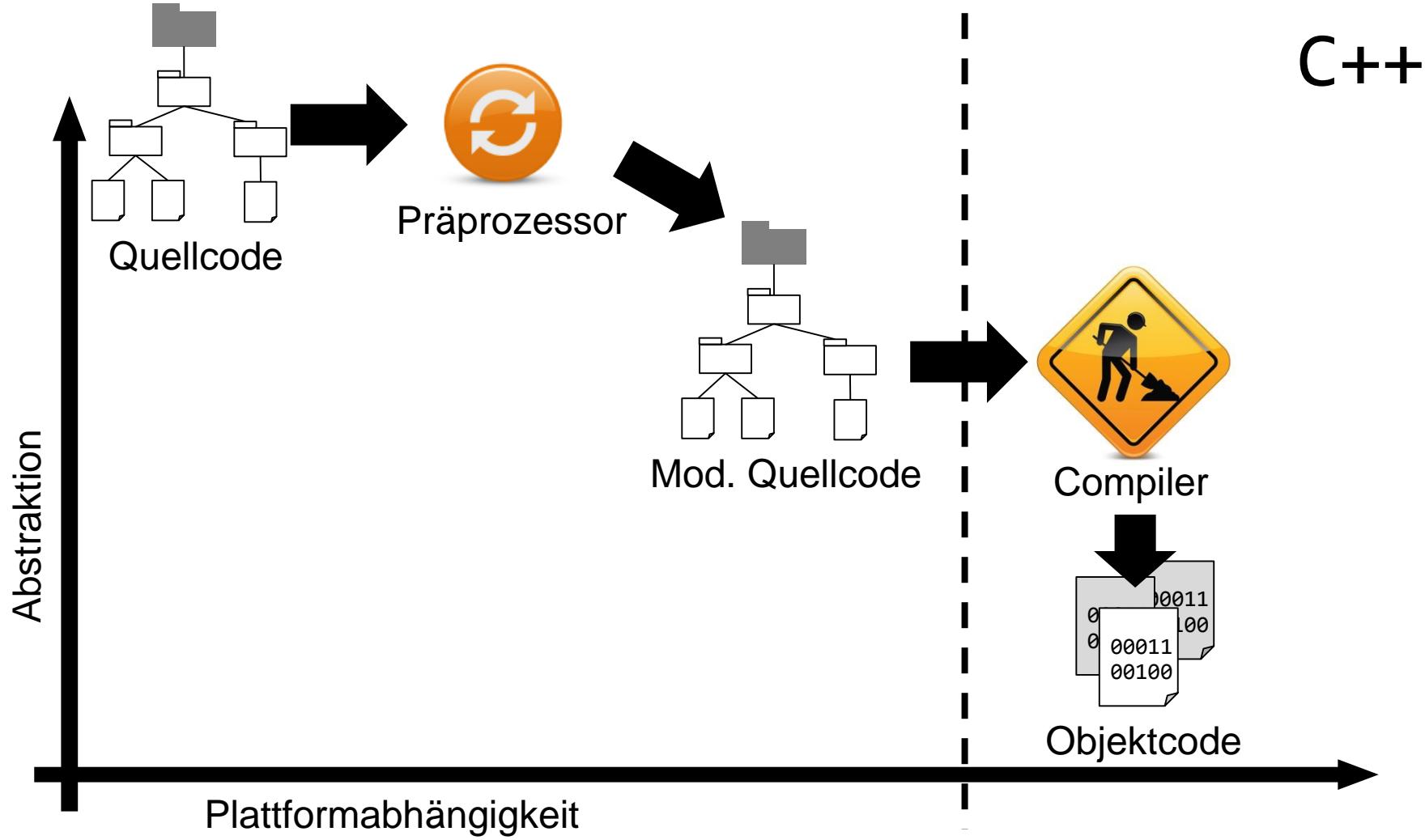


# Kompilierung

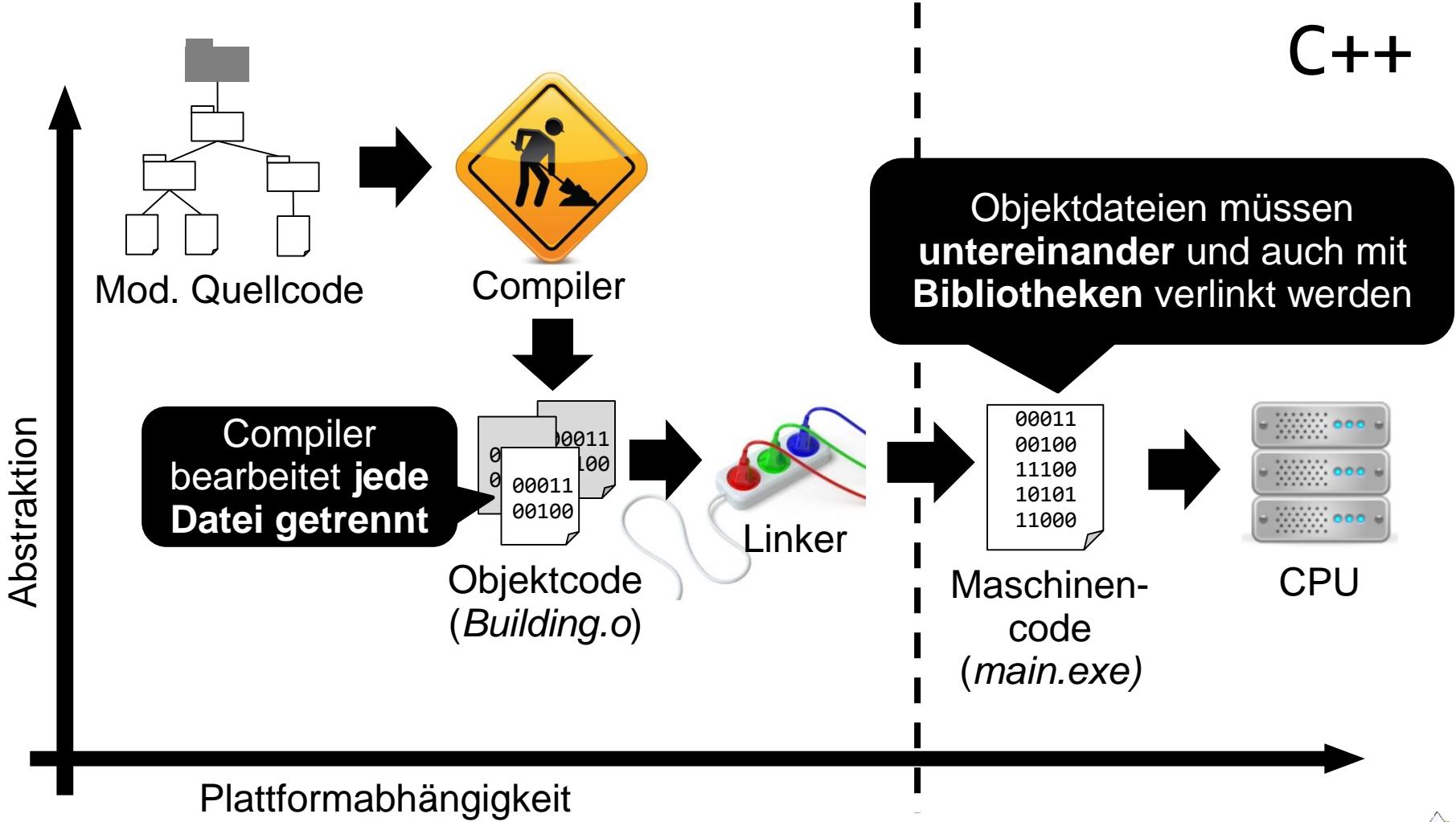


Coffee Cup: <http://www.iconarchive.com/show/cristal-intense-icons-by-tatice/Java-icon.html>

# Kompilierung für C/C++ I



# Kompilierung für C/C++ II



# Was genau macht der Präprozessor?



```
#ifndef BUILDING_HPP_
#define BUILDING_HPP_

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

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

    void runSimulation();

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

#endif /* BUILDING_HPP_ */
```

## Include Guard:

Schützt davor, dass *Building.h* mehrmals eingebunden wird

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

## Weitere Anwendungsfälle des Präprozessors:

- Unterscheidung zwischen DEBUG und RELEASE-Build (z.B. Logging)
- Betriebssystemerkennung (z.B. WIN32, UNIX)
- (in älteren C++-Varianten): Konstanten def.



## Fortgeschrittene Verwendung



### Schlüsselwort `return` neu definieren:

```
#define return DoSomeStackCheckStuff, return
```

Hoffentlich erinnert sich da später noch  
jemand dran...

### Auswertung von Ausdrücken zur Compile-Zeit:

```
/* Force a compilation error if condition is true */
#define BUILD_BUG_ON(condition) ((void)sizeof(char[1 -  
2*!!(condition)]))
```

Angeblich im Linux-Kernel verwendet, um Asserts  
zur Compile-Zeit durchzuführen

Quelle: <http://stackoverflow.com/questions/599365/what-is-your-favorite-c-programming-trick>

# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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

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

Wozu braucht man einen Präprozessor?  
Ist dies bei allen Sprachen der Fall?



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



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# SYSTEMSTART

# Systemstart



Viele Beispiele der Vorlesung sind im SVN-Repository.

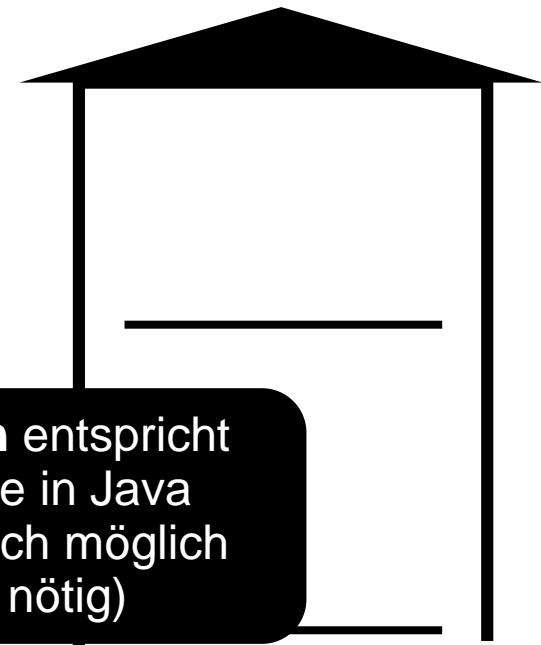
```
//=====
// Name: elevator-example-lecture.cpp
//=====

#include "Building.hpp"

// int main(int argc, char** argv)
int main() {
    Building building(3);
    building.runSimulation();
}
```

**Main-Funktion** entspricht Main-Methode in Java  
(Argumente auch möglich aber nicht nötig)

**Kein Rückgabewert nötig** (implizit 0 für „alles ordnungsgemäß durchgelaufen“), zumindest bei *gcc*



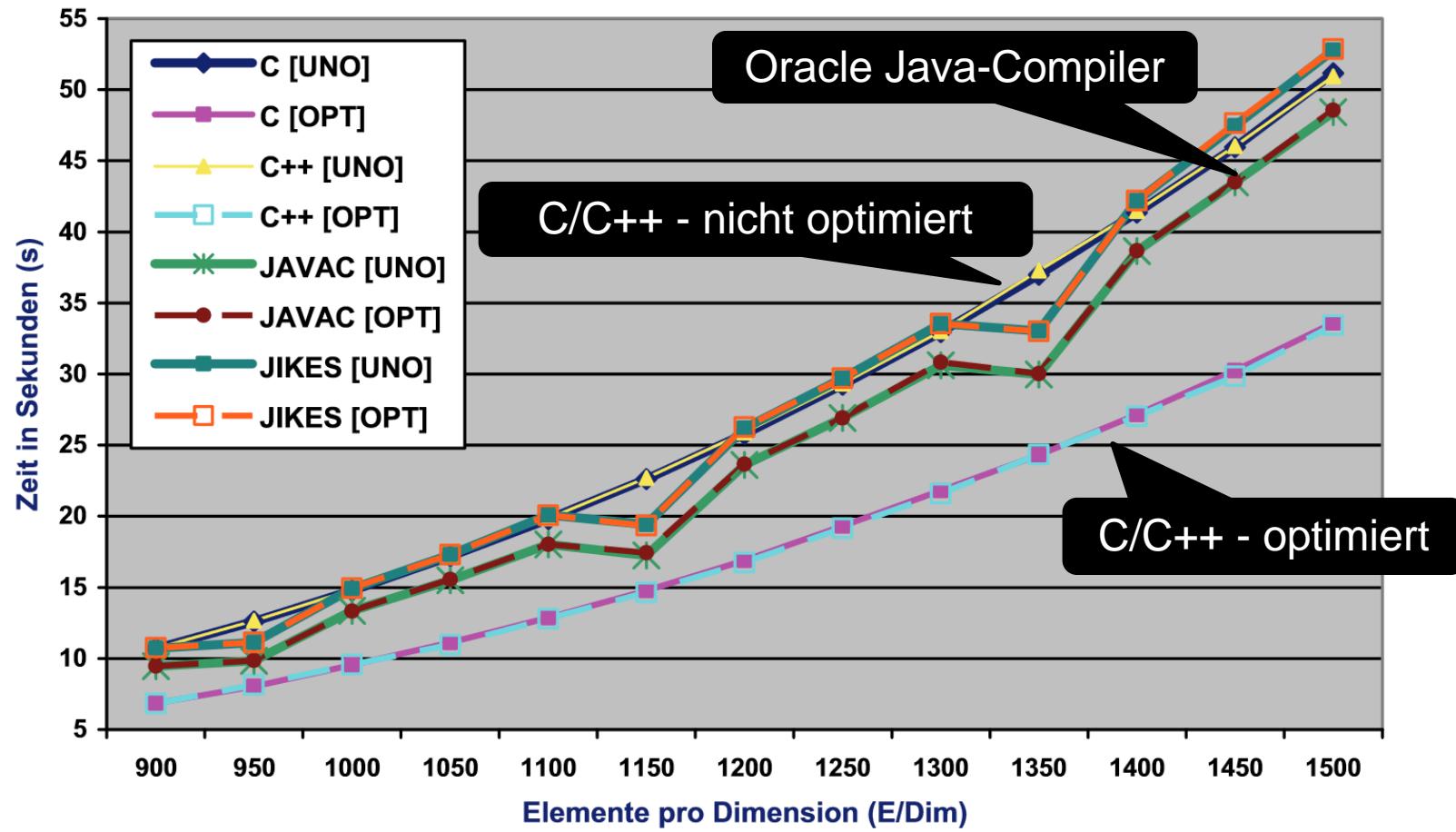


## Java vs. C++: Stärken und Schwächen?

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

# Laufzeitunterschied zwischen Java und C++

## Beispiel Matrixmultiplikation



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

# Demo – Virtuelle Maschine



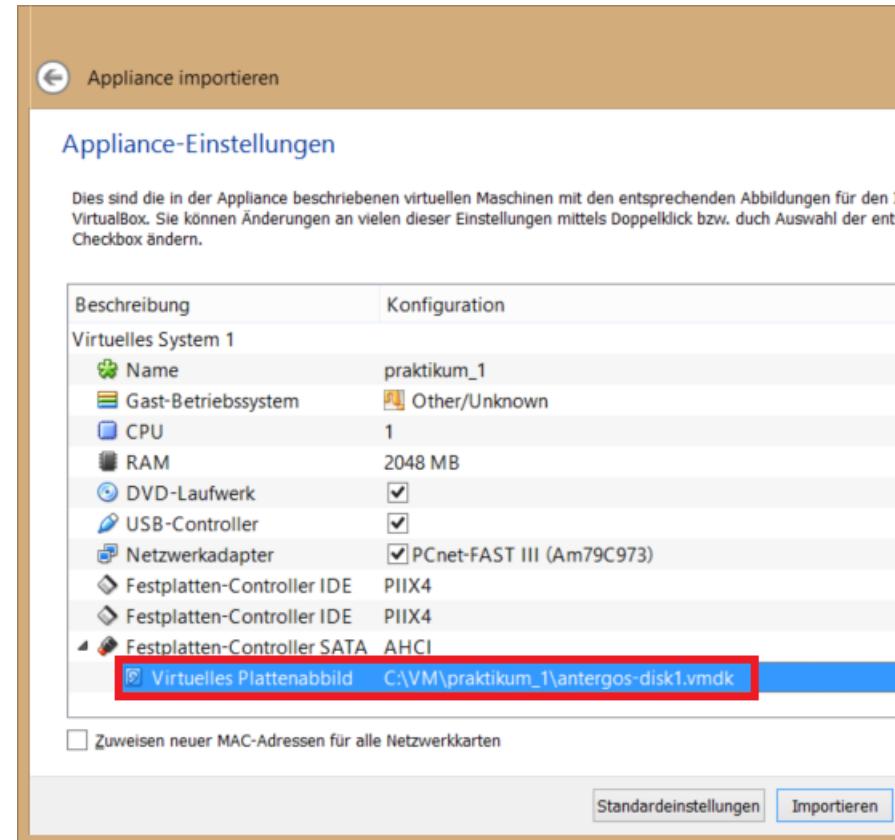
## 1. Herunterladen der VM:

- <http://tiny.cc/es-cppp-vm>
- User: cпп
- PW: >Cppp2015<

## 2. Importieren der Appliance antergos.ova

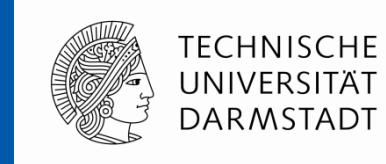
- **WICHTIG:** Beim Importieren muss der Pfad auf C:/VM/ gesetzt werden – ansonsten sprengt Ihr die Quota!

