Programmierpraktikum C und C++

Speicherverwaltung und Lebenszyklus

TECHNISCHE UNIVERSITÄT DARMSTADT

Roland Kluge

roland.kluge@es.tu-darmstadt.de

ES Real-Time Systems Lab

Prof. Dr. rer. nat. Andy Schürr

Dept. of Electrical Engineering and Information Technology

Dept. of Computer Science (adjunct Professor)

www.es.tu-darmstadt.de

Wo leben meine Daten? … und wie lange?

TECHNISCHE UNIVERSITÄT DARMSTADT

Stack und 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

Stack und Heap – Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

C++		Java
`int intOnStack = 42;` `cout << intOnStack << endl;`	„Primitiv“ auf dem Stack	`int intOnStack = 42;` `System.out.println(intOnStack);`

`int *intOnHeap = new int(42);`
`cout << intOnHeap << endl;`
`cout << *intOnHeap << endl;`
	„Primitiv“ auf dem Heap	`// Not possible!`

`Building buildingOnStack(3);` `buildingOnStack.runSimulation();`	Objekt auf dem Stack	`// Not possible!`

`Building *buildingOnHeap =`
`  new Building(3);`
`buildingOnHeap->runSimulation();`
	Objekt auf dem Heap	`Building buildingOnHeap =`
		`  new Building(3);`
		`buildingOnHeap.runSimulation();`

`delete intOnHeap;` `delete buildingOnHeap;`	Heap aufräumen	`// Handled by Garbage Collector!`

Intermezzo

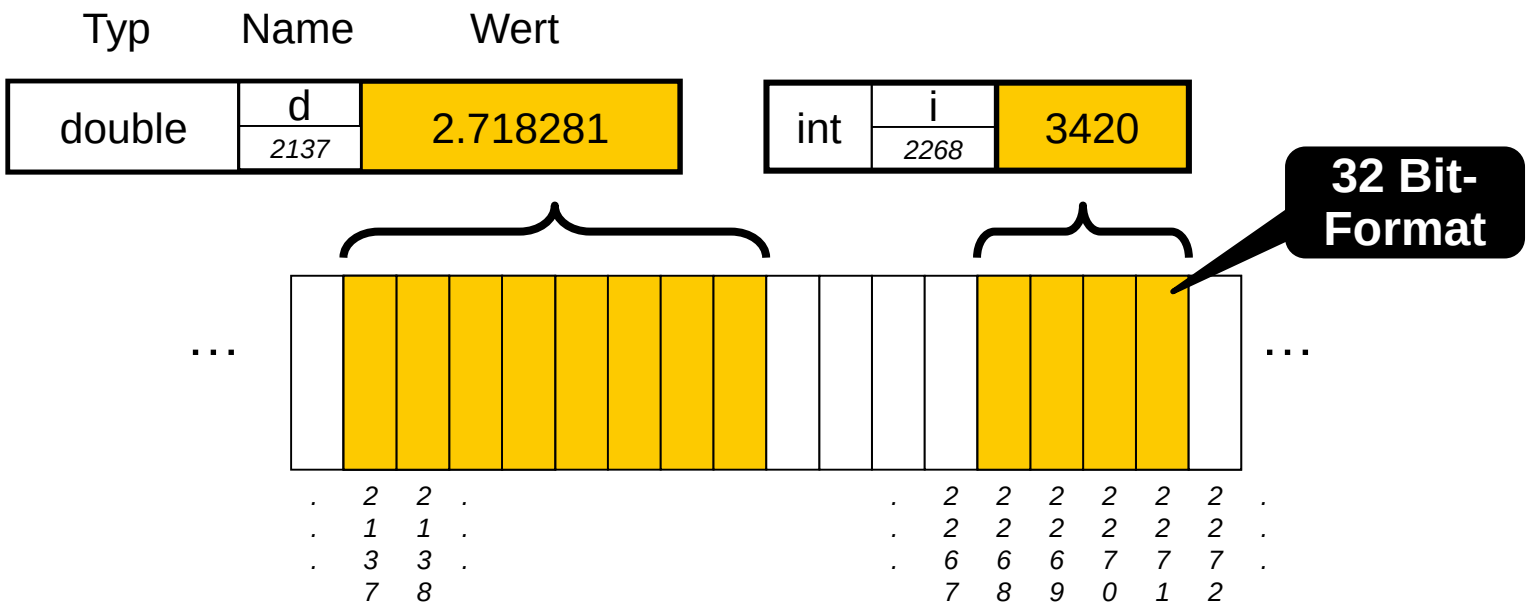
TECHNISCHE UNIVERSITÄT DARMSTADT

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?

