Programmierpraktikum C und C++

Speicherverwaltung und Lebenszyklus

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

Wo leben meine Daten?

Der Stack und der Heap

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Der Stack und der Heap

TECHNISCHE UNIVERSITÄT DARMSTADT

C++

```cpp
// Primitive on Stack
int  intOnStack = 42;
cout << intOnStack << endl;

// Primitive on Heap
int* intOnHeap = new int(42);
cout << intOnHeap << endl;
cout << *intOnHeap << endl;

// Object on Stack
Building buildingOnStack(3);
buildingOnStack.runSimulation();

// Object on Heap
Building* buildingOnHeap = new Building(3);
buildingOnHeap->runSimulation();

// Clean Heap
delete intOnHeap;
delete buildingOnHeap;
```

Java

```java
// Primitive on Stack
int  intOnStack = 42;
System.out.println(intOnStack);

// Primitive on Heap
// Not possible!


// Object on Stack
// Not possible!


// Object on Heap
Building buildingOnHeap = new Building(3);
buildingOnHeap.runSimulation();

// Clean Heap
// Handled by Garbage Collector!
```

Intermezzo

TECHNISCHE UNIVERSITÄT DARMSTADT

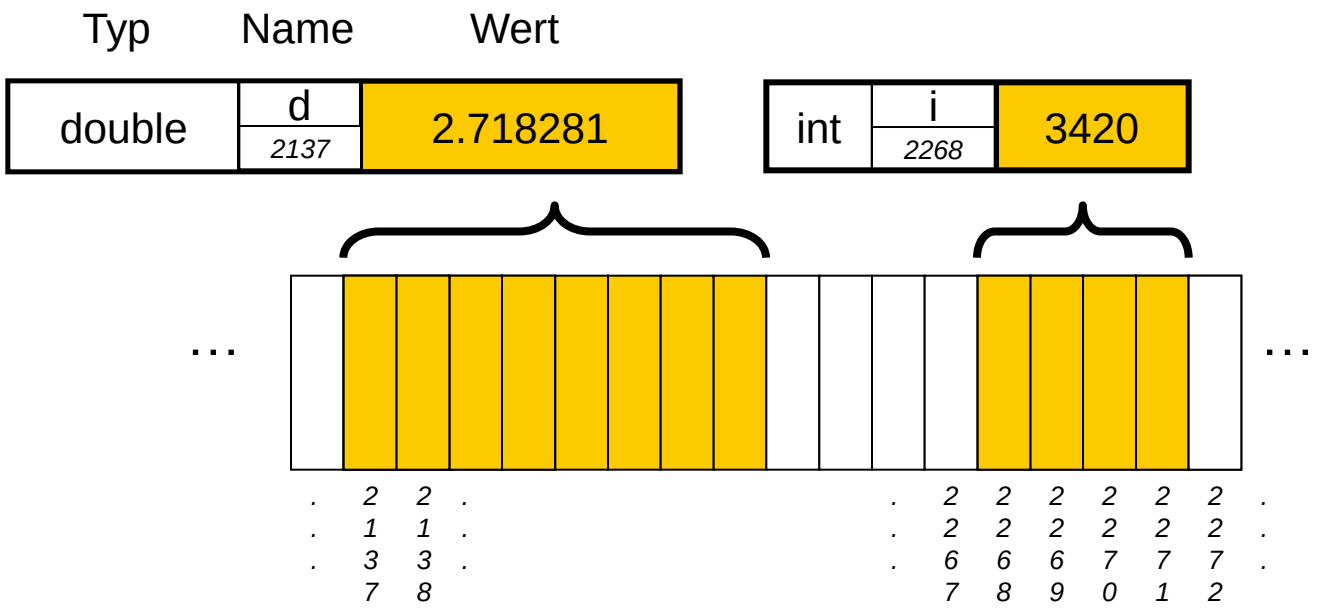
Wieso braucht man überhaupt Speicher auf dem Heap wenn der Stack die Speicherverwaltung übernimmt und auch noch so viel effizienter ist?

Variablen und Zeiger: Was ist eine Variable?