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



# 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?

## Stack

**Statischer** Speicher mit begrenzter Größe

Sehr effizient

Automatische Speicherfreigabe bei Rückkehr zur aufrufenden Funktion



## Heap

Dynamischer Speicher mit „beliebiger“ Größe

Relativ teuer



Flexible Speicherverwaltung zum beliebigen Zeitpunkt



## C++

```
int intOnStack = 42;  
cout << intOnStack << endl;
```

```
int *intOnHeap = new int(42);  
cout << intOnHeap << endl;  
cout << *intOnHeap << endl;
```

```
Building buildingOnStack(3);  
buildingOnStack.runSimulation();
```

```
Building *buildingOnHeap =  
    new Building(3);  
buildingOnHeap->runSimulation();
```

```
delete intOnHeap;  
delete buildingOnHeap;
```

## Java

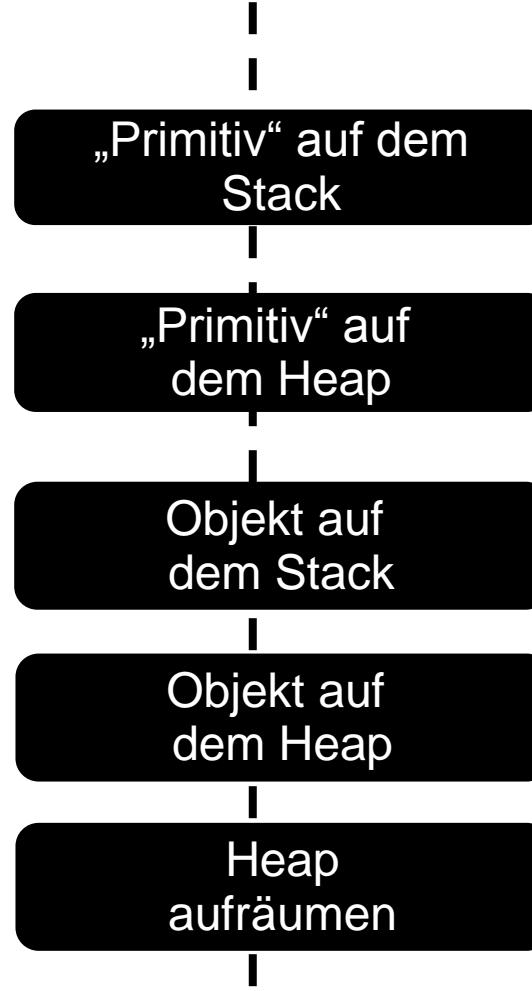
```
int intOnStack = 42;  
System.out.println(intOnStack);
```

// Not possible!

// Not possible!

```
Building buildingOnHeap =  
    new Building(3);  
buildingOnHeap.runSimulation();
```

// Handled by Garbage Collector!



# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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



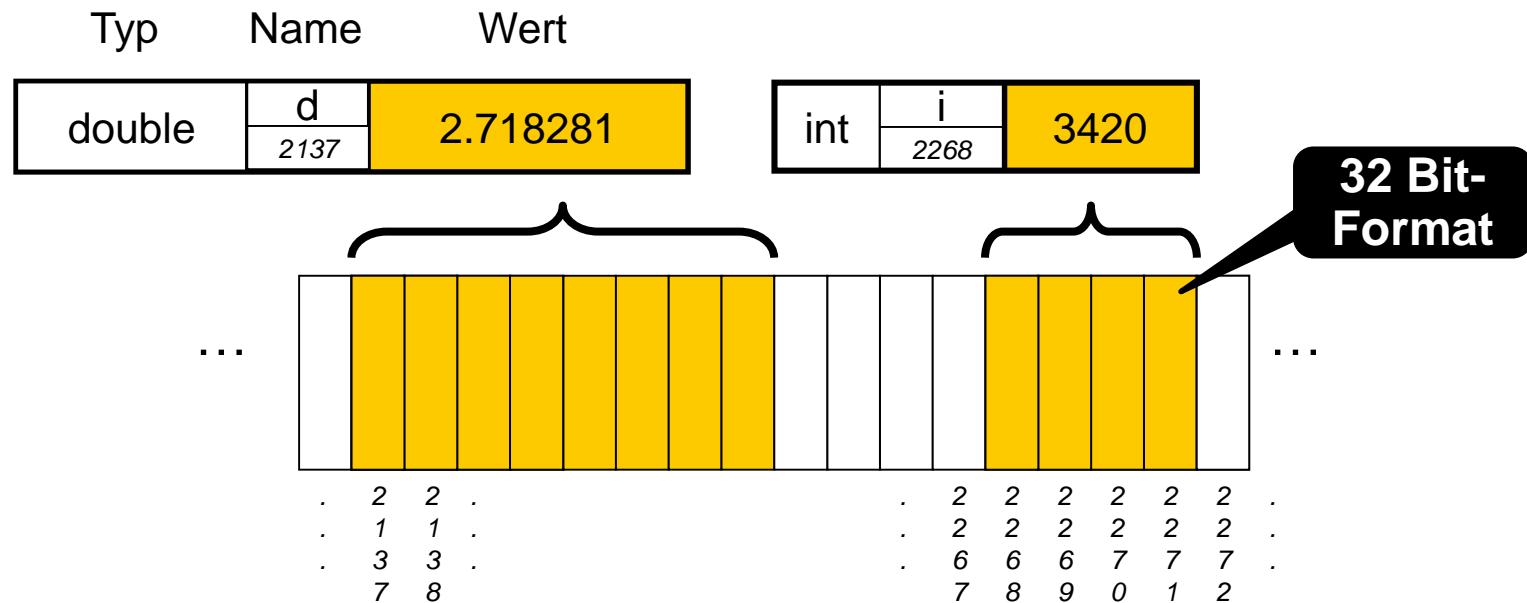
<http://cliparts.co/clipart/2613703>

# Variablen und Zeiger: Was ist eine Variable?



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

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

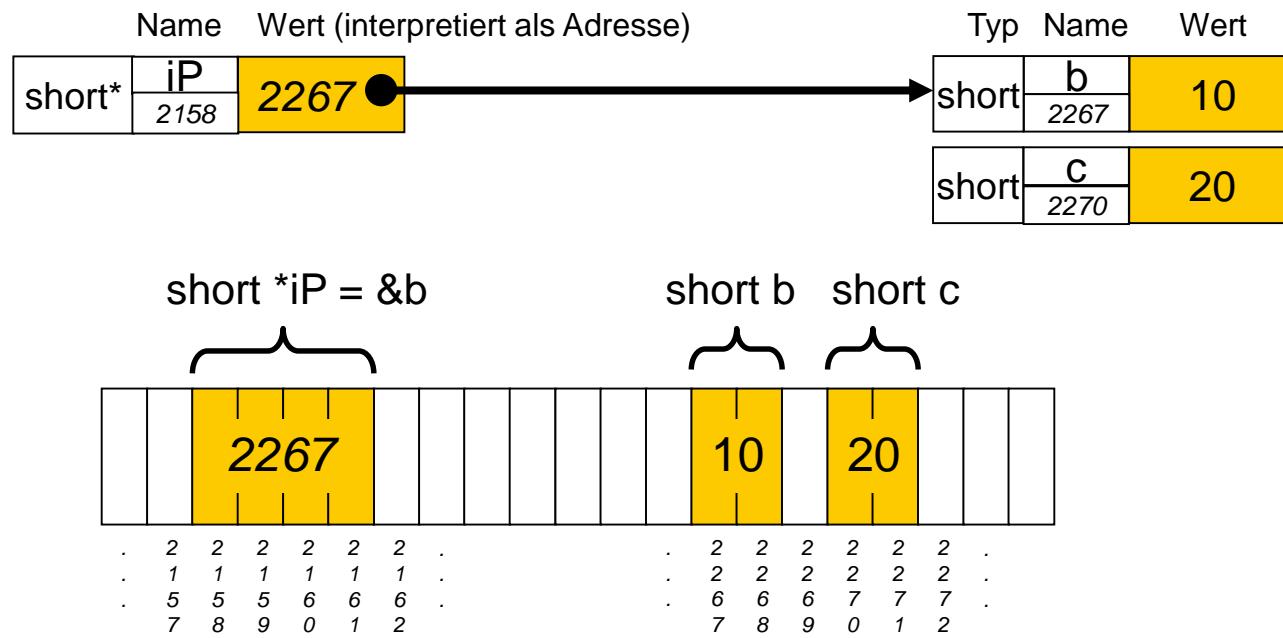
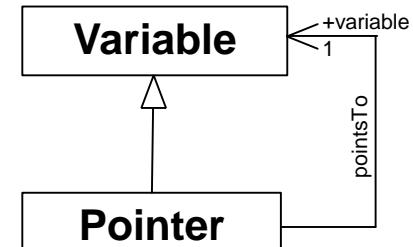


# Variablen und Zeiger: Was ist ein Zeiger?



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

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



# Variablen und Zeiger: Syntax



**Deklaration** (und Default-Initialisierung) eines Zeigers vom Typ *int\** (Zeiger auf *int*)

```
int i = 42;
```

```
int *iP;
```

```
iP = &i;
```

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

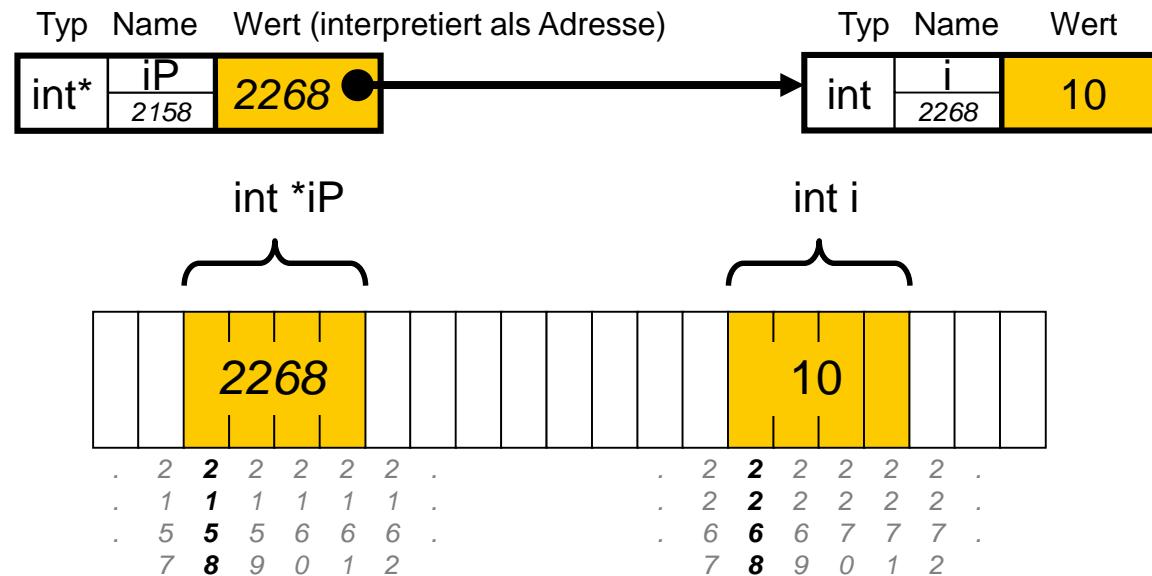
**Dereferenzierung** eines Zeigers, um den Inhalt zu erhalten

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

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

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

# Variablen und Zeiger: Syntax



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





## TODO

- `<cstddef>`
- C++11
- **NULL vs. 0 vs. 0x00**
- **“Mein größter Fehler”**

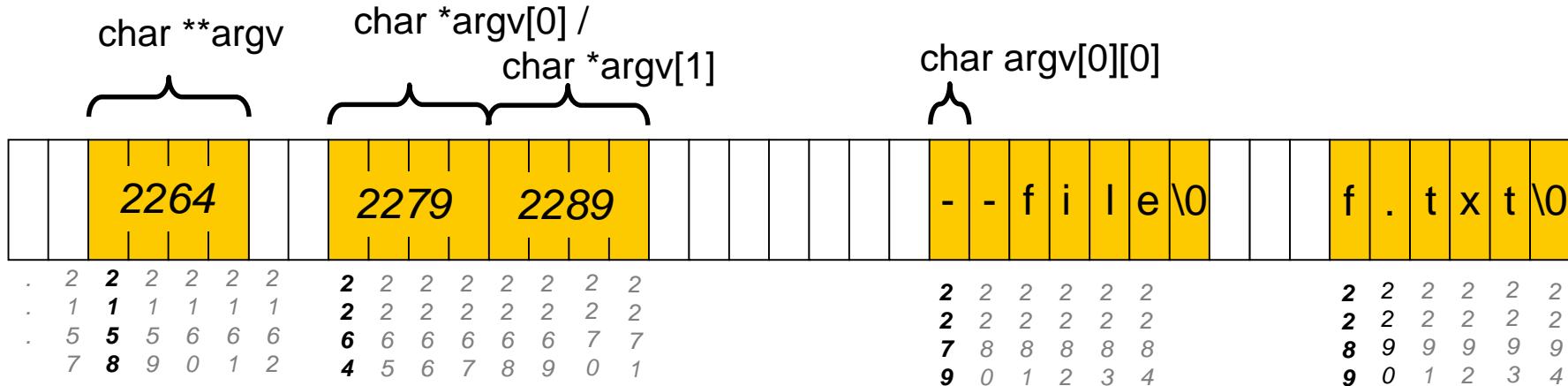
0x0

# Exkurs: Was heißt `char** argv`?



z.B. beim Aufruf `main.exe --file f.txt`

Strings (in C) sind Folgen von `char` (mit `\0` abgeschlossen)



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

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



2158  
--file 2279  
f.txt 2289

2279

**Generischer Pointer**

**Spezieller operator<<  
für `char*`**

# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Braucht man wirklich Zeiger? Wieso kann man nicht einfach nur normale Variablen verwenden?  
Wäre doch viel einfacher, oder?



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

# Unveränderlichkeit - *const*



## Zeiger auf Konstante

vs.

## Unveränderlicher Zeiger

```
int i = 42;
```

```
const int *iP;
```

```
iP = &i;
```

```
(*iP)++;
```

„Assignment of  
read-only  
variable iP“

```
|  
| int i;  
| int j = 7;
```

```
|  
| int *const jP = &j;
```

```
|  
| (*jP)++;
```

```
|  
| jP = &i;
```

Muss sofort initialisiert  
werden, kann nicht neu  
definiert werden

„Assignment of  
read-only  
variable jP“

## Unveränderlicher Zeiger auf Konstante:

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



## Eselstrücke:

Das *const* bezieht sich immer  
auf das „Nächstliegende“.

