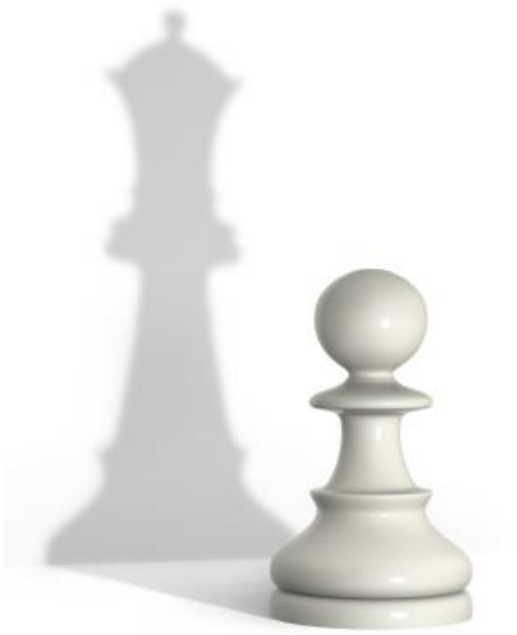
Programmierpraktikum C und C++

Fortgeschrittene Themen

TECHNISCHE UNIVERSITÄT DARMSTADT



Anthony Anjorin

anthony.anjorin@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)