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

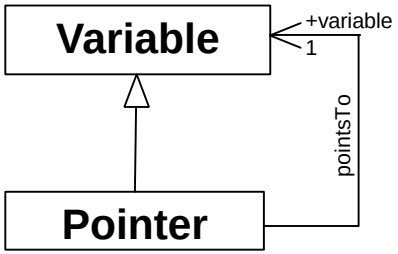
Der **Typ einer Variable** bestimmt die Größe des reservierten Speicherplatzes und die Interpretation der enthaltenen Daten

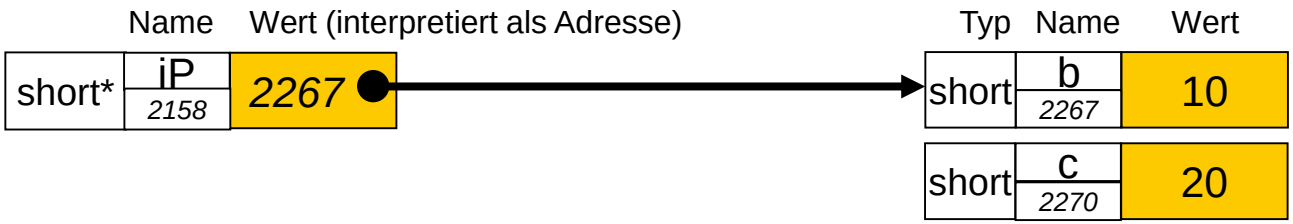


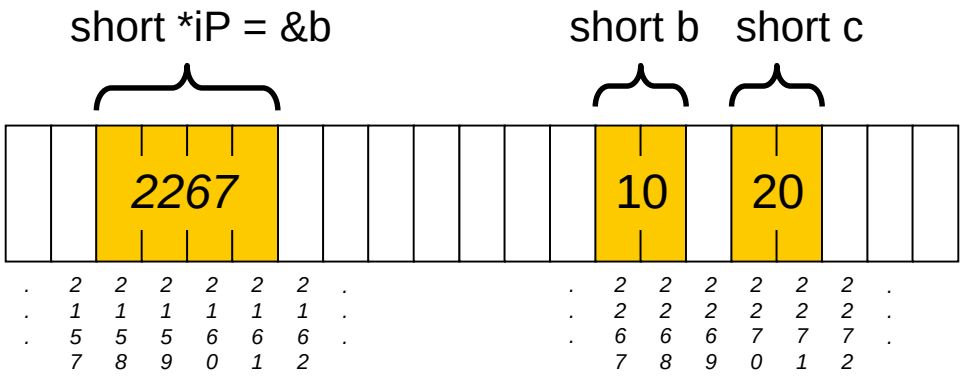
Variablen und Zeiger: Was ist ein Zeiger?

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

TECHNISCHE UNIVERSITÄT DARMSTADT

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

```
int i = 42;

int *iP;

iP = &i;
```