# Was ist eine (C++)-Referenz?



Eine **Referenz** ist ein **alias auf eine Variable**, der automatisch dereferenziert wird. Sie verhält sich wie ein *const Pointer*.

```
int i = 42;
```

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

```
(*iP)++;
```

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

```
cout << *iP << endl;
```

```
int i = 42;
```

```
int &iR = i;
```

```
iR++;
```

```
const int &iR = i;
```

```
cout << iR << endl;
```

Verhält sich  
wie Variable

# 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?

Gibt es eigentlich einen Unterschied zwischen

`int* iP`

und

`int *iP`

?



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

# Wieso **const**?



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

# Objektorientierung mit **const**



```
class Building {  
public:  
    Building(int number_of_floors);  
    ~Building();  
  
    void printFloorPlan() const; // Verändert den Zustand des Objekts nicht (Read-only-Zugriff)  
  
private:  
    std::vector<Floor> floors;  
    Elevator elevator;  
};  
  
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 von *building* aufgerufen werden

# Übersicht – Wo kann **const** auftauchen?



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

Unveränderliches Attribut (-> Initialisierungsliste nötig!).

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

Konstante

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

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

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

Methodenparameter *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

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

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



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



(Copy-)Konstruktor und Destruktor

# BAUEN UND ABREIßEN

# Konstruktor, Destruktor und Copy-Konstruktor



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

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

**Copy-Konstruktor**

**Destruktor**

```
Floor::Floor(string label, int number):  
    label(label),  
    number(number) {  
    cout << "Creating floor"  
        << number << "]" << endl;  
}
```

```
Floor::Floor(const Floor &floor):  
    label(floor.label),  
    number(floor.number+1) {  
    cout << "Copying floor"  
        << floor.number << "]" << endl;  
}
```

```
Floor::~Floor() {  
    cout << "Destroying floor ["  
        << number << "]" << endl;  
}
```

# Parameterübergabe bei Methodenaufrufen

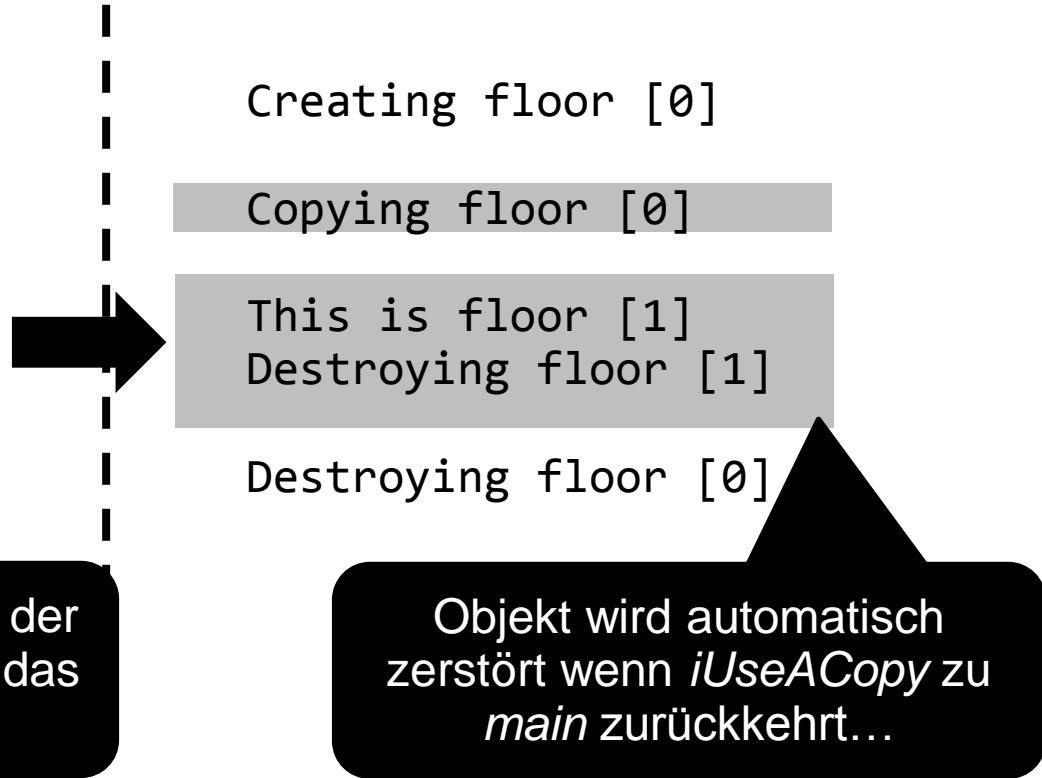


Parameter werden in C++ **immer** per Wert übergeben (**Call by Value**)

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

```
int main() {  
    Floor floor(0);  
    iWorkOnACopy(floor);  
}
```

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



# Parameterübergabe bei Methodenaufrufen (I)



Wieso nicht?

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);  
}
```

Eine Referenz wird  
„per Wert übergeben“

Es wird keine Kopie des  
Objekts angelegt

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

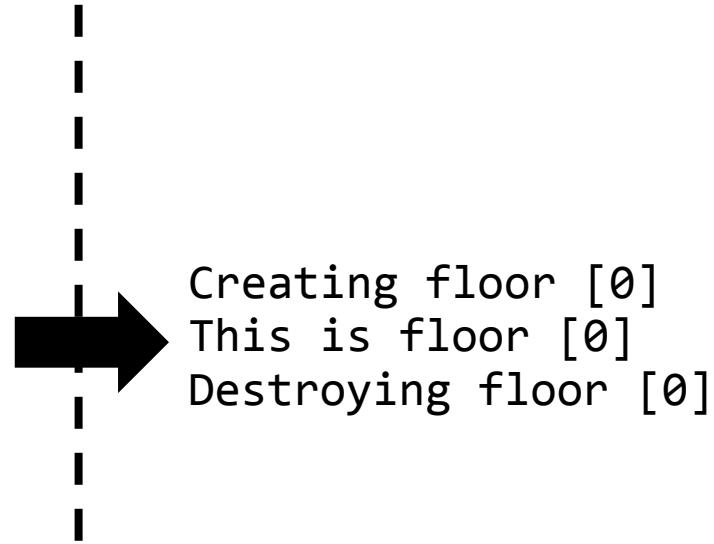
! *iUseAReference* kann  
aber das Objekt  
beliebig verändern!

# 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);  
}
```



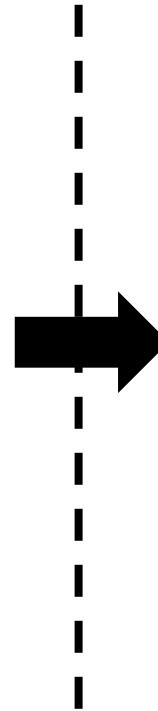
! Dies sollte grundsätzlich die Default-Übergabestrategie sein.

# 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);  
}
```



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

# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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

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

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



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



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

**Beispiel:**

```
class EnergyMinimizingStrategy {  
public:  
    inline EnergyMinimizingStrategy() {  
        cout << "Constructor called" << endl;  
    }  
  
    inline EnergyMinimizingStrategy(const EnergyMinimizingStrategy &a) {  
        cout << "Copy constructor called" << endl;  
    }  
  
    inline void operator=(const EnergyMinimizingStrategy &a) {  
        cout << "operator= called" << endl;  
    }  
};
```



Was soll das?



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

# Rule of Three



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

Beispiel:

```
#include <fstream>

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

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

Default Copy-Konstruktor versucht,  
*logfile* zu kopieren.

Ist das schlau?

# Rule of Three II



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

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



Hängende Zeiger und Speicherlecks

# STOLPERFALLEN BEI DER SPEICHERVERWALTUNG

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

Creating floor [0]

Copying floor [0]

Making next floor[1]

Destroying floor [1]

Next floor is floor [1]

Destroying floor [0]

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

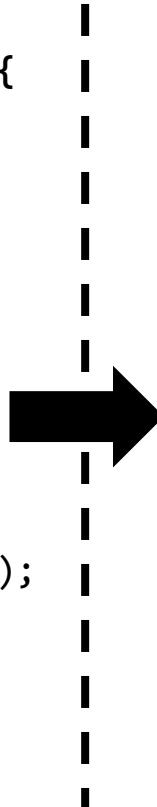
# Rückgabe von Objekten durch Kopieren



```
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;  
}
```

g++ ist in der Lage, zu erkennen, wann Kopien vermieden werden können:  
[http://en.wikipedia.org/wiki/Copy\\_elision](http://en.wikipedia.org/wiki/Copy_elision)



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

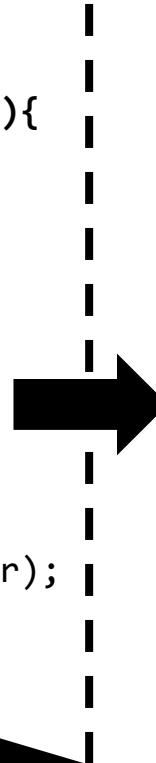
Tatsächlich

# 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

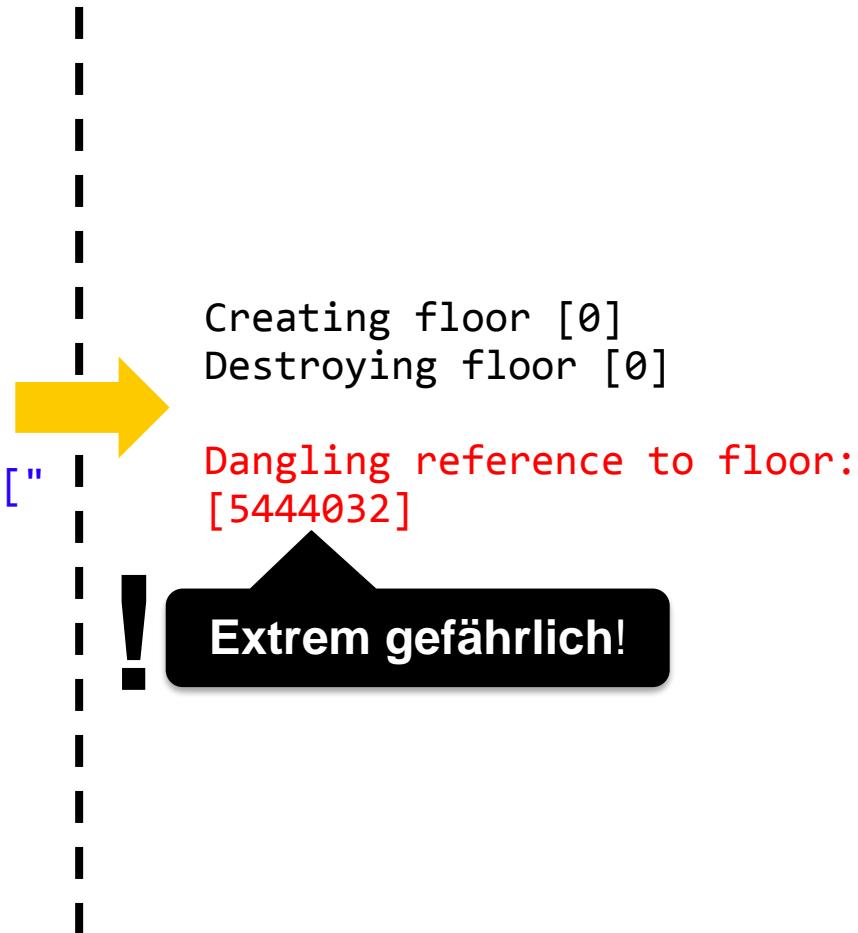
## Frühzeitige Zerstörung von Objekten



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

    delete floor;

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



# Hängende Zeiger

## Nochmalige Zerstörung von Objekten

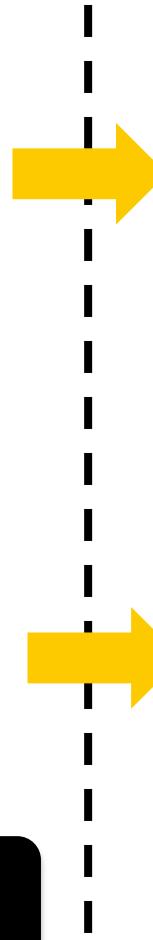


```
int main() {  
    Floor *floor = new Floor(0);  
  
    delete floor;  
    delete floor;  
}
```



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

Nach dem Löschen  
immer auf „null“ setzen!



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

Extrem gefährlich!

Creating floor [0]  
Destroying floor [1]

# Speicherlecks

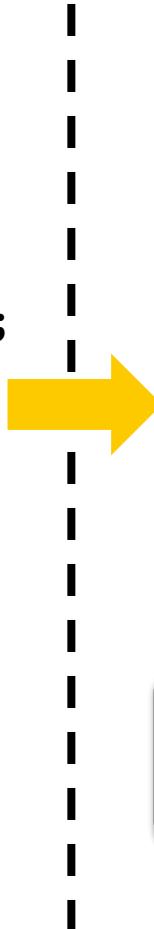


```
int main() {
    Floor *floor = new Floor(0);
    Floor *otherFloor = new Floor(1);

    floor = otherFloor; // -> floor [0]
    otherFloor = floor; // -> floor [0]

    delete floor;
    delete otherFloor;
}
```

Wieso ist das hier  
einfach nur doof?



```
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.

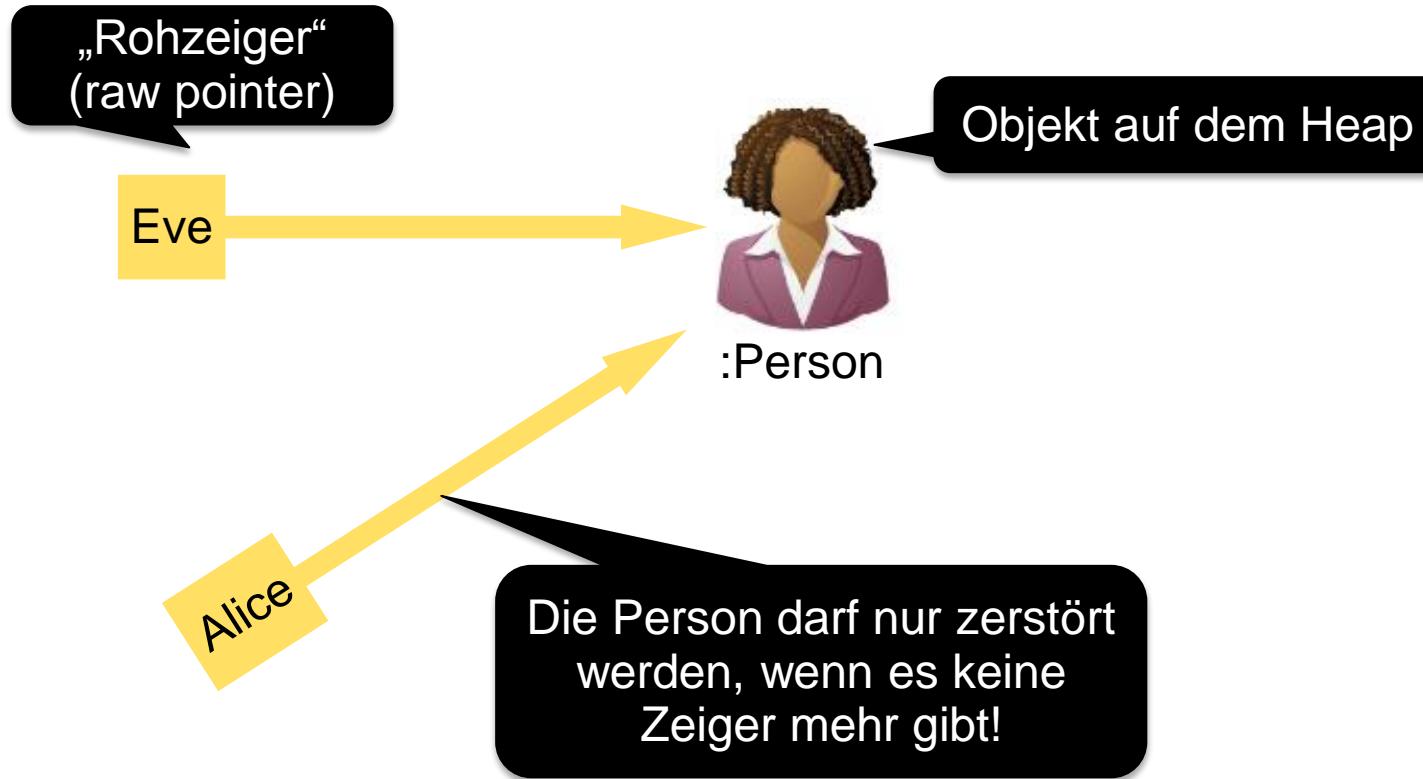
Wie können wir (1) – (3) klären und vor allem (4) immer garantieren?