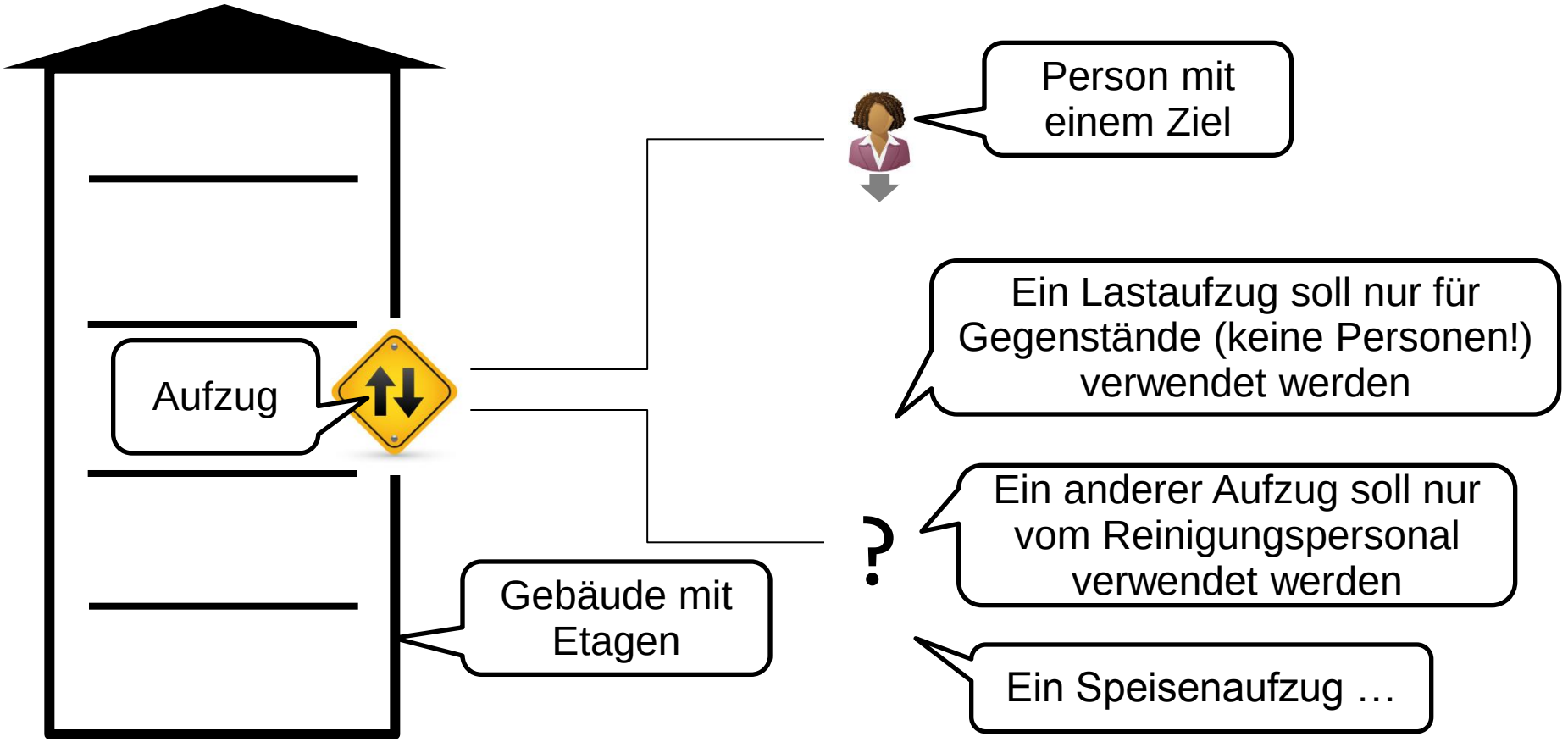
www.es.tu-darmstadt.de

Agenda

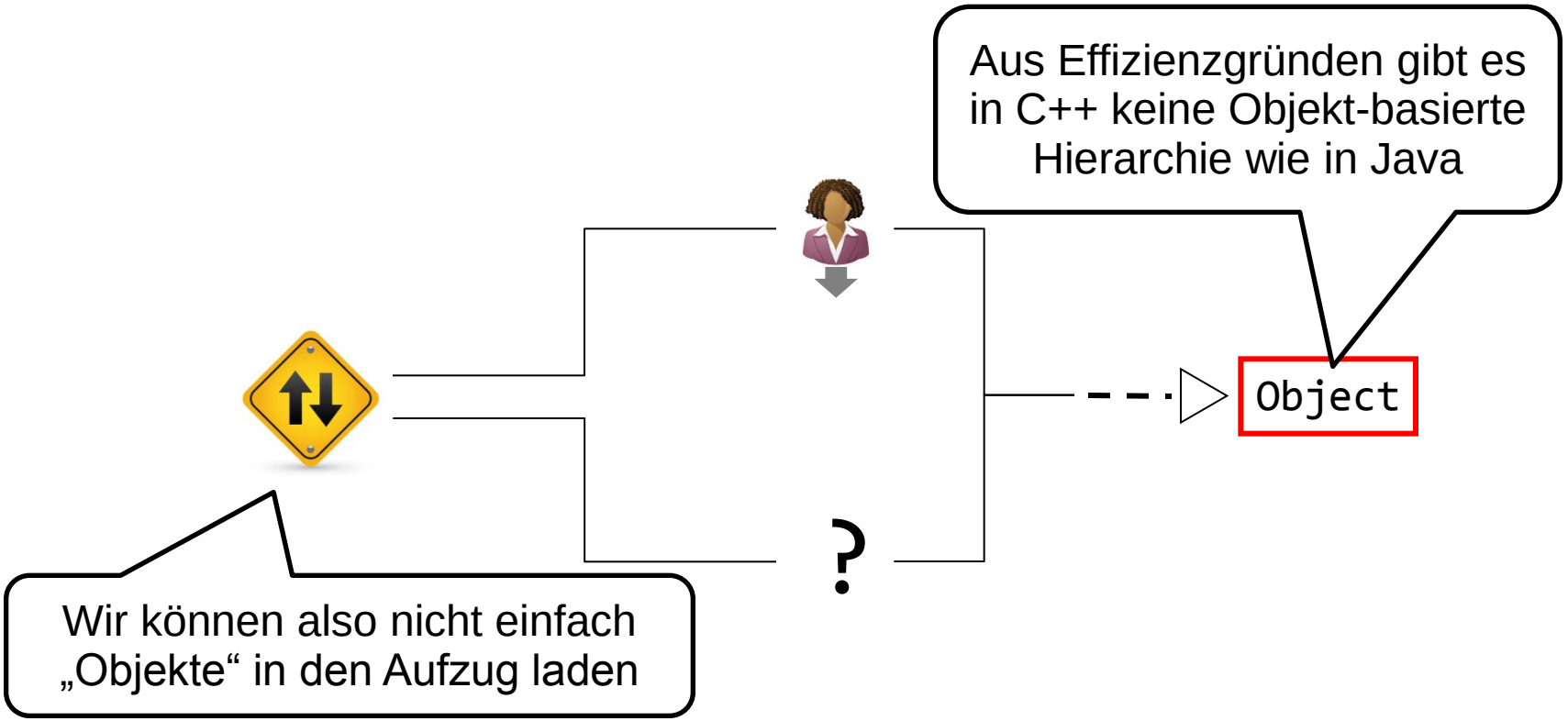
TECHNISCHE UNIVERSITÄT DARMSTADT

1. Templates
2. Mehrfachvererbung
3. Zeiger auf Funktionen, Methoden und Funktionsobjekte
4. Überblick der Standard C++ Library

Templates: Motivation

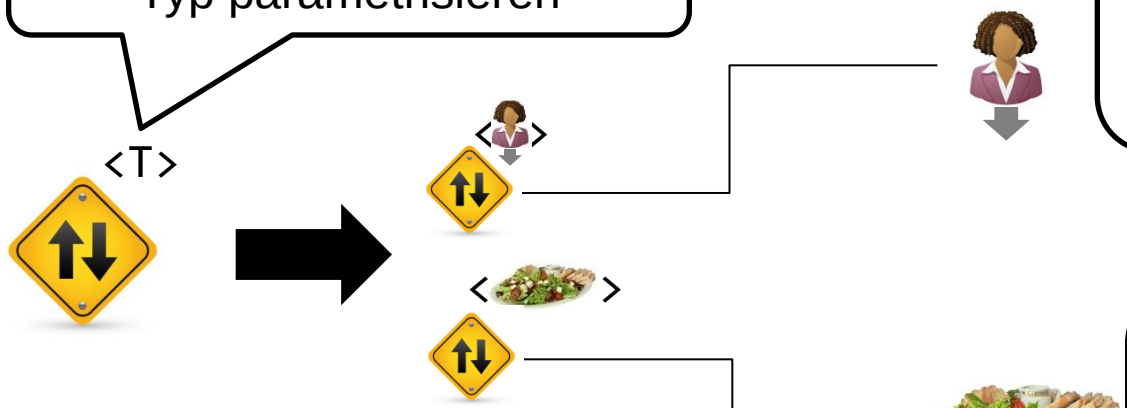


Templates: Motivation



Templates: Idee

Implementierung mit einem Typ parametrisieren



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

Bei Bedarf wird die richtige Version der Implementierung zur Kompilierzeit generiert