TECHNISCHE UNIVERSITÄT DARMSTADT

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

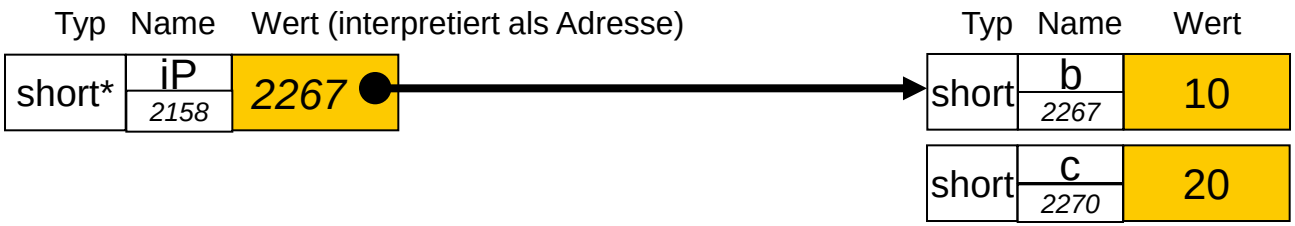


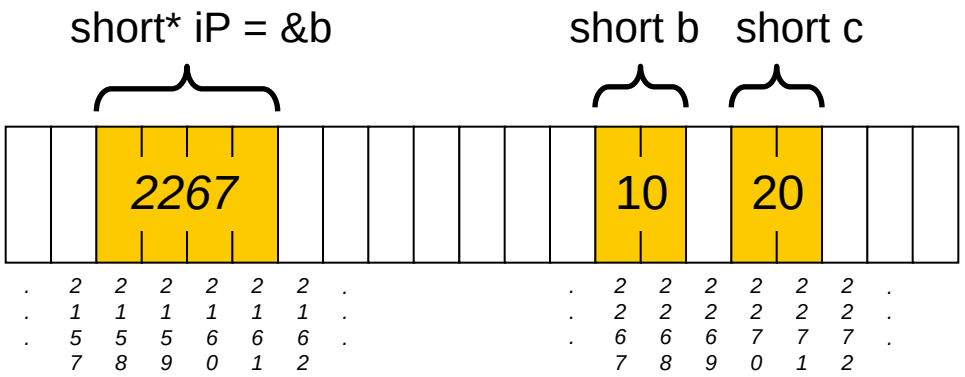
Variablen und Zeiger: Was ist ein Zeiger?

TECHNISCHE UNIVERSITÄT DARMSTADT

Ein Zeiger 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

TECHNISCHE UNIVERSITÄT DARMSTADT

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

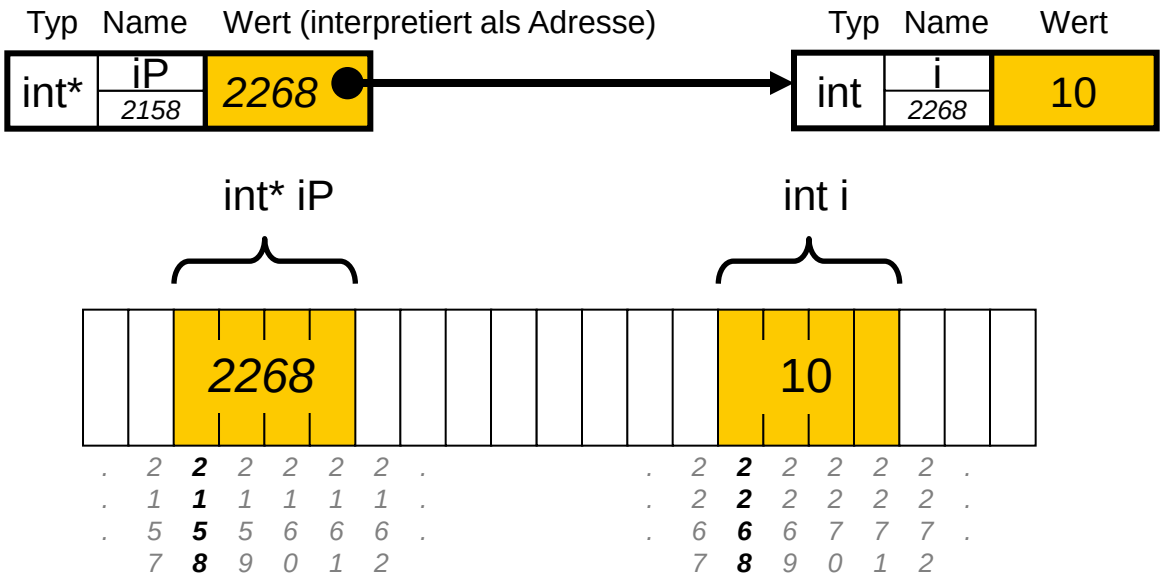
```
int i = 42;

int* iP;

iP = &i;

int j = *iP;

int* jP = iP;
```