---

## TODO: neue Folie zu Smart Pointern?

# Ohne Smart Pointer



# Intermezzo

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

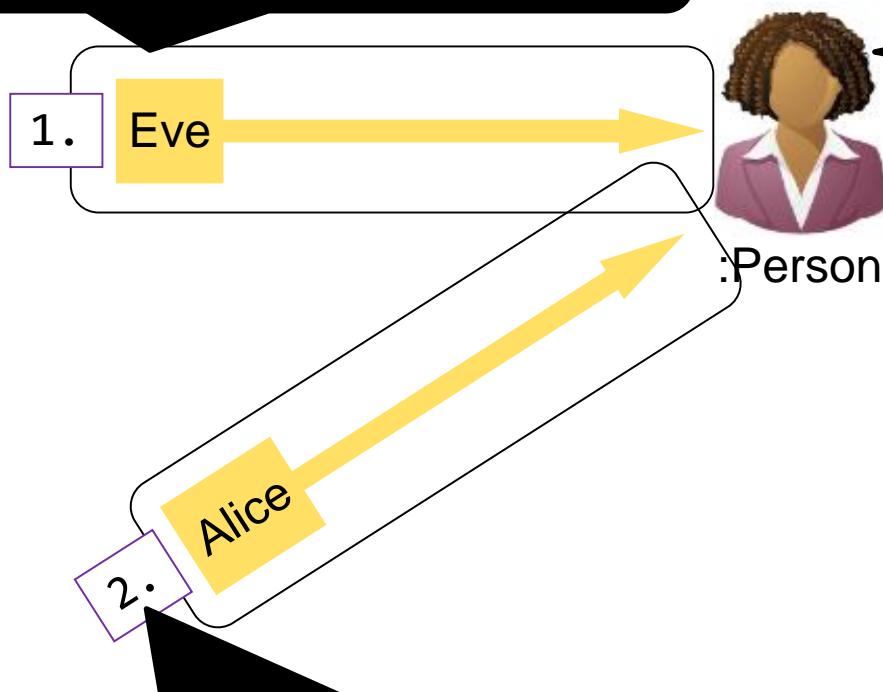


<http://cliparts.co/clipart/2613703>

# Mit boost::shared\_ptr



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



Objekt auf dem Heap

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

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

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

# Ohne Smart Pointer



Person.hpp

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

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

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

private:
    const string name;
};
```

Person.cpp

```
#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;
}
```

# Ohne Smart Pointer



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

#include "Person.hpp"

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

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

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

    delete passerBy;
    passerBy = 0;
}

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

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

    delete eve;
    eve = 0;
}
```

main.cpp

Created Eve

Greeting Eve  
Isn't the weather quite pleasant today,  
Eve?

Created Sir

Isn't the weather quite pleasant today,  
Sir?

Good bye Sir

Greeting Eve

Isn't the weather quite pleasant today,  
Eve?

Created Sir

Isn't the weather quite pleasant today,  
Sir?

Good bye Sir

Good bye Eve

# Mit Shared Pointer



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

Person.hpp

```
#include <boost/shared_ptr.hpp>
```

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

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

private:
    const string name;
};

typedef boost::shared_ptr<Person>
PersonPtr;

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

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

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;
}
```

Person.cpp

# Mit Smart Pointer



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

#include "Person.hpp"

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

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

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

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

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

main.cpp



Created Eve

Greeting Eve  
Isn't the weather quite pleasant today,  
Eve?

Created Sir

Isn't the weather quite pleasant today,  
Sir?  
**Good bye Sir**

Greeting Eve

Isn't the weather quite pleasant today,  
Eve?

Created Sir

Isn't the weather quite pleasant today,  
Sir?  
**Good bye Sir**

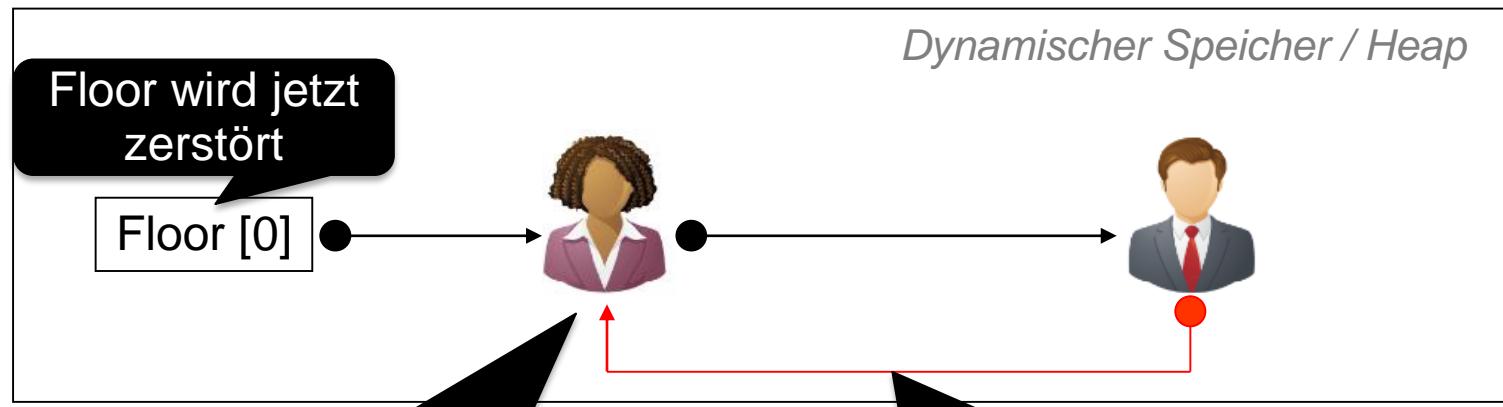
**Good bye Eve**

# Weak SmartPointer: Motivation



*boost::shared\_ptr* ist nicht perfekt:

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



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

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



- **weak\_ptr** für eine Richtung der Beziehung zwischen Personen verwenden (z.B.: Eve zeigt stark auf Bob, Bob schwach auf Eve)
- **shared\_ptr** um „extern“ auf Personen zu zeigen (Floor auf Person )
- Ein schwacher (weak) Zeiger verlangt, dass mindestens ein „starker“ (strong) Zeiger (z.B. ein **shared\_ptr**) bereits auf die Person zeigt
- Person wird gelöscht, sobald nur noch schwache Zeiger darauf verweisen

# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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

Wie hätte man das sonst lösen können?

Was wäre die Konsequenz?



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

# Mögliche Lösung für zyklische Zeiger



---

Wir verzichten einfach ganz auf Zeiger.

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



Welches neue Problem  
handeln wir uns damit ein?



Eine Person existiert jetzt  
mehrfach!

# Mögliche Lösung für zyklische Zeiger II



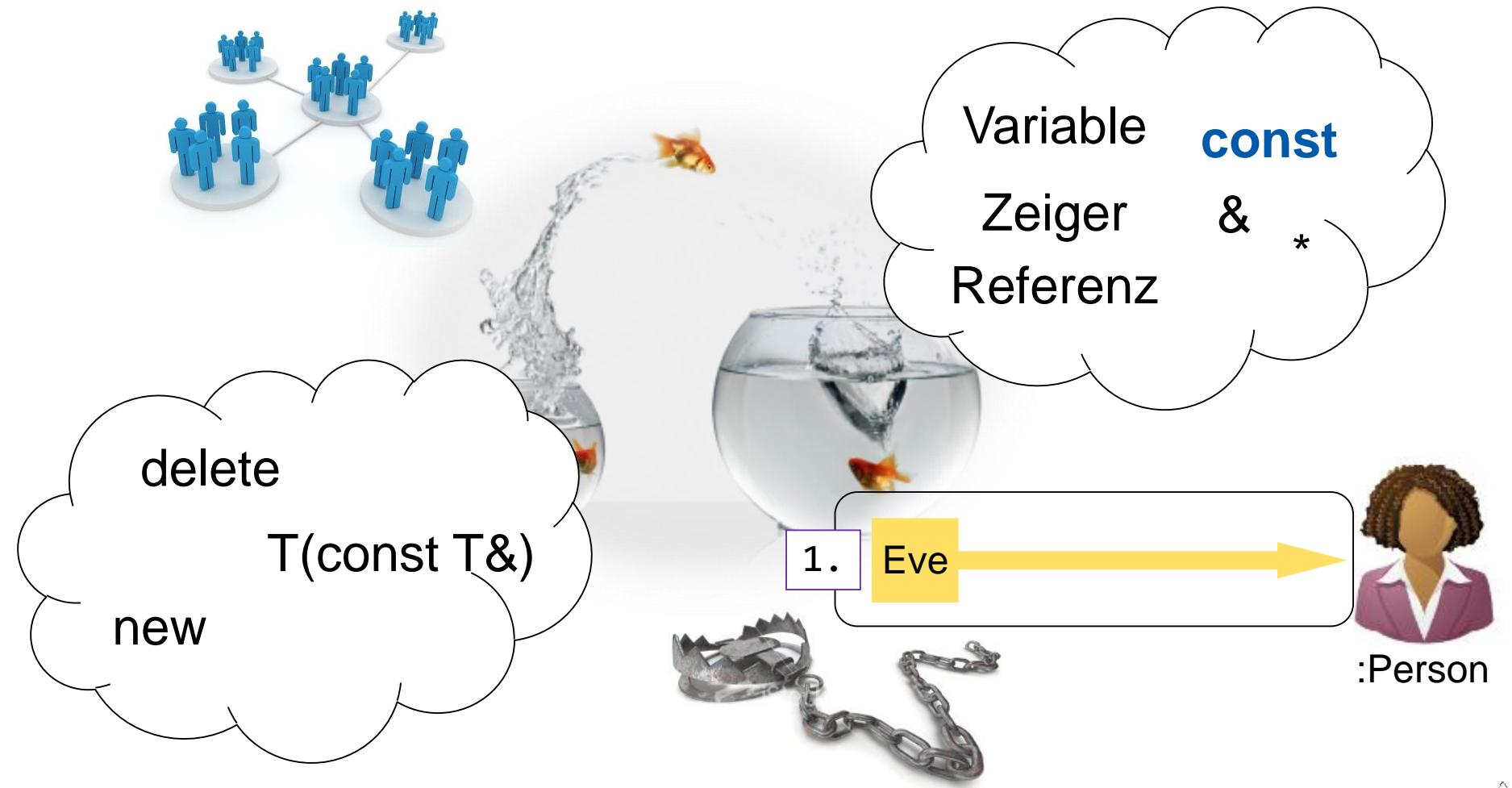
```
int main(int argc, char **argv) {  
  
    Person eve("Eve", 55.0); // initial weight: 55kg  
    Person bob("Bob", 80.0); // initial weight: 80kg  
  
    cout << bob.getName() << " has weight " << bob.getWeight() << endl;  
  
    Person::makeFriends(eve, bob);  
  
    Person &bobAsEvesFriend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

## Ausgabe:

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

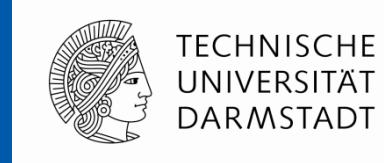
Kann man mit immutablen  
Objekten (wie *java.lang.String*)  
umgehen.

# 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

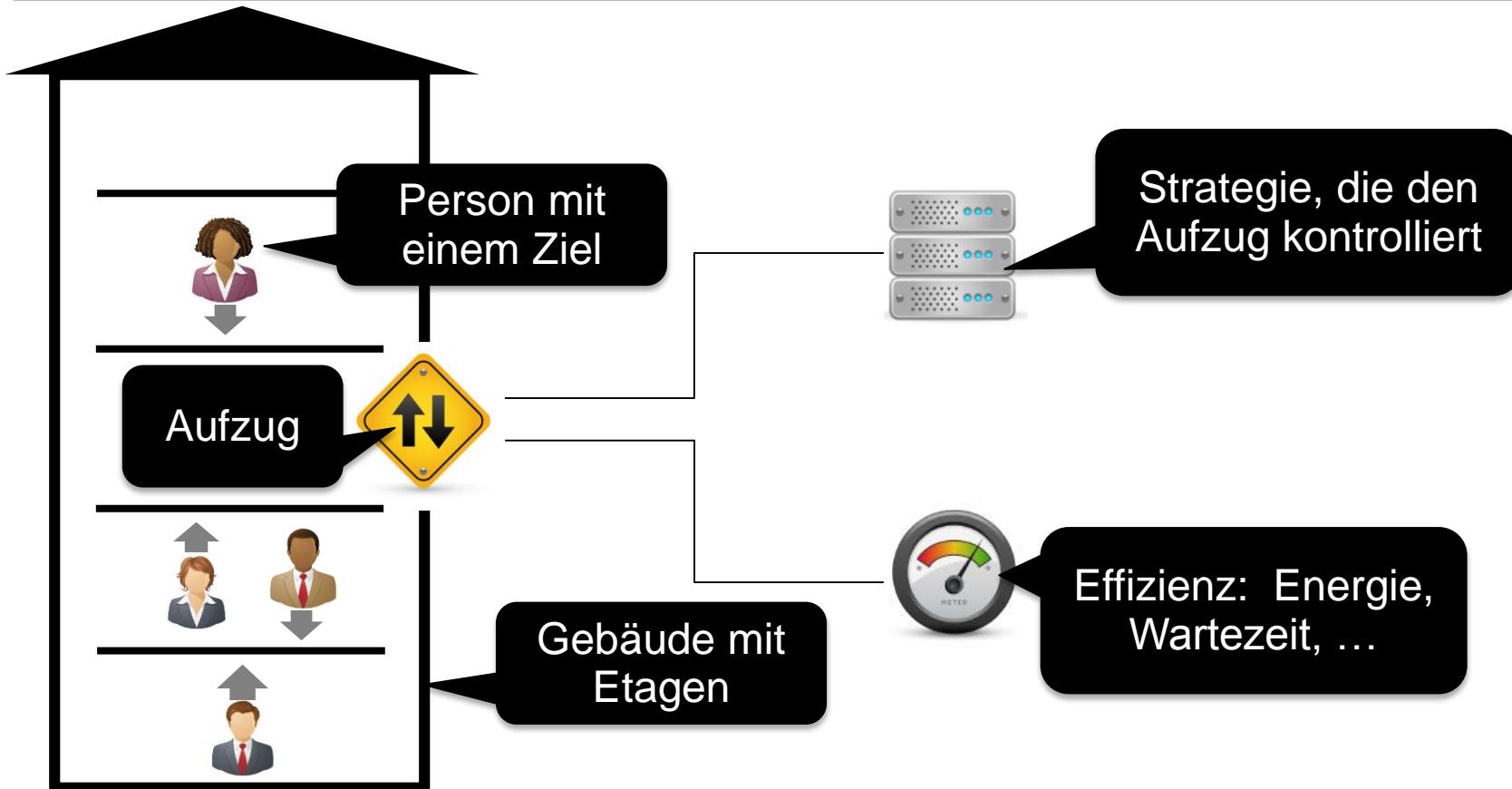
Dept. of Computer Science (adjunct Professor)

[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)

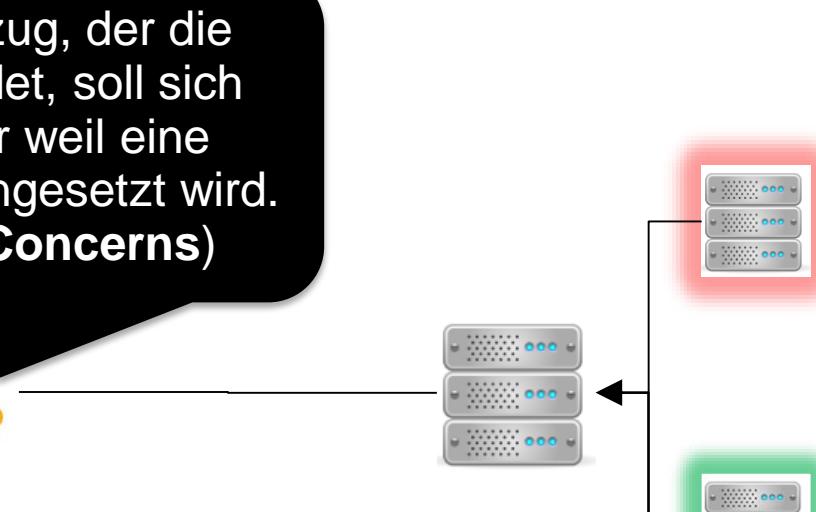
# Was ist Polymorphie?