Intermezzo

TECHNISCHE UNIVERSITÄT DARMSTADT

Wieso ist „Object“ teuer?

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

Wie wird dieses „Problem“ in einer Sprache wie C gelöst?

Was ist mit Sprachen wie Scheme/Haskell/Python/Ruby?

Class Templates: Syntax am Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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 st          e;
};
```

Gewicht von Gerichten wird pauschal mit 1.5kg abgerundet

Implementierungsdateien sind einfach …

Beachte die unterschiedlichen Rückgabetypen

Class Templates: Syntax am Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

T wird deklariert als Typparameter.
Es ist möglich Defaultwerte zu vergeben.

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

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

      transportedObjects.push_back(object);
}

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

T wird nach Bedarf einfach verwendet
(hier werden Methoden aufgerufen)

Erst bei der Expansion des Templates wird sich herausstellen, ob der Typparameter wirklich diese Methoden hat oder nicht

Typischerweise werden Template-Klassen nicht in Header- und Impl-Dateien getrennt

Function Templates: Syntax am Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Default Typparameter wird verwendet

```
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

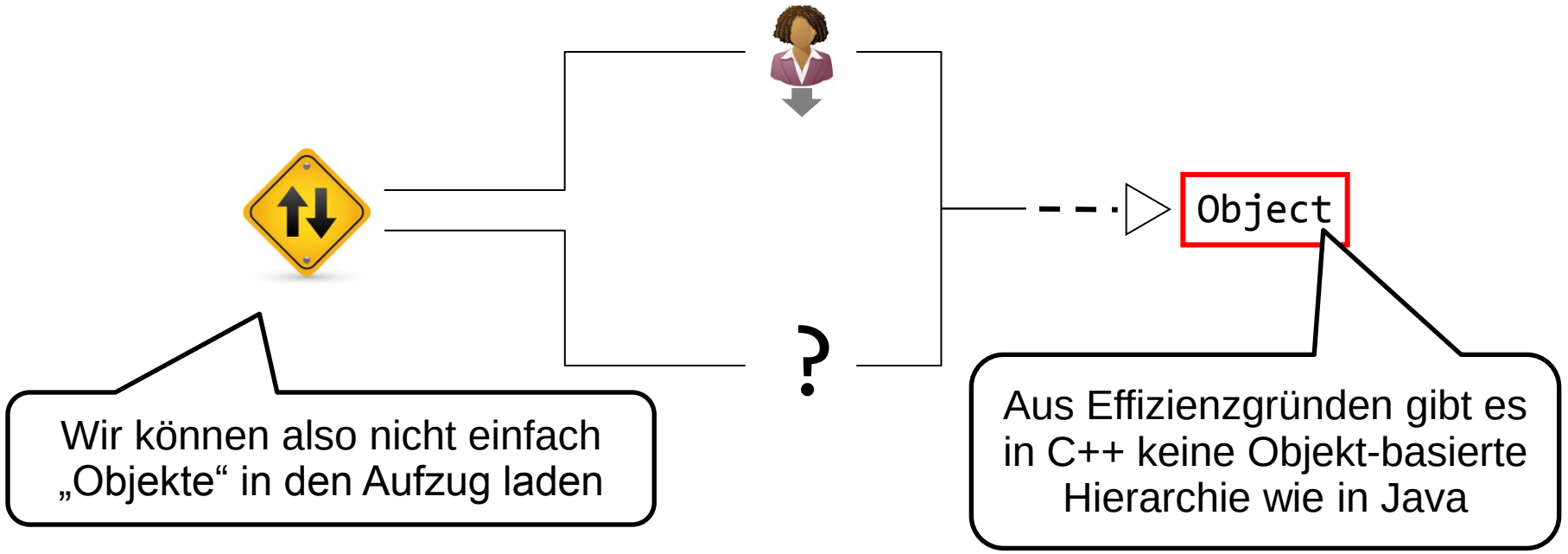
Was ist genau damit gemeint, dass Templates eine Schnittstelle induzieren?

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

Mehrfachvererbung: Historie

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ursprünglich als Lösung für Containerproblem (bevor es Templates gab)



Mehrfachvererbung: Nicht mehr so relevant!

TECHNISCHE UNIVERSITÄT DARMSTADT

- Lösung mit Mehrfachvererbung:

?