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

Dereferenzierung eines Zeigers, um auf den gezeigten Inhalt zu kommen

Ohne Dereferenzierung bekommt man den Wert des Zeigers (die Adresse eines ints)

Variablen und Zeiger: Syntax




```
cout <<  i   << endl;
cout <<  iP  << endl;
cout << &i   << endl;
cout << *iP  << endl;
cout << &iP  << endl;
```

```
10
2268
2268
10
2158
```

Intermezzo

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Immer const verwenden wenn möglich!

TECHNISCHE UNIVERSITÄT DARMSTADT

Zeiger auf const int

```
int i = 42;

const int* iP;

iP = &i;

(*iP)++;  // assignment of read-only variable iP
```

Nicht erlaubt!

Const Zeiger auf int

```
int j = 7;

int* const jP = &j;

(*jP)++;
```

Muss sofort initialisiert werden, kann nicht neu definiert werden

erlaubt

```
jP = &i;
```

Nicht erlaubt!

Const Zeiger auf const int

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

Was ist eine (C++)-Referenz?

TECHNISCHE UNIVERSITÄT DARMSTADT

Eine Referenz ist ein **const** Zeiger, der automatisch dereferenziert wird (angenehme Syntax)

```
int i = 42;

int* const iP = &i;

(*iP)++;
```

```
int i = 42;

int& iR = i;

iR++;
```

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

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

```
const int& iR = i;

cout << iR << endl;
```

Intermezzo

TECHNISCHE UNIVERSITÄT DARMSTADT

Wieso soll ich konsequent const verwenden?

Wann soll ich const verwenden und wann nicht?

Wieso const?

TECHNISCHE UNIVERSITÄT DARMSTADT

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. Wird für Objekte und Methoden sinnvoll verallgemeinert (sehen wir gleich am Beispiel)

OO mit const

TECHNISCHE UNIVERSITÄT DARMSTADT

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

void printFloorPlan() const;

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


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

const Methode, verändert den Zustand des Objekts nicht

building darf nicht verändert werden

Es dürfen nur const Methoden aufgerufen werden

Konstruktor, Destruktor und Copy-Konstruktor

Konstruktor, Destruktor und Copy-Konstruktor

TECHNISCHE UNIVERSITÄT DARMSTADT

Initialisierungsliste für Konstruktor

Copy-Konstruktor

Destruktor

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

private:
    int number;
};
```

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

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

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


Parameterübergabe bei Methodenaufrufen

TECHNISCHE UNIVERSITÄT DARMSTADT

Parameter werden in C++ immer per Wert übergeben (call-by-value)

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

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

```
Creating floor [0]

Copying floor [0]

This is floor [1]
Destroying floor [1]