# Was ist Polymorphie?



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



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

# Lösung ohne Polymorphie



TODO: Enum/consts für Strategien

```
void Elevator::moveToNextFloor(int strategy){  
    switch(strategy){  
        case 0:  
            cout << "Choose next floor to minimize energy."  
            << endl;  
            break;  
        case 1:  
            cout << "Choose next floor to minimize waiting time."  
            << endl;  
            break;  
        // and so on ...  
    }  
}
```



„Dispatch“ geschieht von Hand  
mit Hilfe einer „Tabelle“

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

# Lösung mit Polymorphie

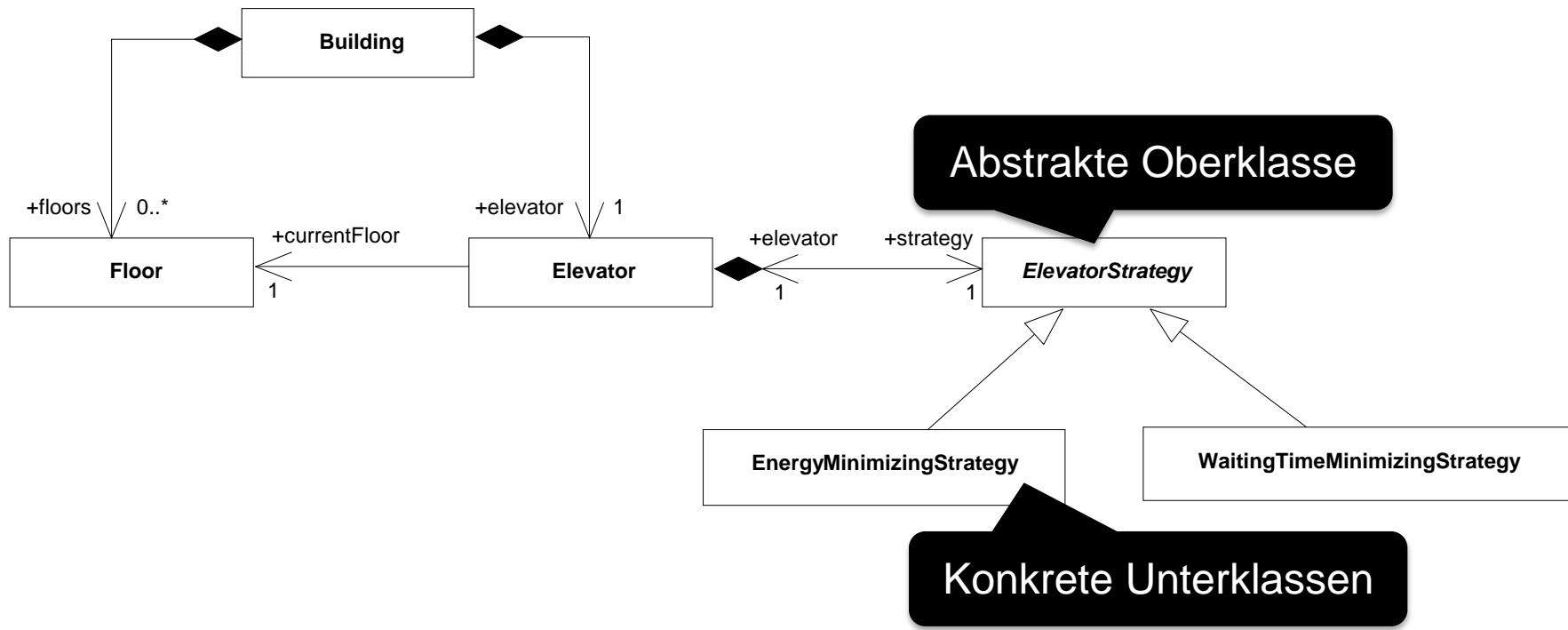


```
void Elevator::moveToNextFloor(){  
    currentFloor = strategy->next(this);  
}
```

Konkrete Strategie wird bei der Erzeugung des Aufzugs gesetzt.

Dieser Code behandelt die Strategie polymorph und muss für neue Strategien nicht verändert werden!

# Aufzugsimulation (reloaded)



# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Nochmal – was ist der Vorteil von Polymorphie?

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

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



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

# Ein Blick auf die Klassen Floor



```
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

```
#include "Floor.hpp"

Floor::Floor(int number):
    number(number) {
    cout << "Floor(): "
        << "Creating floor ["
        << number
        << "]" << endl;
}

Floor::Floor(const Floor& floor):
    number(floor.number) {
    cout << "Floor(const Floor&): "
        << "Copying floor ["
        << floor.number
        << "]" << endl;
}

Floor::~Floor() {
    cout << "~Floor(): "
        << "Destroying floor ["
        << number
        << "]" << endl;
}
```

Floor.cpp

Kleine Methoden können  
*inline* definiert werden!

# Ein Blick auf die Klassen



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

```
#include <boost/shared_ptr.hpp>
#include "Floor.hpp"

class Elevator;

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

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

typedef
boost::shared_ptr<ElevatorStrategy>
ElevatorStrategyPtr;

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

In der Impl-Datei ist dies  
aber kein Problem!

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

using namespace std;

ElevatorStrategy::ElevatorStrategy() {
    cout << "ElevatorStrategy(): "
        << "Creating basic strategy"
        << endl;
}

ElevatorStrategy::~ElevatorStrategy() {
    cout << "~ElevatorStrategy(): "
        << "Destroying basic strategy"
        << endl;
}

const Floor*
ElevatorStrategy::next(const Elevator *elevator) const {
    cout << "ElevatorStrategy::next(...): "
        << "Using basic strategy ..."
        << endl;

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

Sinnvolle Strategien entwickeln  
wir in der Übung ☺

# Ein Blick auf die Klassen Elevator



Elevator.hpp

```
#include "ElevatorStrategy.h"
#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;
};
```

Typen ohne Namen  
auch möglich

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

```
include "Elevator.hpp"
```

```
llevator::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);
```

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

Elevator.cpp

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

# Ein Blick auf die Klassen Building



```
#include <vector>

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

class Building {
public:
    Building(int number_of_floors,
              ConstElevatorStrategyPtr strategy);
    ~Building();

    inline int number_of_floors() const {
        return floors.size();
    }

    inline Elevator& get_elevator() {
        return elevator;
    }

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

Building.hpp

Strategie wird  
an Elevator  
weitergereicht

```
#include <iostream>
using std::cout;
using std::endl;
#include <algorithm>
#include "Building.hpp"

Building::Building(int number_of_floors,
                   ConstElevatorStrategyPtr strategy):
    floors(number_of_floors, Floor(0)),
    elevator(&floors[0], strategy)
{
    for (int i = 0; i < number_of_floors; i++)
        floors[i].set_number(i);

    cout << "Building(...): "
         << "Creating building with "
         << number_of_floors
         << " floors." << endl;

    cout << "Building(...): "
         << "Elevator is on Floor: "
         << elevator.get_current_floor()->get_number()
         << endl;
}

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

Building.cpp

# Ein Blick auf die Klassen EnergyMinimizingStrategy



EnergyMinimizingStrategy.h

Vererbung in C++  
wird so angegeben

```
#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

EnergyMinimizingStrategy.cpp

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

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

EnergyMinimizingStrategy::EnergyMinimizingStrategy()
    : ElevatorStrategy() {
    cout << "EnergyMinimizingStrategy(): "
        << "Creating energy minimizing strategy"
        << endl;
}

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

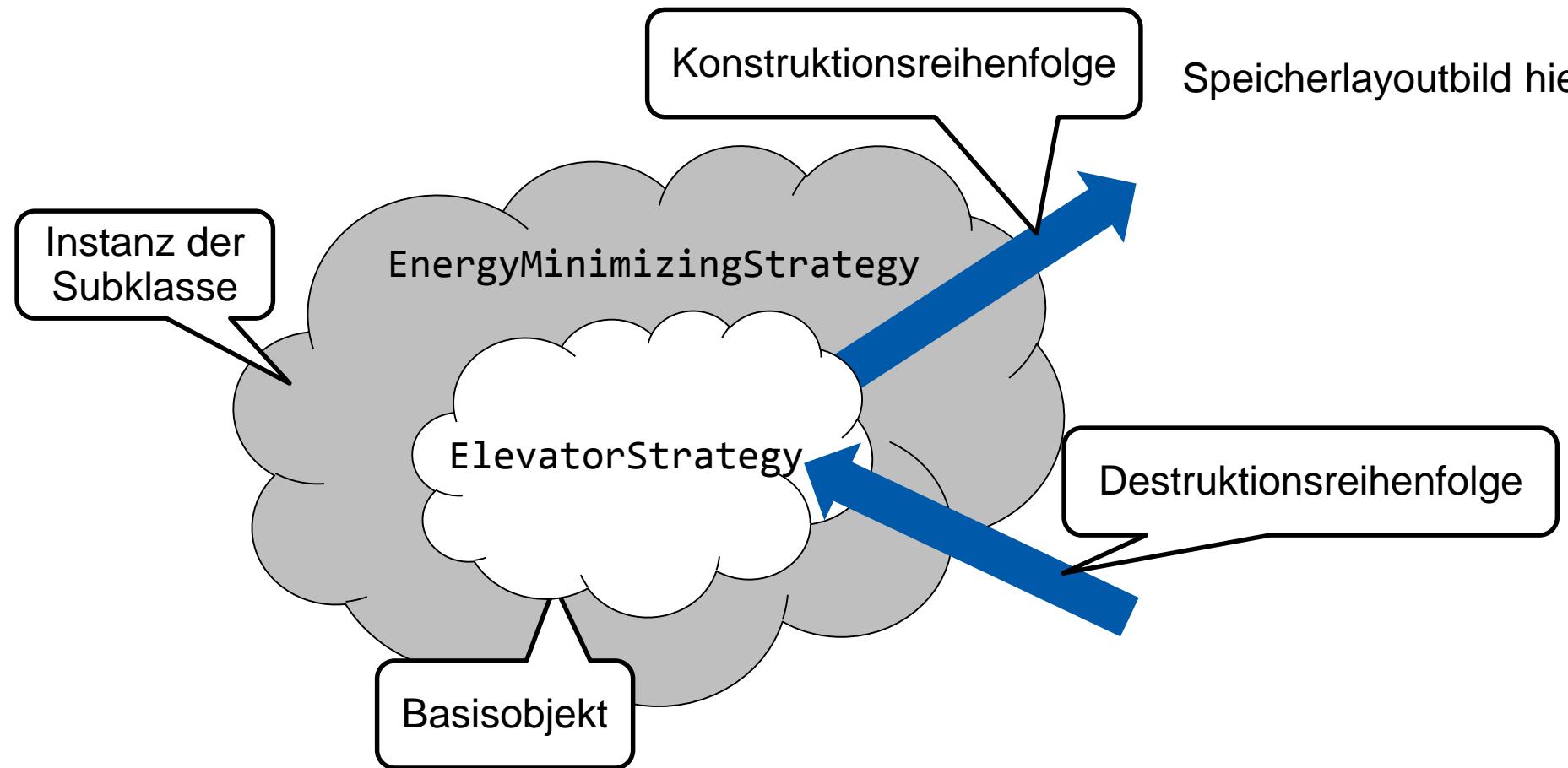
const Floor* EnergyMinimizingStrategy::
next(const Elevator *elevator) const{
    cout << "EnergyMinimizingStrategy::next(...): "
        << "Perform some complex calculation ..."
        << endl;
}

return elevator->getCurrentFloor();
```

# Konstruktion und Dekonstruktion von Objekten bei Vererbung



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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



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

# Probelauf unserer Simulation



Eure Aufgabe in der  
Übung

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

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

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

    // Do something...

    ConstElevatorStrategyPtr strategy(strg);
    Building hbi(6, strategy);

    hbi.getElevator().moveToNextFloor();
}
```

# Probelauf unserer Simulation



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

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

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

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

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

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