`ContentOfContainer`

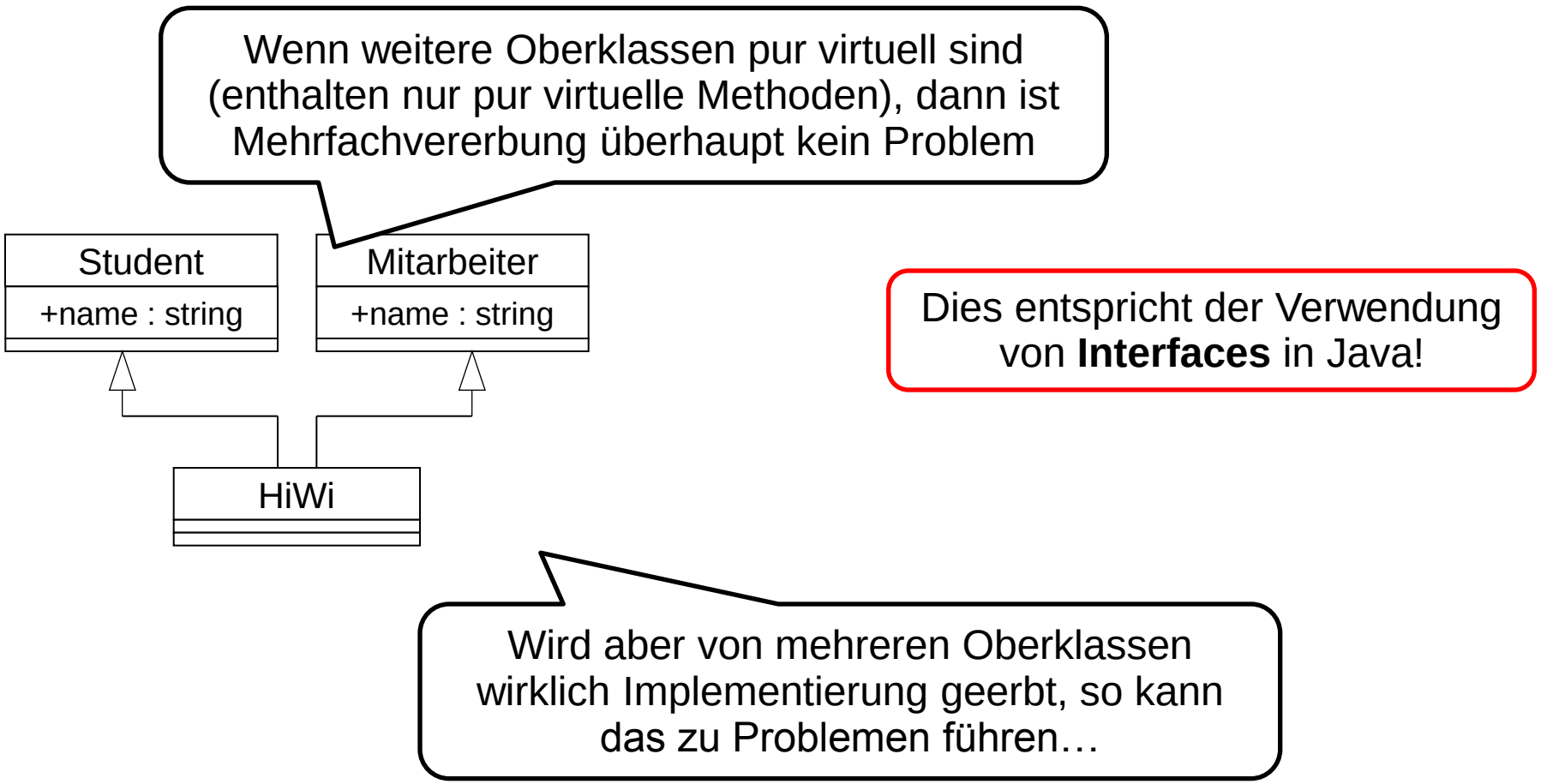
Jeder Typ, der in den Behälter soll, erbt einfach zusätzlich von diesem Typ des Behälterinhalts

Führte zu unnötig komplexen Vererbungshierarchien, die mit Entwurfsentscheidungen nichts mehr zu tun hatten ☹