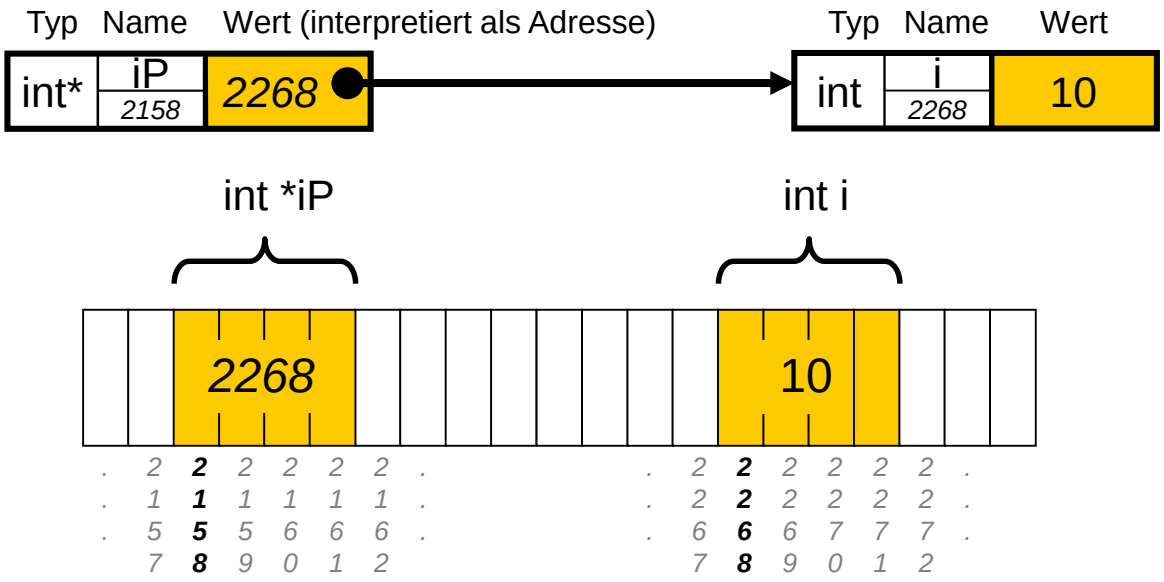
Definition eines Zeigers vom Typ *int** durch Zuweisung einer Adresse (Referenzierung)

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

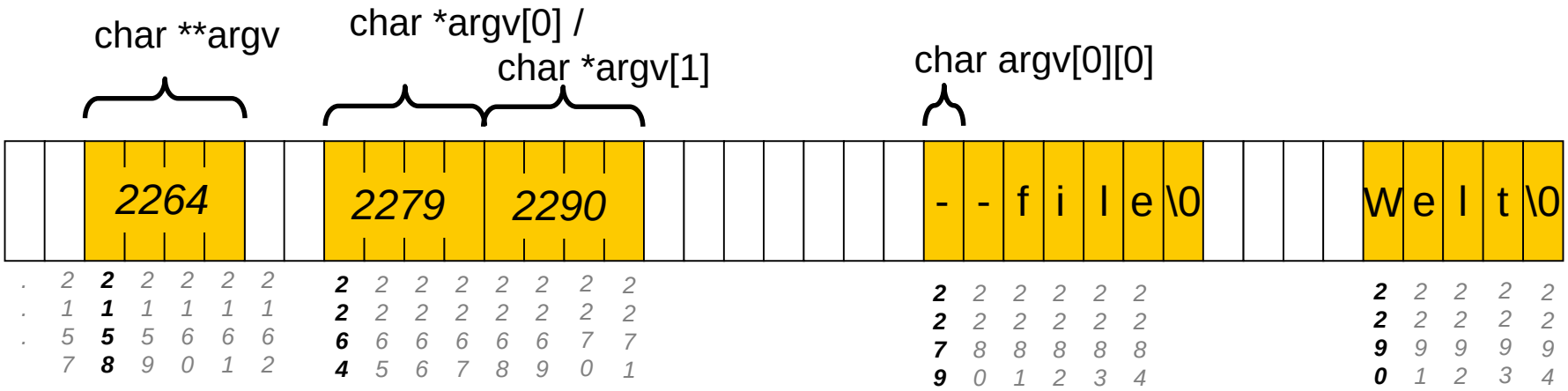



```
cout <<  i   << endl;
cout <<  iP  << endl;
cout << &i   << endl;
cout << *iP  << endl;
cout << &iP  << endl;
```

```
10
2268
2268
10
2158
```

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
Hallo 2279
Welt 2290

2279
```

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?

Unveränderlichkeit - *const*

TECHNISCHE UNIVERSITÄT DARMSTADT

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;
```

Eselsbrücke:
Das *const* bezieht sich immer auf das „Nächstliegende".