```
~ElevatorStrategy(): Destroying basic strategy
```

Konstruktoren werden  
richtig aufgerufen

! Polymorpher Aufruf hat  
aber **nicht funktioniert!**

! Destruktor der  
Subklasse wurde nicht  
aufgerufen!

# Virtuelle Methoden

---

Im Gegensatz zu Java ist bei C++ aus Effizienzgründen die polymorphe Behandlung von Methoden **per Default ausgeschaltet**

Es muss explizit mit dem Schlüsselwort **virtual** angegeben werden, welche Methoden polymorph zu behandeln sind

# Virtuelle Methoden



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

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

Methoden werden als virtuell gekennzeichnet (nur im Header)

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

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

# Intermezzo

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

Wie ist das mit virtuellen Konstruktoren?



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

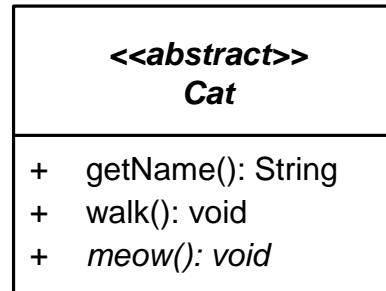
# Virtual Method Table

## Der Mechanismus der dynamischen Bindung



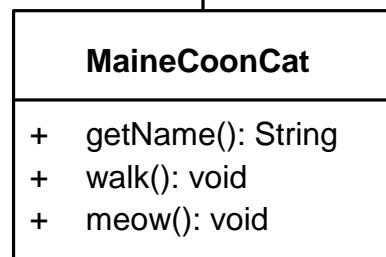
Egal wie der Pointer auf ein Objekt deklariert ist (z.B. *ElevatorStrategy*\*), das Objekt behält seinen Typ (z.B. *EnergyMinimizingStrategy*).

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	MainCoonCat::walk
meow	MainCoonCat::meow

Falls kein Eintrag:  
Verwende Methode des  
Typs des Pointers.

# Probelauf mit virtuellen Methoden



ElevatorStrategy(): Creating basic strategy

EnergyMinimizingStrategy(): Creating energy minimizing strategy

Floor(): Creating floor [0]

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

~Floor(): Destroying floor [0]

Elevator(): Creating elevator.

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

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

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

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

~Building(): Destroying building.

~Elevator(): Destroying elevator.

~Floor(): Destroying floor [0]

~Floor(): Destroying floor [1]

~Floor(): Destroying floor [2]

~Floor(): Destroying floor [3]

~Floor(): Destroying floor [4]

~Floor(): Destroying floor [5]

~EnergyMinimizingStrategy(): Destroying energy minimizing strategy

~ElevatorStrategy(): Destroying basic strategy



Polymorpher Aufruf  
funktioniert jetzt



Und alle Destruktoren werden in der  
richtigen Reihenfolge aufgerufen

# Pure Virtual

*ElevatorStrategy* kann nicht mehr instantiiert werden.



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

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

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

# Intermezzo



Wieso sind virtuelle Methoden „teuer“?

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

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

Was ist der Unterschied zwischen Zeilen (2) und (3):

1. EnergyMinimizingStrategy strg0;
2. EnergyMinimizingStrategy strg1 = strg0;
3. EnergyMinimizingStrategy strg2(strg0);



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

# Copy Elision



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

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

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

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



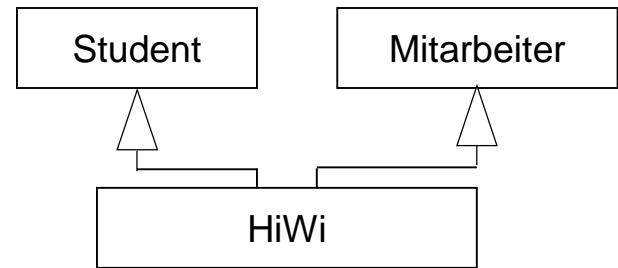
[EN] /Copy Elision

## Ausgabe:

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

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

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

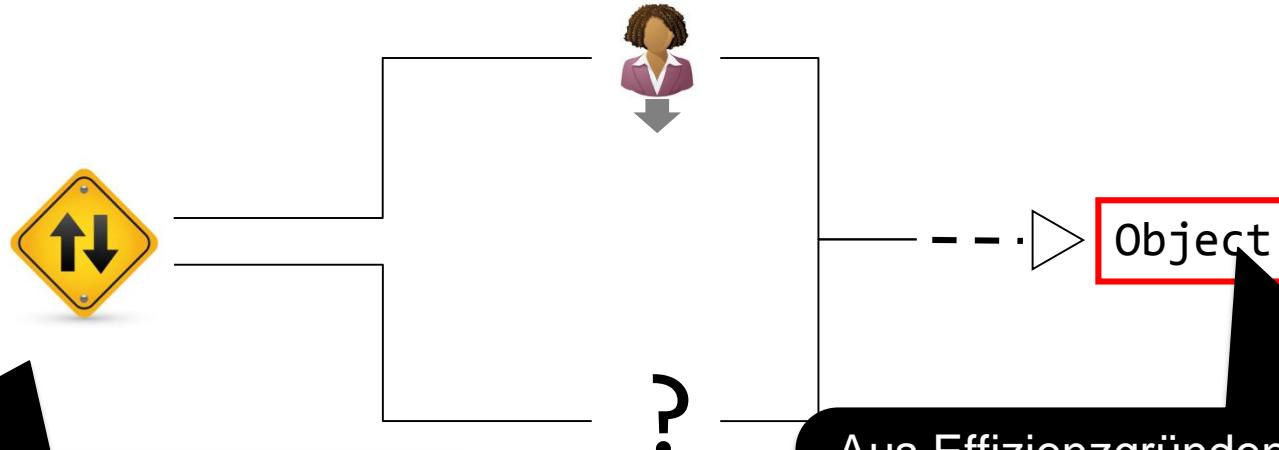


# MEHRFACHVERERBUNG

# Mehrfachvererbung: Historie



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



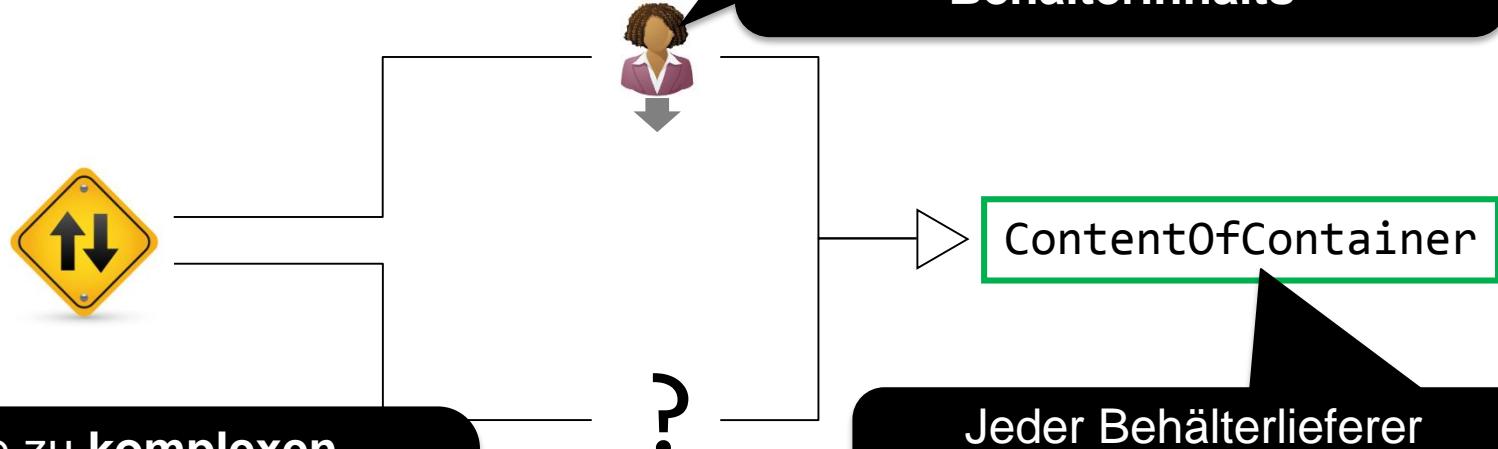
Wir können also nicht einfach „Objekte“ in den Aufzug laden

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

# Mehrfachvererbung: Nicht mehr so relevant!



Lösung mit Mehrfachvererbung:



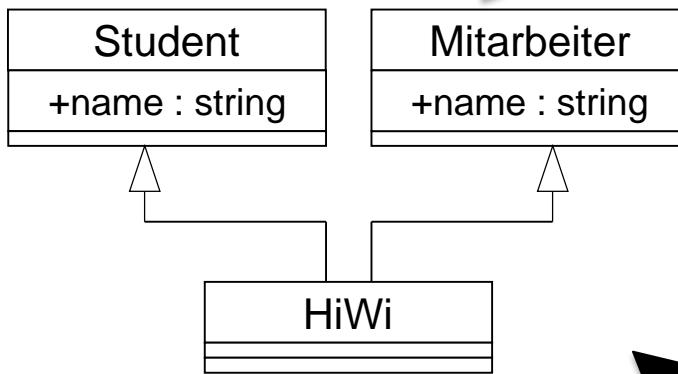
Führte zu **komplexen Vererbungshierarchien**, die mit Entwurfsentscheidungen nichts mehr zu tun hatten 😞

Mit Templates (siehe Tag 4) ist es jetzt hier möglich, auf **Mehrfachvererbung zu verzichten**

# Schnittstellen- vs. Implementierungsvererbung



**Schnittstellenvererbung:** Wenn weitere Oberklassen *pure virtual* sind (**enthalten nur pure virtual Methoden**), dann ist Mehrfachvererbung überhaupt kein Problem



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

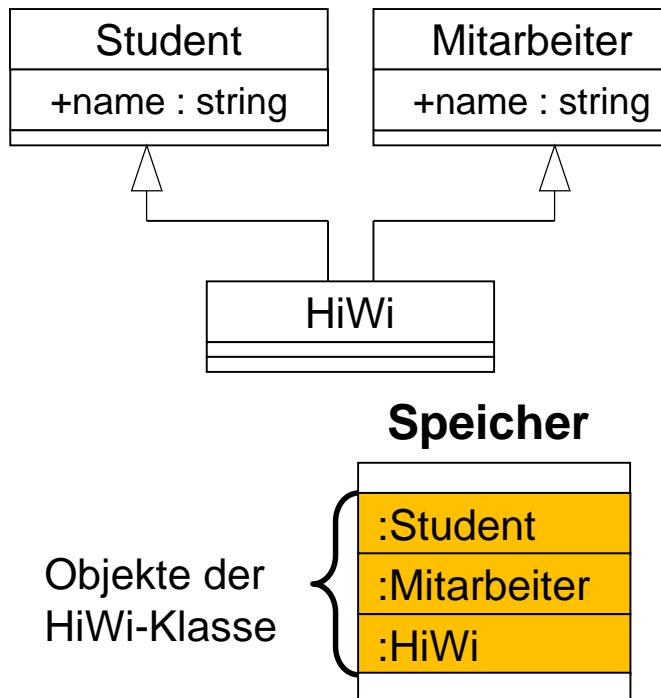
**Implementierungsvererbung:** Wird aber von mehreren Oberklassen wirklich **Implementierung** geerbt, so kann das zu Problemen führen...

# Implementierungsvererbung: Konflikte



## Mehrfachvererbung kann zu Mehrdeutigkeit führen

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



```
class Student { public: string name; };
class Mitarbeiter { public: string name };

class HiWi : public Student, public Mitarbeiter
{
    ...
}
```

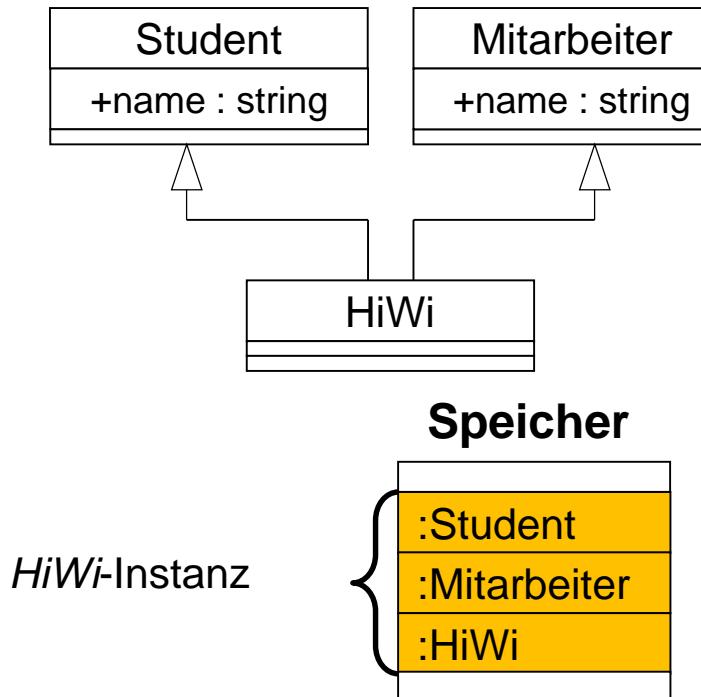
```
HiWi* h = new HiWi();
h->name = "Christian";
```

**Namenskonflikt!**  
Keine eindeutige  
Zuweisung ...

# Implementierungsvererbung: Konflikte



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



```
class Student { public: string name; };
class Mitarbeiter { public: string name };

class HiWi : public Student, public Mitarbeiter
{ ... }

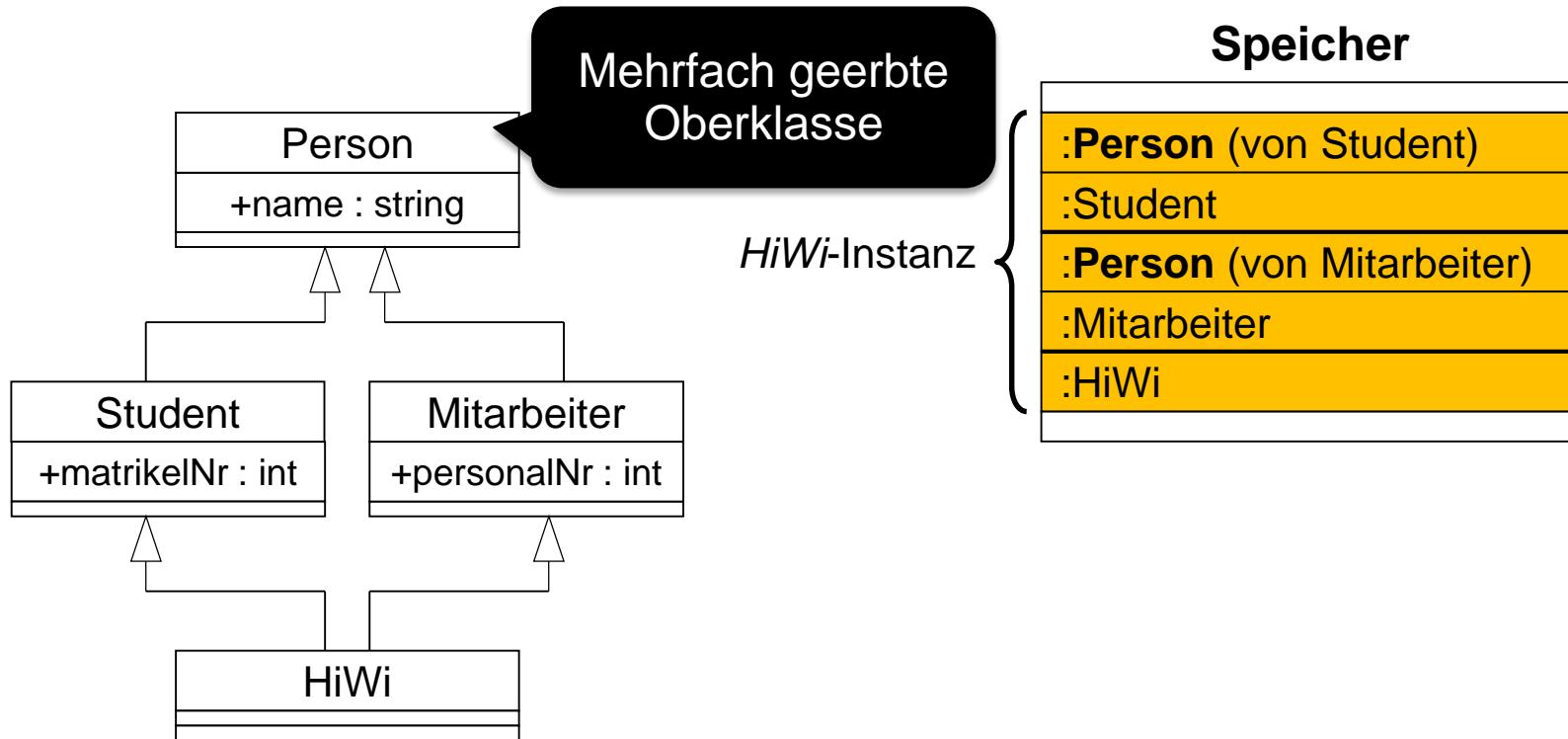
HiWi* h = new HiWi();
h->Student::name = "Christian";
h->Mitarbeiter::name = "Mark";
```

Scope-Operator

# Implementierungsvererbung: Speicherproblematik



Mehrfach geerbte Oberklassen führen auch zur unnötigen Bindung von Speicher

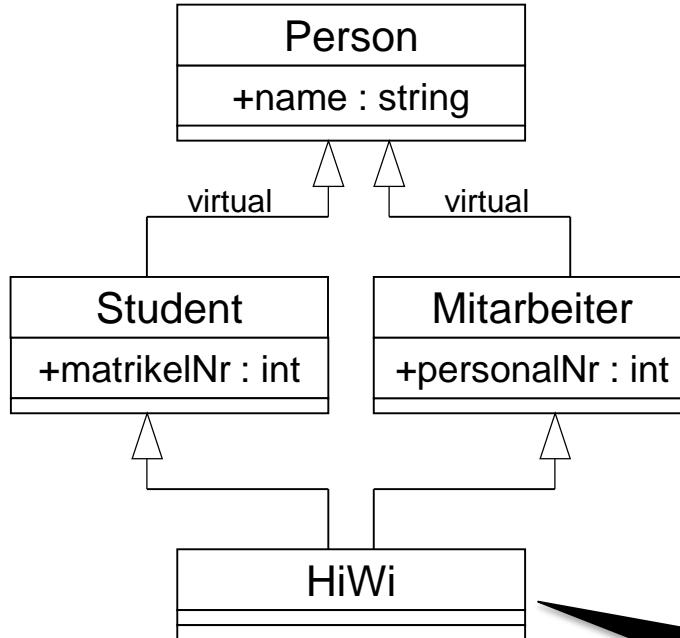


# Implementierungsvererb.: Speicherproblematik



## Lösung: Mehrfach geerbte Oberklassen nur einmal einbinden

Schlüsselwort **virtual** ermöglicht virtuelle Oberklassen / Vererbung