Jeder Behälterlieferer definiert einen Typ, den der Behälter enthalten kann

Mit Templates ist es jetzt möglich, komplett auf Mehrfachvererbung zu verzichten

Mehrfachvererbung: Schnittstellenvererbung

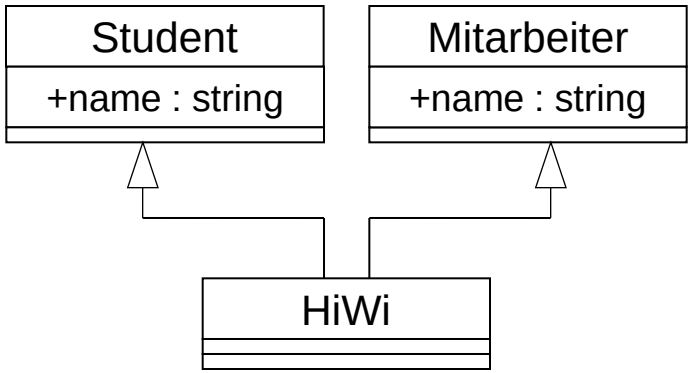


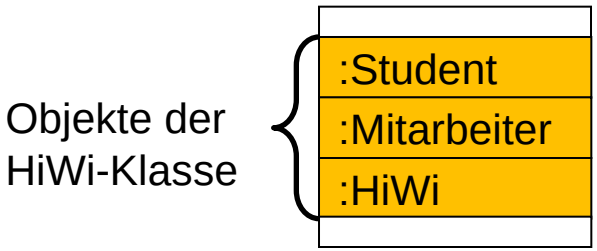
Implementierungsvererbung: Konflikte

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Mehrfachvererbung kann zu Mehrdeutigkeit führen**

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



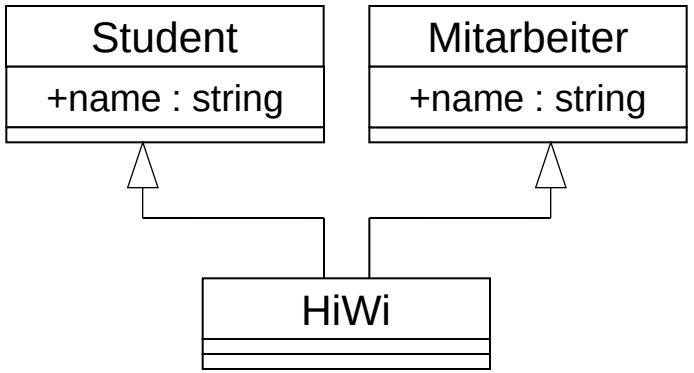


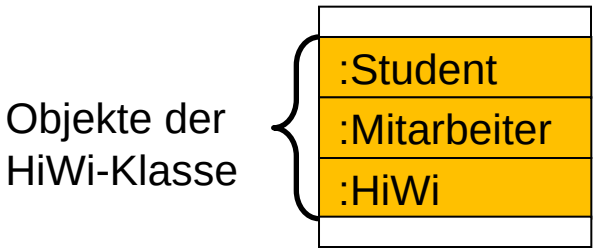

```
class Student { public: string name; };
class Mitarbeiter { public: string name };

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)**



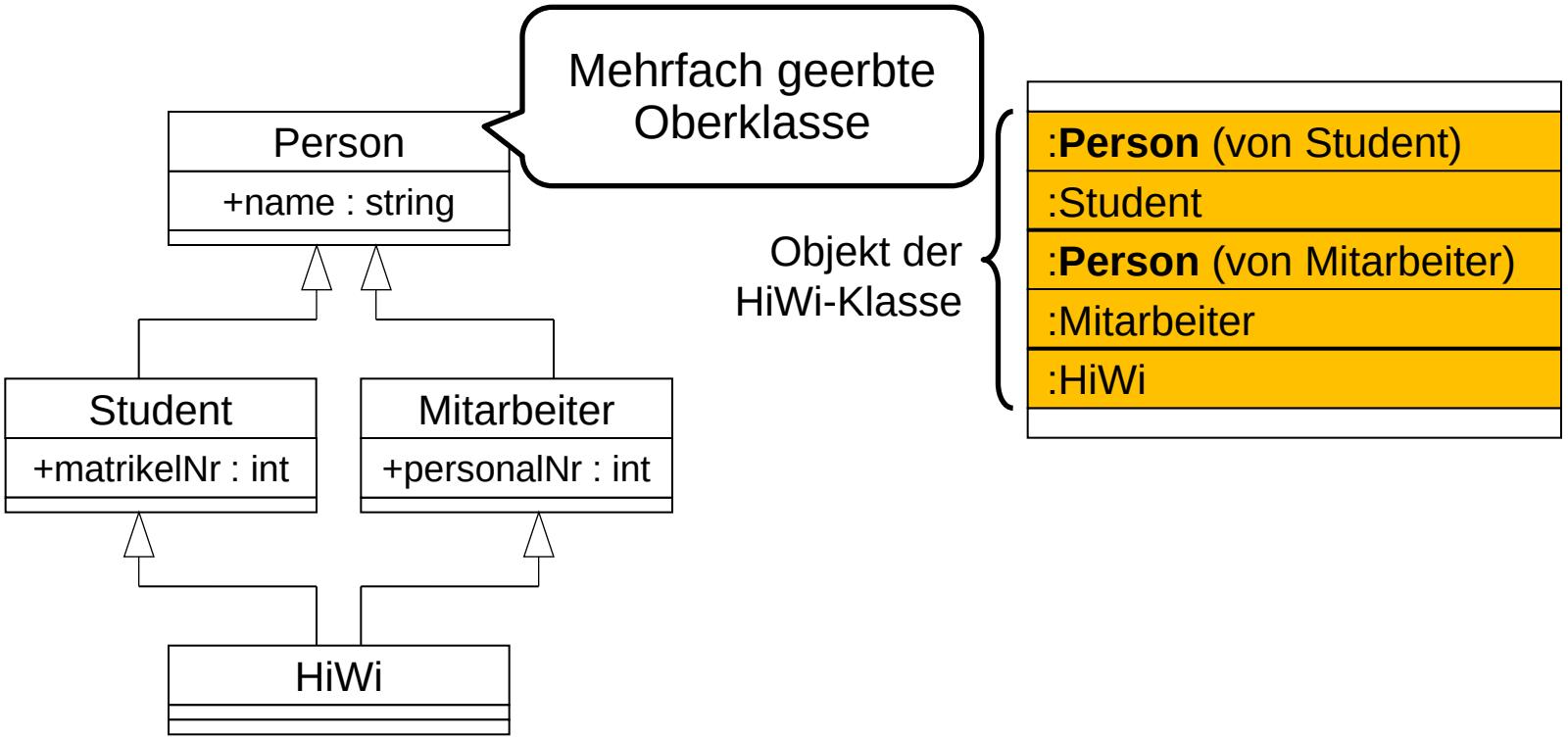