Destroying floor [0]
```

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

Objekt wird automatisch zerstört wenn iWorkOnACopy zu main zurückkehrt…

Parameterübergabe bei Methodenaufrufen

TECHNISCHE UNIVERSITÄT DARMSTADT

Wieso nicht?

Da dies nicht immer gewollt ist, gibt es folgende Möglichkeiten:

(1): Übergabe „per Referenz“

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

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

Es wird keine Kopie des Objekts angelegt

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

Eine Referenz wird „per Wert übergeben“

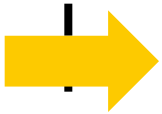
iUseAReference kann aber das Objekt beliebig verändern!

Parameterübergabe bei Methodenaufrufen

TECHNISCHE UNIVERSITÄT DARMSTADT

Da dies nicht immer gewollt ist, gibt es folgende Möglichkeiten:
(2): Übergabe per const Referenz

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

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



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

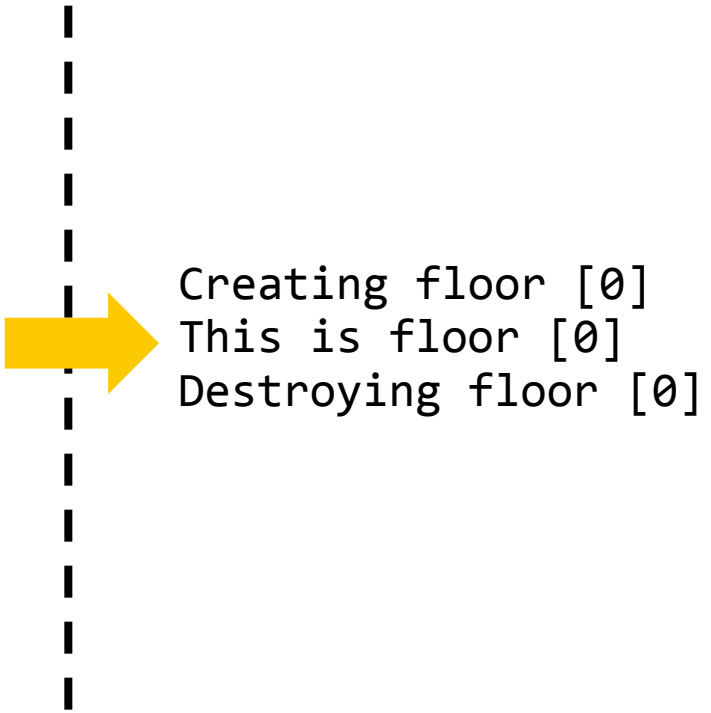
Dies sollte grundsätzlich die Default-Übergabestrategie sein!

Parameterübergabe bei Methodenaufrufen

Da dies nicht immer gewollt ist, gibt es folgende Möglichkeiten:

(3): Übergabe per Zeiger

```
void iUseAPointer(Floor* floor){
  cout << "This is floor ["
       << floor->getNumber()
       << "]"
       << endl;
}

int main() {
  Floor floor(0);
  iUseAPointer(&floor);
}
```



Intermezzo

TECHNISCHE UNIVERSITÄT DARMSTADT

Wieso ist die Übergabe per const & ein sinnvoller Default?

Wann ist die Übergabe per const & nicht möglich?

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

Stolperfallen bei der Speicherverwaltung

TECHNISCHE UNIVERSITÄT DARMSTADT

1. Hängende Zeiger
2. Speicherlecks

Hängende Zeiger:
Rückgabe nicht mehr existierender Objekte

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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!

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

```
Creating floor [0]
Copying floor [0]
Making next floor[1]
Destroying floor [1]
Next floor is floor [1]
Destroying floor [0]
```

Rückgabe von Objekten durch Kopieren

TECHNISCHE UNIVERSITÄT DARMSTADT

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

int main() {
  Floor floor(0);

  Floor nextFloor = makeNextFloor(floor);

  cout   << "Next floor is floor ["
         << nextFloor.getNumber()
         << "]"
         << endl;
}
```

```
Creating floor [0]

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

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

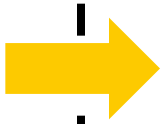
```
Creating floor [0]

Copying floor [0]
Made next floor [1]

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

gcc ist in der Lage, zu erkennen, wann Kopien vermieden werden können:
http://en.wikipedia.org/wiki/Copy_elision

Rückgabe von Objekten auf dem Heap

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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;
}
```



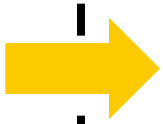
```
Creating floor [0]

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

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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;
}
```

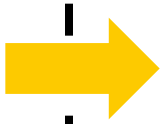


```
Creating floor [0]

Copying floor [0]
Made next floor [1]

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

Hängende Zeiger:
Frühzeitige Zerstörung von Objekten

TECHNISCHE UNIVERSITÄT DARMSTADT

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

  delete floor;

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



```
Creating floor [0]
Destroying floor [0]

Dangling reference to floor:
[5444032]
```

Extrem gefährlich!