```
class Person { public: string name; };
class Student : virtual public Person { ... };
class Mitarbeiter : virtual public Person { ... };

class HiWi : public Student, public Mitarbeiter { ... }

HiWi* h1 = new HiWi();
H1->name = „Max“; // eindeutig (nur 1x vorhanden)
```

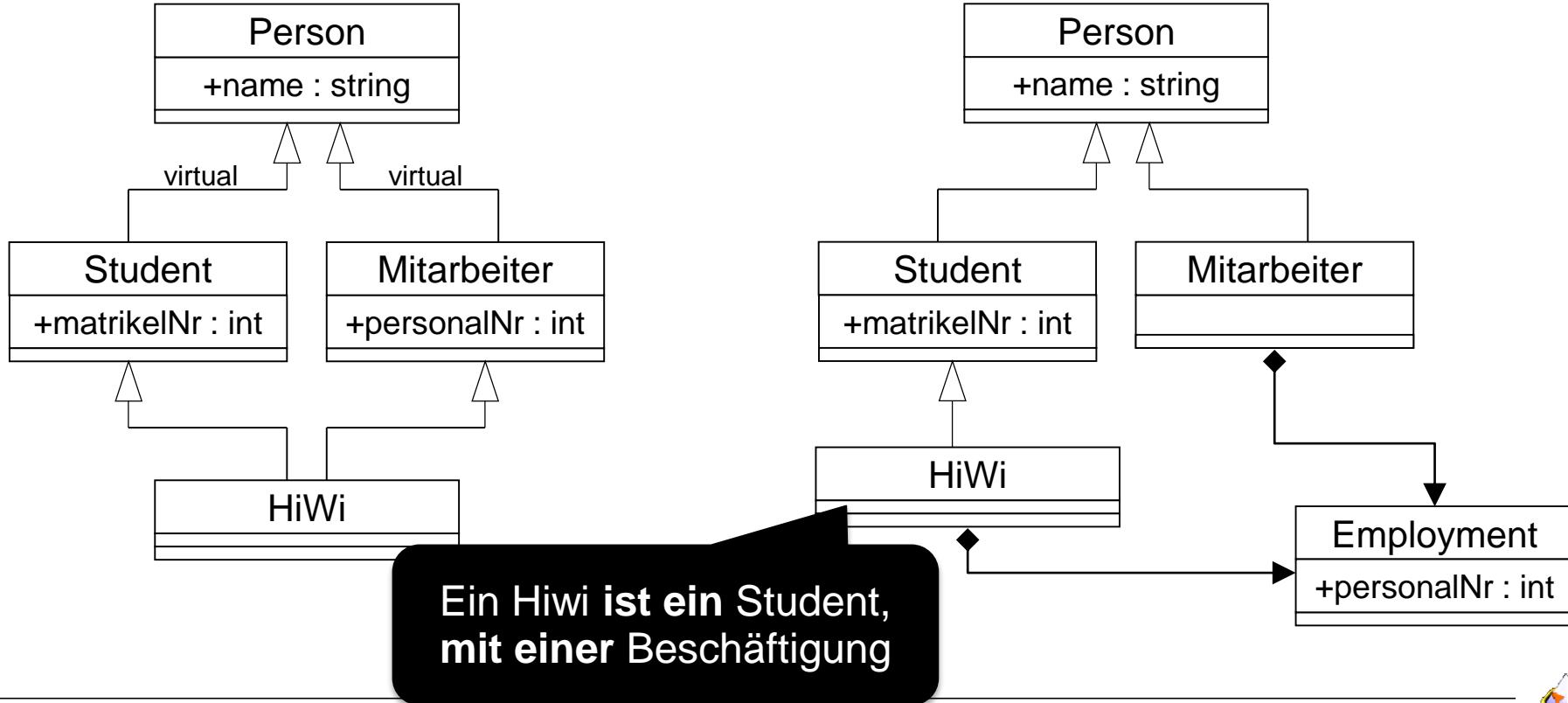
Aber: Die **virtual**-Deklaration findet nicht an der Stelle statt, an der sie nötig wird (*HiWi*)!

# Implementierungsvererbung: Schlechtes Design?



## Mehrfachvererbung kann auf „schlechtes“ Design hindeuten

Gemeinsamkeiten sollen explizit extrahiert bzw. das Design vereinfacht werden



# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

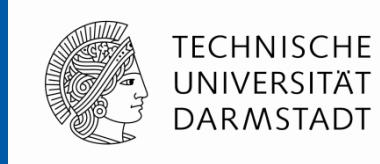
Also – Mehrfachvererbung: Ja oder nein?



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

# 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

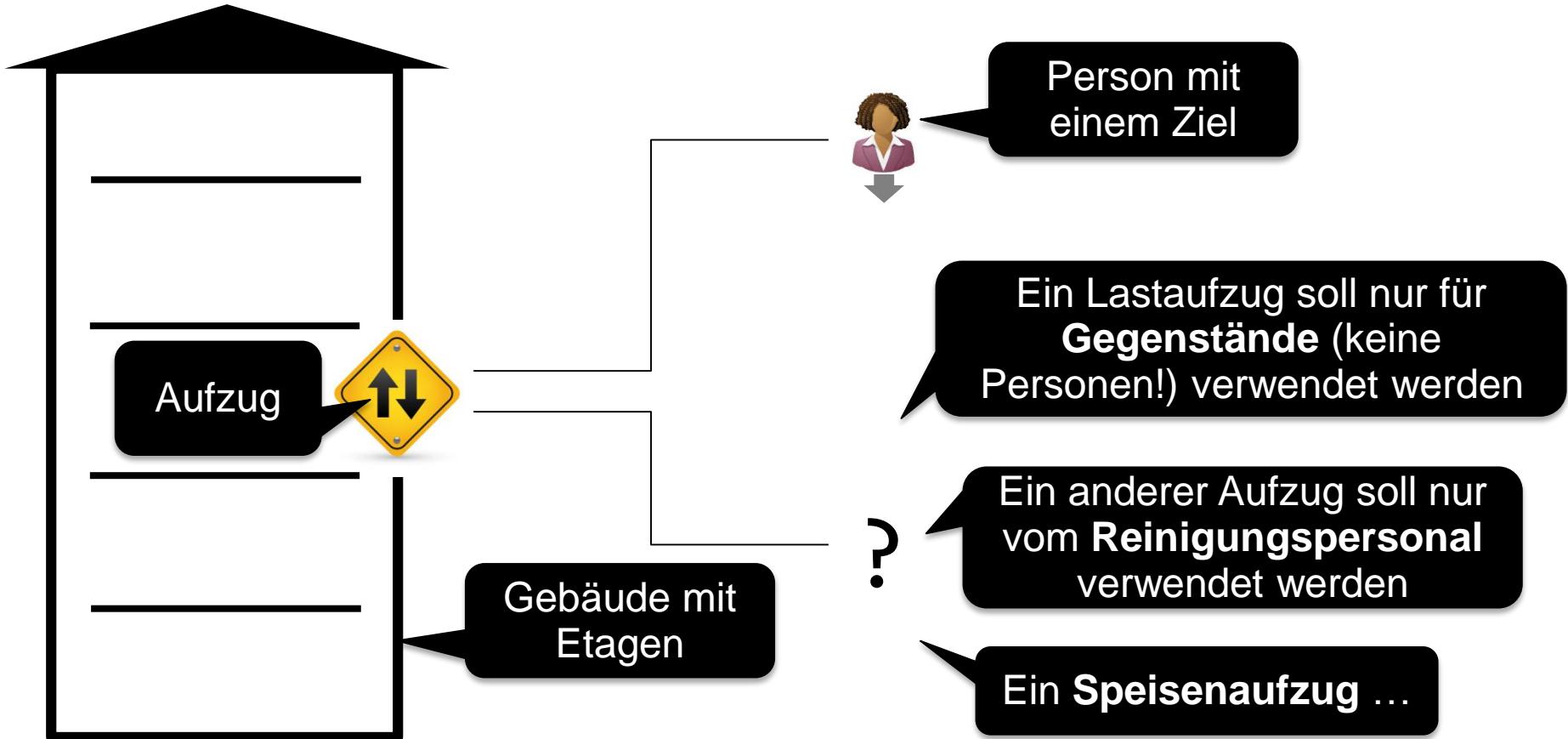
```
all: main.exe

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

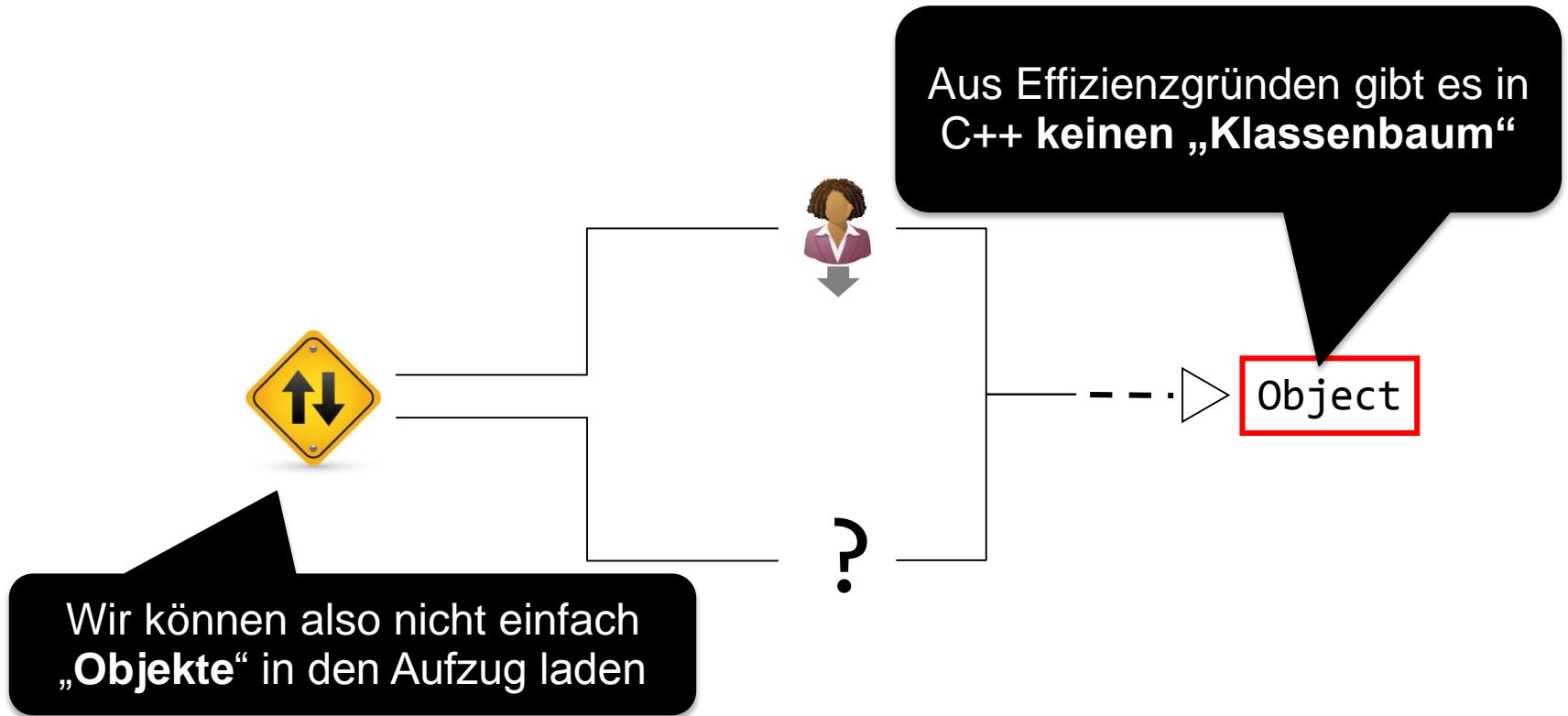


# TEMPLATES

# Templates: Motivation



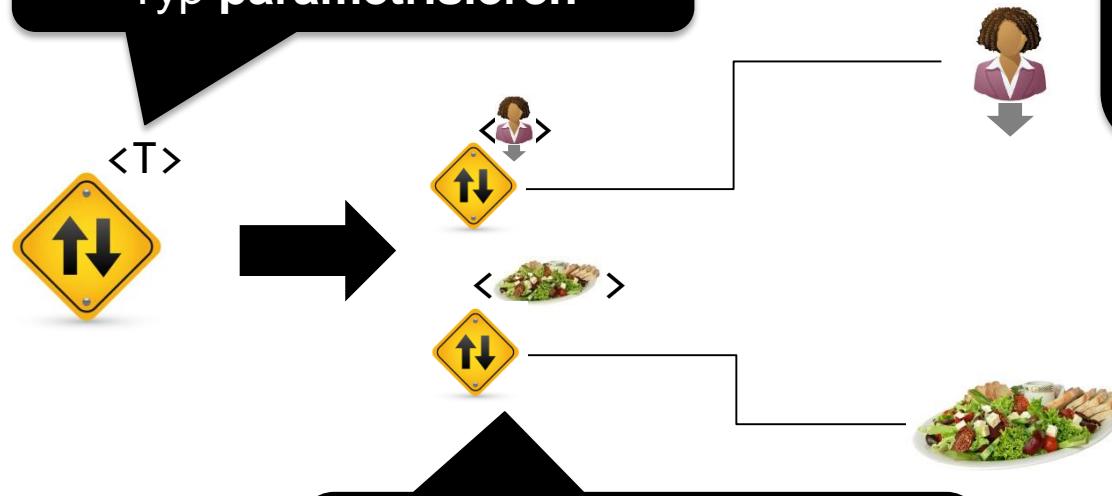
# Templates: Wieder mal das Containerproblem



# Templates: Idee



Implementierung mit einem  
Typ parametrisieren



Bei Bedarf wird die richtige  
Version der Implementierung  
zur Kompilierzeit generiert

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

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

# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Wieso ist „Object“ teuer?

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

Wie wird dieses „Problem“ in anderen Sprache  
gelöst?

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



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

# 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;  
};
```

Gewicht von Gerichten wird pauschal mit 1.5kg abgerundet

```
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;
```

Beachte die unterschiedlichen Rückgabetypen

# Class Templates: Syntax am Beispiel



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

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

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

    transportedObjects.push_back(object);
}

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

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.

# Function Templates: Syntax am Beispiel



Mehrere Typparameter möglich  
(auch bei Class Templates)

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

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

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

    return total;
}
```

Dies ist besonders für generische Algorithmen sehr nützlich

Typ kann genauso wie in einer Klasse frei verwendet werden

# Templates: Verwendung



Defaulttyp *Person* wird verwendet

```
int main(int argc, char **argv) {
Elevator<Person> 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 Nachteile und Vorteile dieser Art von „impliziten“ Schnittstellen?

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



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

# Mixins: Mehrfachvererbung trifft Templates



```
template<
```

```
    class Logger,  
    class Security,  
    class OperatingSystem,  
    class Platform
```

Mixins werden als  
Typparameter definiert

```
>
```

```
class System :
```

```
    public Logger,  
    public Security,  
    public OperatingSystem,  
    public Platform
```

```
{
```

```
};
```

Und „reingemischt“ mit  
Mehrfachvererbung!

# Mixins: Mehrfachvererbung trifft Templates



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

Benutzer kann eine konkrete Implementierung „zusammenmischen“

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

Und das Verhalten der Instanz wird dadurch flexibel **kombiniert** und **konfiguriert**



Wo sind eigentlich die Methoden *print* und *checkPassword* definiert?

# Wiederholung Mehrfachvererbung



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



# FUNKTIONSZEIGER UND FUNKTOREN

# Funktionszeiger: Beispiel

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

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

Ermöglicht kompakte, elegante, und sehr generische Algorithmen

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

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

Sogenannte **Callback-Funktionen** können Listener/Observer in Java komplett ersetzen

# Funktionszeiger: Beispiel



Auch Funktionen können  
Typparameter tragen.

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

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

Zeiger auf eine Funktion mit  
const string& Parameter

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

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

Verwendung wie ein  
normaler Funktionsaufruf

# Funktionszeiger: Syntax



Typ des Rückgabewerts

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

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

**Zeigertyp**, Klammern sind  
notwendig um Rückgabetyp und  
Zeiger auseinanderzuhalten

**Adresse der Funktion** (hier  
durch Instanzierung eines  
Funktion-Templates)

# Funktionsobjekte und Templates



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

Syntax soll hier identisch bleiben, obwohl wir eine Methode aufrufen

```
class ConsoleLogger {
public:
    ConsoleLogger();
    ~ConsoleLogger();
```

Dafür muss man nur **operator()** überladen

```
inline void operator()(int i) const {
    std::cout << "user:~ /$ " << i << std::endl;
}
};
```

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

```
int n[] = {-1, 20, 33, 120};
applyToSequence(ConsoleLogger(), n, n + 4);
```

# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Wieso sind Zeiger auf Funktionen nützlich?

Gibt es auch Nachteile?

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



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

# Zeiger auf Funktionen: Fazit



Zeiger auf Funktionen ermöglichen einen **eher funktionalen Programmierstil** (ideal für generische Algorithmen)

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 **Funktoren** sinnvoll.



# STANDARD-BIBLIOTHEKEN IN C++

# Standard-Bibliotheken in C++



Viele Funktionen für  
Stringmanipulation

`<string>`

TODO: besser darstellen

Flexible, erweiterbare IO

`<iostream>`

Standard Template Library (STL):

Generische Algorithmen

Wir schauen uns `copy`  
und `remove_copy_if`  
als Beispiel an

Generische Behälter

(Boost)

Nicht offiziell -  
Viele erweiterte  
Funktionalitäten

Wir schauen uns  
`priority_queue`  
als Beispiel an

# Boost: „Brutschränk“ für C++-Standardkomponenten



<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

# Generische STL-Algorithmen: `copy`



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

muss `++`, `*`, `==`, und `!=` unterstützen

muss `++` und `*` unterstützen

## Parameters:

### `first, last`

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



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

**InputIterator:** müssen `++`, `*`, `==`, und `!=` unterstützen

**OutputIterator:** müssen `++` und `*` unterstützen

Wieso ist diese Forderung notwendig?



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

# Generische STL-Algorithmen: `copy`



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

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

using namespace std;

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

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

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

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: *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:

[...]  
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: *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 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: *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**  
(vector wird als Default verwendet)

Damit Compiler weiß, dass *value\_type* ein Typ ist

**Binäres Prädikat** (*less* wird als Default verwendet)

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: *priority\_queue*



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

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

```
template<class T>  
void process_queue(T& queue){  
    while(!queue.empty()){

        cout << queue.top()
            << ",";  

        queue.pop();
    }
}
```

Einfache Hilfsfunktion  
für die Ausgabe

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

Standard Funktionsobjekt

```
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

Ist das hier wirklich lesbarer als eine Schleife?