```
class Student { public: string name; };
class Mitarbeiter { public: string name };

class HiWi : public Student, public Mitarbeiter
{ … }

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

Scope-Operator

Implementierungsvererb.: Speicherproblematik

- **Mehrfach geerbte Oberklassen führen auch zur unnötigen Bindung von Speicher**

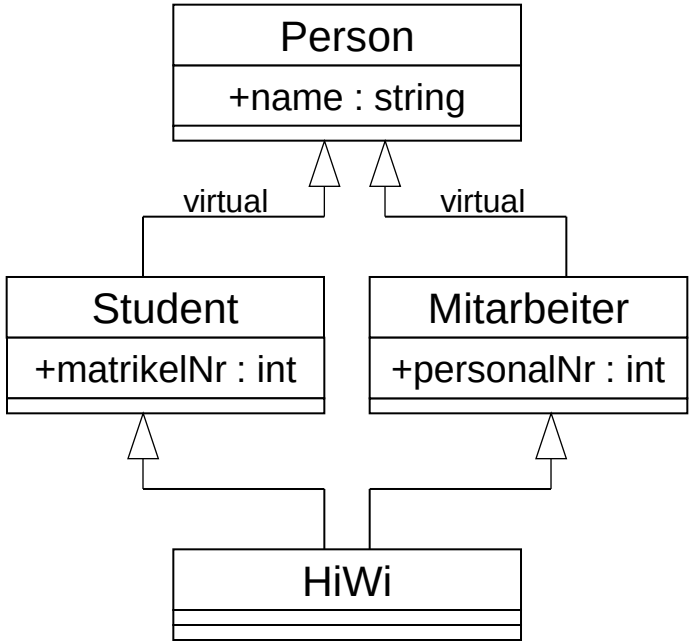


Implementierungsvererb.: Speicherproblematik

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Lösung: Mehrfach geerbte Oberklassen nur einmal einbinden**