Hängende Zeiger:
Nochmalige Zerstörung von Objekten

TECHNISCHE UNIVERSITÄT DARMSTADT

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

  delete floor;
  delete floor;
}
```

```
Creating floor [0]
Creating floor [1]
Destroying floor [1]
Destroying floor [5903232]
```

Extrem gefährlich!

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

  delete floor;

  floor = 0;

  delete floor;
}
```

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

Nach dem Löschen immer auf „null“ setzen!

Speicherlecks

TECHNISCHE UNIVERSITÄT DARMSTADT

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

  floor = otherFloor;
  otherFloor = floor;

  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.

SmartPointer: Motivation

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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?

SmartPointer: Boost to the rescue!

TECHNISCHE UNIVERSITÄT DARMSTADT

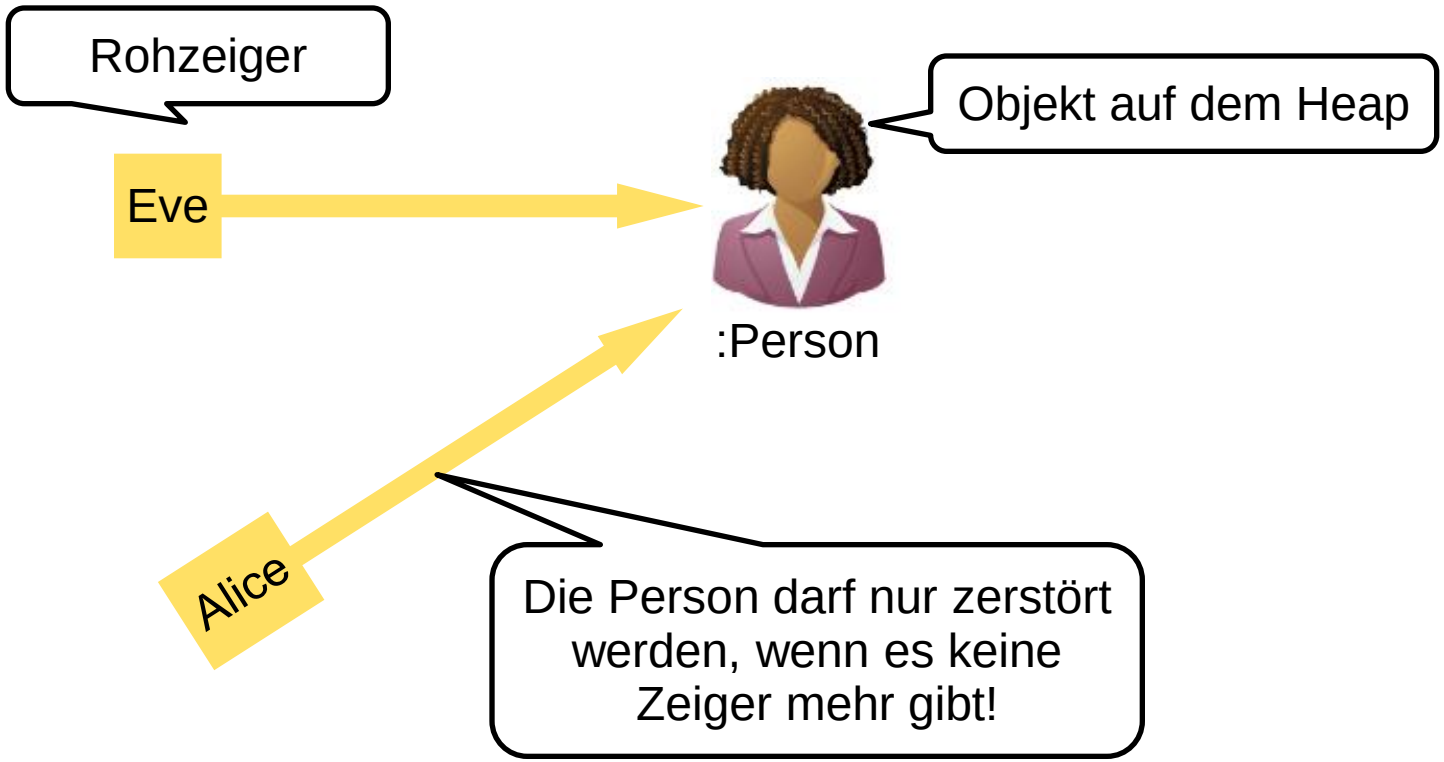
"...one of the most highly regarded and expertly designed C++ library projects in the world."