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

Was ist daran „schön“ oder zumindest praktisch?



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

# Standard Template Library: Fazit



**Mächtig, effizient, ausgereift und gut dokumentiert**

Steile Lernkurve (erfordert Wissen über Templates, Funktoren, Iteratoren, Mixins, ...)

**Boost** als mächtiger „Brutkasten“ für die nächsten Standards

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



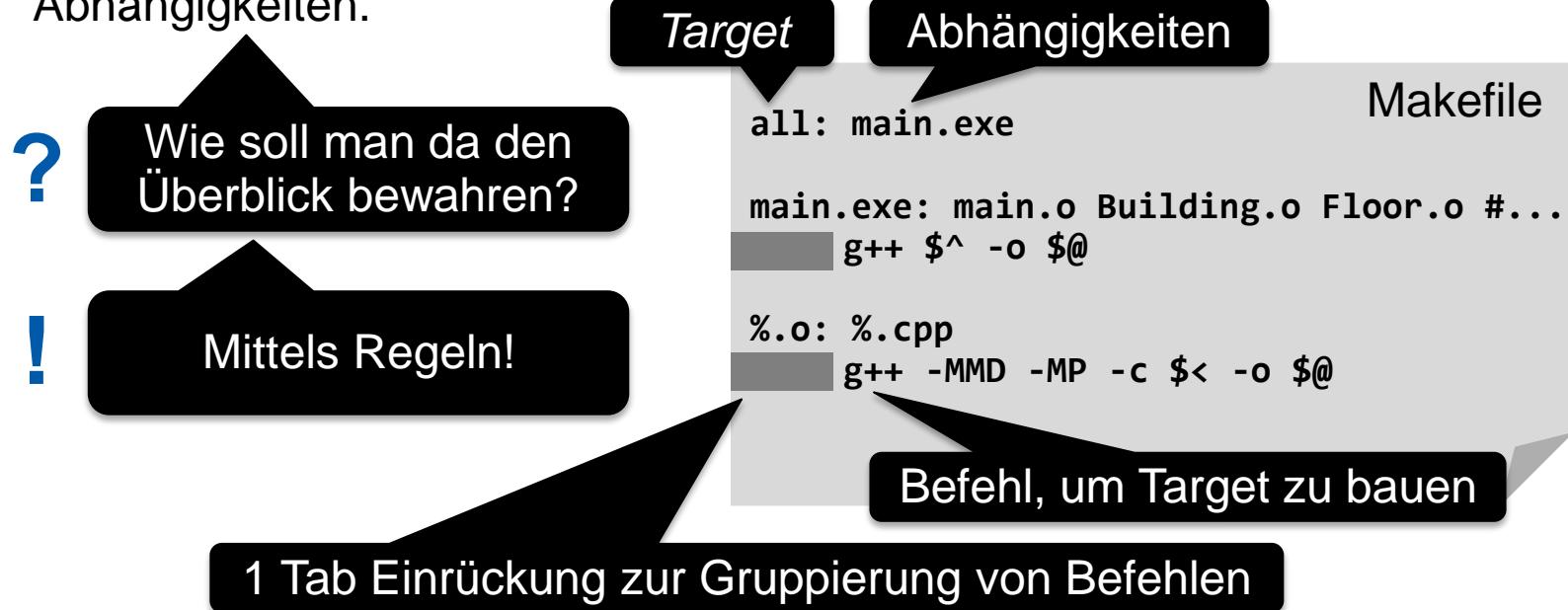
TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# MAKEFILES

# Makefiles: Motivation



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



# 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*.

Erste Regel ist immer der Default-Einstiegspunkt. Eclipse will *all*.

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

„Suffixregel“; \$< - Input; \$@ - output

Administrative 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
-invoke $(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:

- *cmake, qmake*: Generatoren für Makefiles (letzterer von Qt)
- *Ant, Maven, Ivy, Gradle*: ... eher für Java gedacht

# Look down!

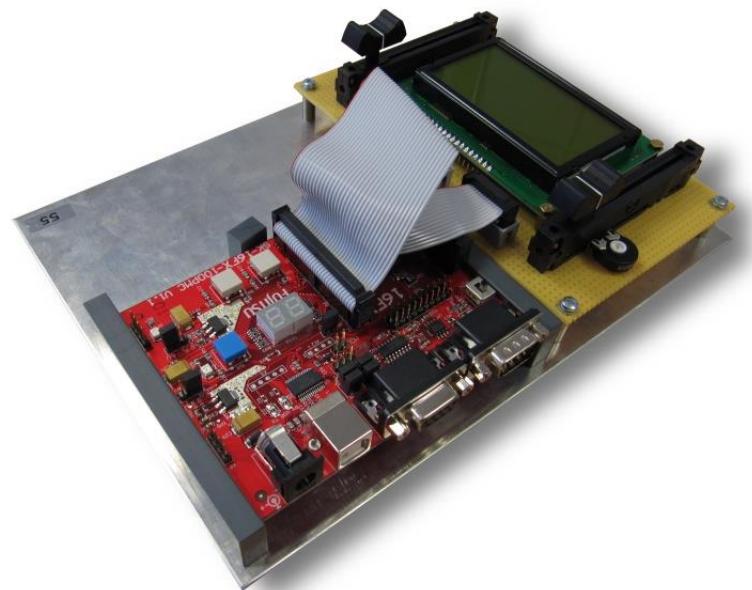
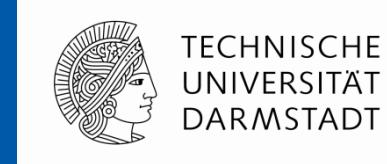


TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Nützliche Kommentare  
finden sich  
auch in den Notizen!

# Programmierpraktikum C und C++

## C für Microcontroller – Einführung



ES Real-Time Systems Lab

Prof. Dr. rer. nat. Andy Schürr

Dept. of Electrical Engineering and Information Technology

Dept. of Computer Science (adjunct Professor)

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

**Roland Kluge**

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

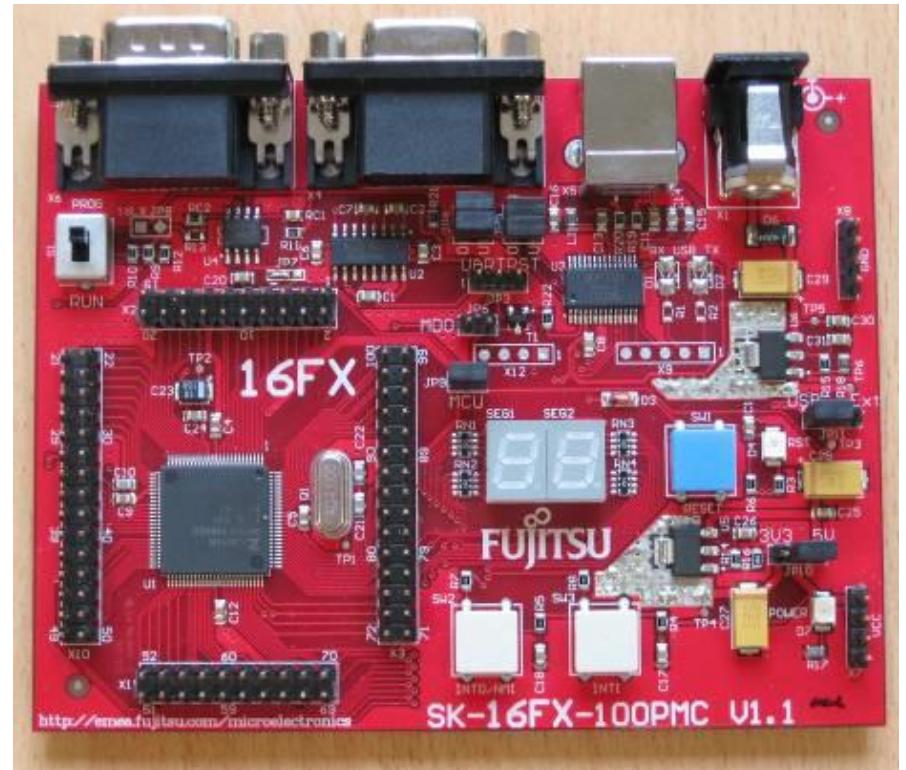
# Entwicklungsboard

# **MB96F348HSB Mikrocontroller**

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

# **Starterkit SK-16FX-EUROscope**

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

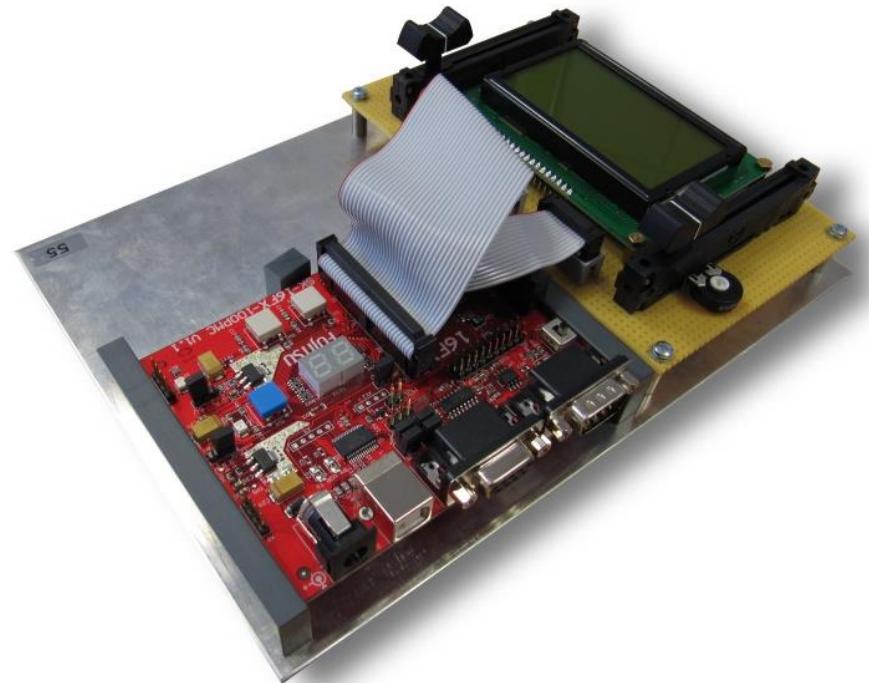


# Erweiterungen gegenüber der Standardausführung



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- LC-Display
  - AV128641 von Anag Vision
  - Vollgraphisch
  - 128 x 64 Pixel
  - hintergrundbeleuchtet
- Zwei Schiebepotentiometer





- Von Fujitsu Microelectronics Ltd.
- Unterstützt nur **ANSI C90**,
  - zusätzlich auch einzeilige // Kommentare
  - Variablen-deklaration 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“



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Compiler kann nicht wissen, welche Komponenten angeschlossen sind
- Es gibt **keine Ausgabe über printf()**
  - Alternative: 7-Segment-Anzeige, LCD(, LEDs)
- 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

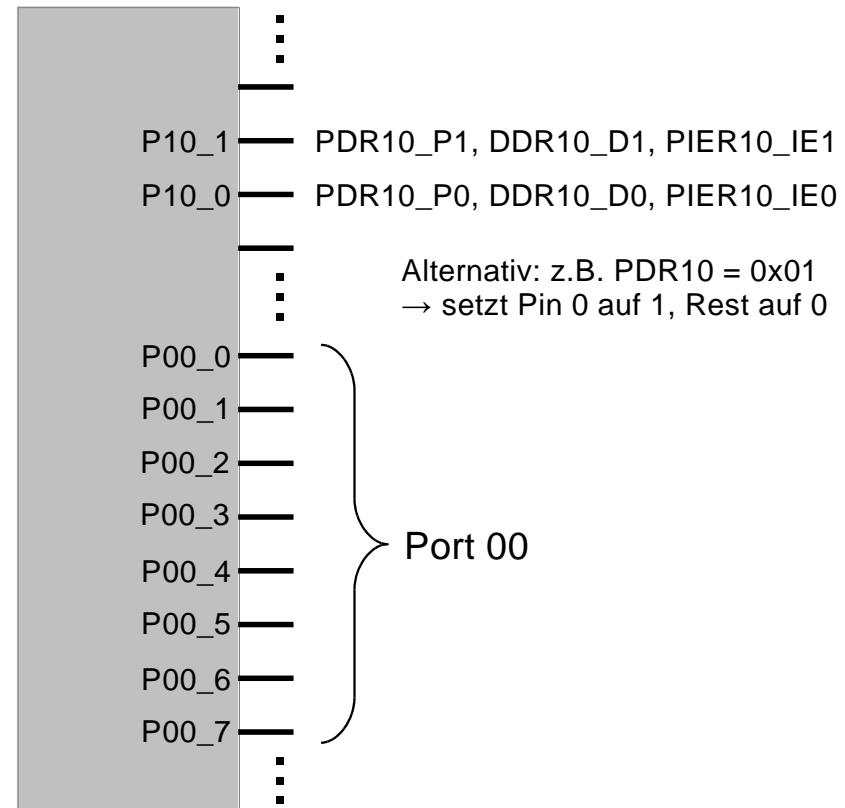
- Einzelne digitale Ein/Ausgänge
- Analoge Eingänge
- Schnittstellen, z.B.
  - UART (serielle Schnittstelle)
  - CAN (serieller Bus)



## 8 Pins = Port

Je Pin mehrere Register, u.a.:

- **Port-Data-Register (PDR)**
  - Eingang: Abfrage des Zustandes
  - Ausgang: Setzen des Pegels
- **Data-Direction-Register (DDR)**
  - Setzen auf Eingang oder Ausgang
  - 0 → Eingang, 1 → Ausgang
- **Port-Input-Enable-Register (PIER)**
  - Bei Eingangspin den Eingang aktiv schalten



# Beispielcode: Pins abfragen



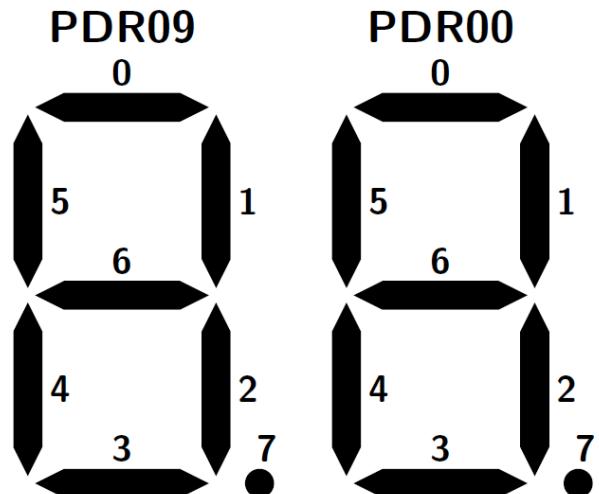
```
/* Beispiel: Pins als Eingang */
char status;
DDR07_D0 = 0;           // Pin 0 von Port 07 als Input
PIER07_I0 = 1;          // Pin 0 von Port 07 als Eingang aktiv

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
```