  Schlüsselwort *virtual* ermöglicht virtuelle Oberklassen / Vererbung



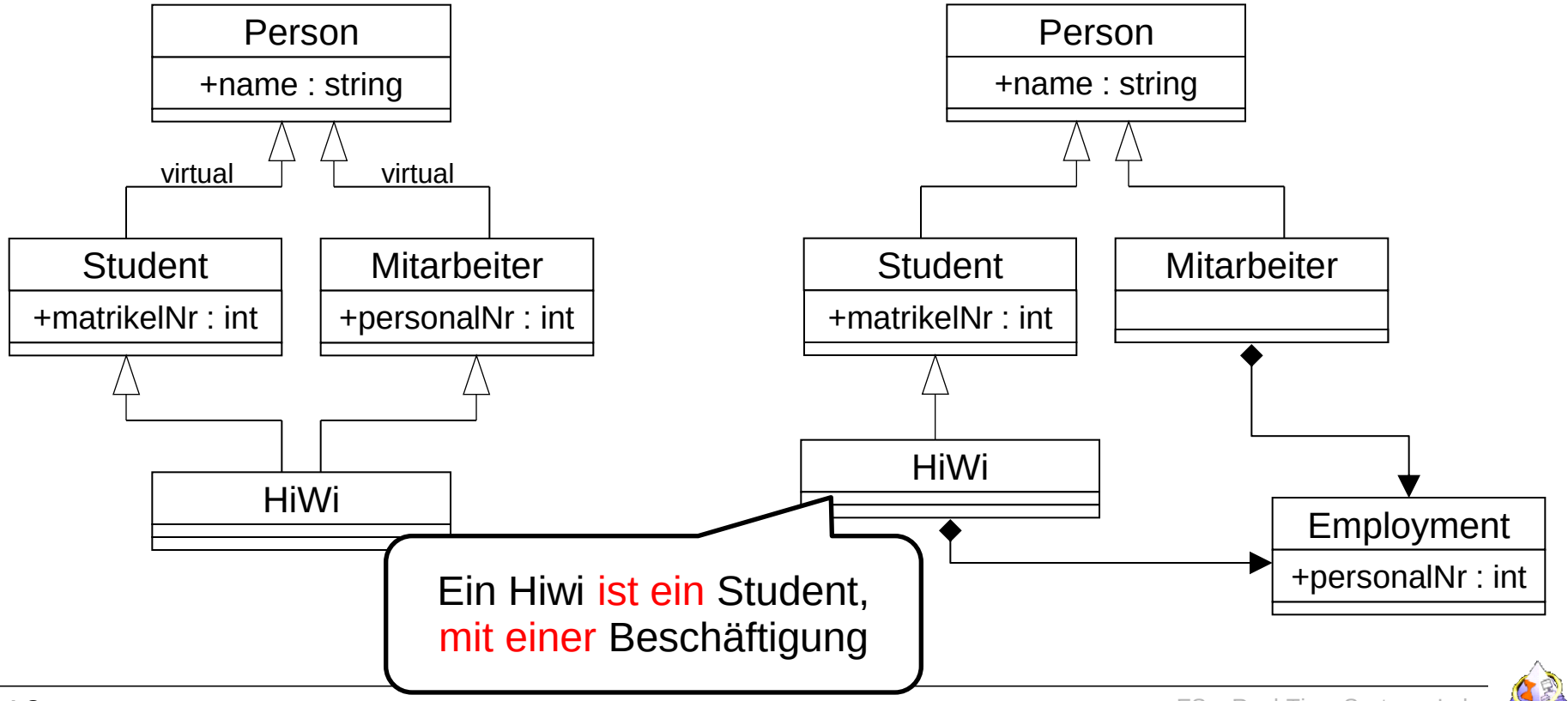

```
class Person { public: string name; };
class Student : virtual public Person { … };
class Mitarbeiter : virtual public Person { … };

class HiWi : public Student, public Mitarbeiter { … }

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

Implementierungsvererb.: Schlechtes Design?

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Mehrfachvererbung kann auf „schlechtes“ Design hindeuten**

  Gemeinsamkeiten sollen explizit extrahiert bzw. das Design vereinfacht werden



Mehrfachvererbung: Mixins

TECHNISCHE UNIVERSITÄT DARMSTADT

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

};
```

Mixins werden als Typparameter definiert

Und reingemischt mit Mehrfachvererbung!

Mehrfachvererbung: Mixins

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Intermezzo

TECHNISCHE UNIVERSITÄT DARMSTADT

Also – Mehrfachvererbung: Ja oder nein?

Mehrfachvererbung: Ja oder Nein?

TECHNISCHE UNIVERSITÄT DARMSTADT

1. Schnittstellenvererbung sinnvoll und nützlich (Design!)

2. Implementierungsvererbung problematisch und zu vermeiden (Komposition vorziehen)

3. Mixins durchaus sinnvoll (eigentlich eine Art Komposition)

Zeiger auf Funktionen: Motivation

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Zeiger auf Funktionen: Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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;
     }
}
```

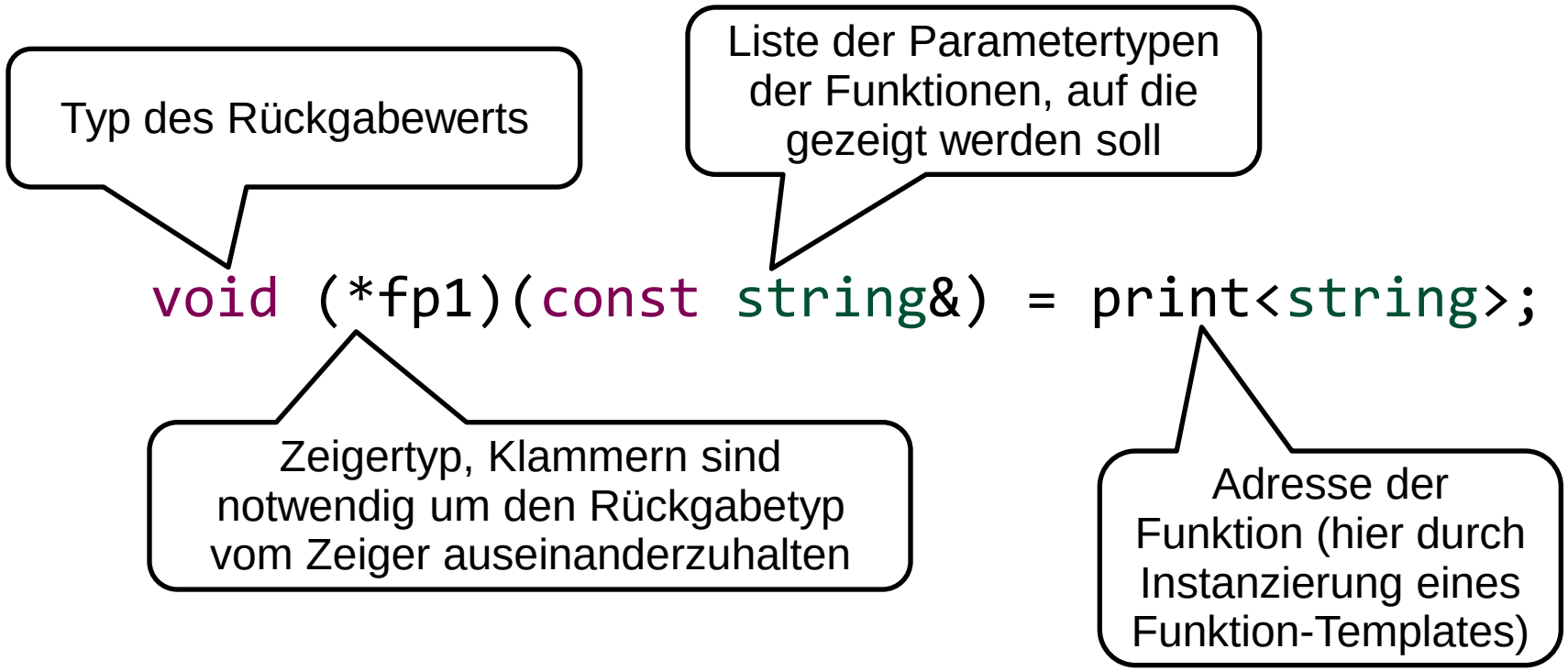
Zwei Funktionen, eine mit einem Typparameter

Zeiger auf eine Funktion