Herb Sutter, Andrei Alexandrescu, C++ Coding Standards

Im folgenden werden wir boost Smartpointer zur Lösung des Speicherverwaltungsproblems in C++ kennenlernen

http://www.boost.org/

Ohne SmartPointer

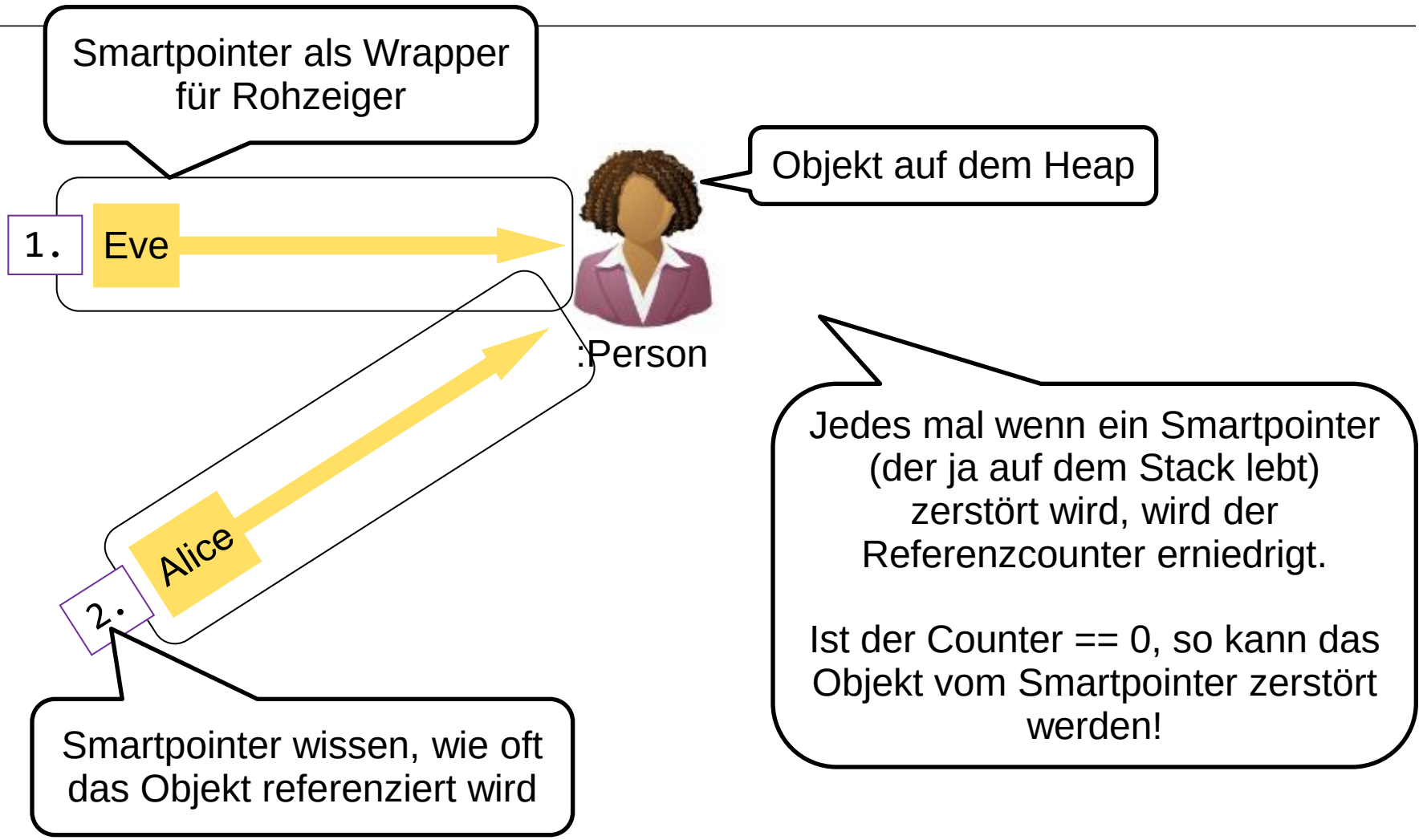


Intermezzo

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Mit boost::shared_ptr



Ohne SmartPointer

TECHNISCHE UNIVERSITÄT DARMSTADT

```
#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;
};
```

```
#include "Person.h"
#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 SmartPointer

TECHNISCHE UNIVERSITÄT DARMSTADT

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

#include "Person.h"

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;
}
```

```
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 boost::shared_ptr

TECHNISCHE UNIVERSITÄT DARMSTADT

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

#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.h"
#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;
}
```