```
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

Zeiger auf Funktionen: Syntax



Zeiger auf Methoden: Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

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

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

Normale Methode einer Klasse

Zeiger auf eine Methode

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

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

Aufruf nur mit einer Instanz der Klasse

Beim Zeiger auf Methoden muss die Klasse als „Scope“ angegeben werden

Zeiger auf Funktionen vs. Zeiger auf Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

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

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

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

Das würde so nicht gehen, da die Instanz fehlt, dessen Methode aufgerufen wird

Funktionsobjekte und Templates

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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();

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

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

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

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

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?

Zeiger auf Funktionen: Fazit

- Zeiger auf Funktionen ermöglichen einen eher funktionalen Programmierstil (ideal für generische Algorithmen höherer Ordnung)
- In Verbindung mit Templates entsteht typischerweise ein schlankeres, kompakteres Design als in Java (reine OO)
- Ideal für kleine Funktionen, um eine Explosion an kleinen Klassen (z.B. mit jeweils nur einer Methode und ohne Zustand) zu vermeiden
- Syntax und Fehlermeldungen vom Compiler sind aber recht gewöhnungsbedürftig!

Überblick der Standard C++ Library

Strings

Viele Funktionen für Stringmanipulation

IOstreams

Flexible erweiterbare IO

Die Standard Template Library (STL):

Generische Algorithmen

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

Generische Behälter

Wir schauen uns `priority_queue` als Beispiel an

Generische STL-Algorithmen: copy

TECHNISCHE UNIVERSITÄT DARMSTADT

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

müssen ++ und * unterstützen

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

müssen ++, *, ==, 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.

Intermezzo

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

Wieso ist diese Forderung/Konvention sinnvoll?

Generische STL-Algorithmen: copy

TECHNISCHE UNIVERSITÄT DARMSTADT

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

STL-Behälter bieten InputIteratoren an

Erzeugt einen OutputIterator aus einem Stream

Generische STL-Algorithmen: remove_copy_if

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

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

Zusätzlich mit einem Prädikat filtern

```
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
}
```

Funkionszeiger oder Funktionsobjekt übergeben

Generische Behälter: priority_queue

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Typ vom Inhalt der Warteschlange

Typ des darunterliegenden Behälters (vector wird als Default verwendet)

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

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!

Generische Behälter: priority_queue

TECHNISCHE UNIVERSITÄT DARMSTADT

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

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

```
#include <iostream>
#include <queue>
#include <functional>

using namespace std;

template<class T>
void process_queue(T& queue){
     while(!queue.empty()){
          cout  << queue.top()
                << ",";
          queue.pop();
     }
}
```

Einfache Hilfsfunktion für die Ausgabe

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

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


     priority_queue<   int,
                       vector<int>,
                       greater<int> >
     ascending(numbers, numbers + 5);
     process_queue(ascending); // 1,2,3,4,5
}
```

Standard Funktionsobjekt

Intermezzo

TECHNISCHE UNIVERSITÄT DARMSTADT

Ist das hier wirklich lesbarer als eine Schleife?

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

Was ist denn daran „schön“?

Was ist der Vorteil von intelligenten Behältern?

Standard C++ Library: Fazit

TECHNISCHE UNIVERSITÄT DARMSTADT

- Mächtig, effizient und ausgereift
- Gut dokumentiert
- Steile Lernkurve (erfordert Wissen über Templates, Functionobjects, Iteratoren, Mixins, …)
- Wird mit Boost noch mehr ausgebaut
- Vielleicht sogar als der Vorteil von C++ zu betrachten!