Mit boost::shared_ptr

TECHNISCHE UNIVERSITÄT DARMSTADT

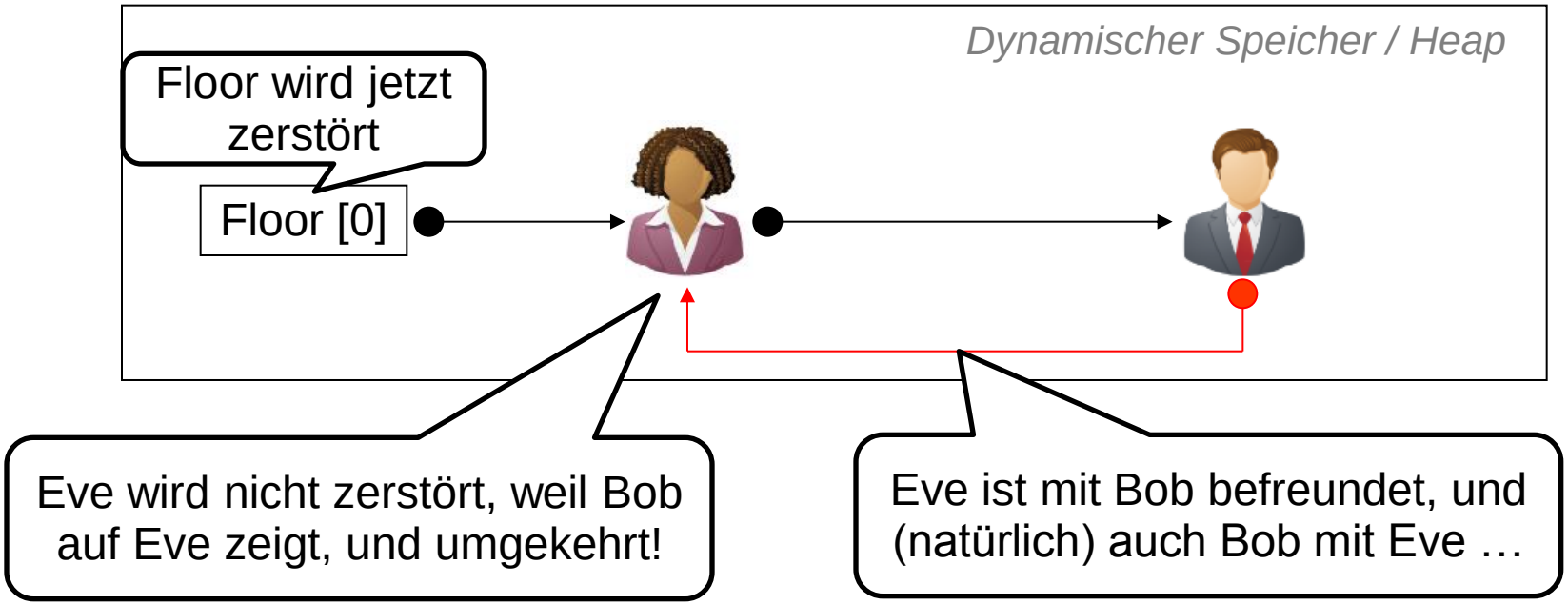
```
#include <iostream>
using namespace std;

#include "Person.h"

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);
}
```

```
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

shared_ptr ist nicht perfekt:

- Etwas langsamer als Rohzeiger
- Erkennt zirkuläre Abhängigkeiten nicht:



boost::weak_ptr

TECHNISCHE UNIVERSITÄT DARMSTADT

- **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, das 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?

Zusammenfassung

Variable const

Zeiger &

Referenz *

delete

T(const T&)

new

1. Eve

:Person