Was ist eine (C++)-Referenz?

TECHNISCHE UNIVERSITÄT DARMSTADT

Eine **Referenz** ist ein ***const* Zeiger**, der automatisch dereferenziert wird („Syntactic Sugar“).

```
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
```

?

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. Absicht des Programms wird dem Leser **„expliziter“.**

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

Objektorientierung 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();
}
```

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?

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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*).

Konstruktor, Destruktor und Copy-Konstruktor

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;
};
```

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

}
```

Copy-Konstruktor

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

Destruktor

```
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 iUseACopy(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 *iUseACopy* zu *main* zurückkehrt…

Parameterübergabe bei Methodenaufrufen (I)

TECHNISCHE UNIVERSITÄT DARMSTADT

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!

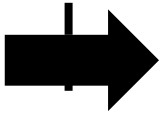
Parameterübergabe bei Methodenaufrufen (II)

TECHNISCHE UNIVERSITÄT DARMSTADT

Kopieren bei der Übergabe ist oft nicht gewollt. Lösungsmöglichkeiten:

(2) Übergabe per ***const*** **Referenz**

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

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



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

! **Dies sollte grundsätzlich die Default-Übergabestrategie sein.**

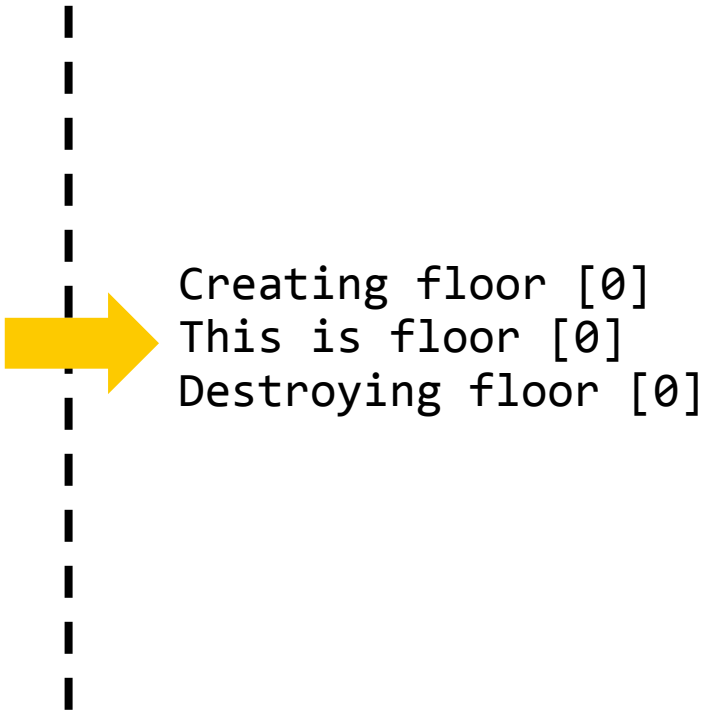
Parameterübergabe bei Methodenaufrufen (III)

TECHNISCHE UNIVERSITÄT DARMSTADT

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);
}
```



Intermezzo

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Schlagwort: *Const Correctness*

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

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

Assignment Operator

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

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

TECHNISCHE UNIVERSITÄT DARMSTADT

Rule of Three:
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.

Stolperfallen bei der Speicherverwaltung

TECHNISCHE UNIVERSITÄT DARMSTADT

1. Hängende Zeiger
2. Speicherlecks

http://static.tvtropes.org/pmwiki/pub/images/Bear_Trap_7423.jpg

Hängende Zeiger
Referenzen auf gelöschte Objekte zurückgeben

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!

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

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


Rückgabe von Objekten durch Kopieren

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;
}
```

Erwartet

```
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]
```

Tatsächlich

```
Creating floor [0]

Copying floor [0]
Made next floor [1]

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

g++ 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

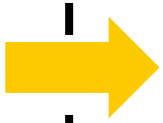
```
Floor* makeNextFloor(const Floor &floor){
  Floor *next = new Floor(floor);
  cout   << "Made next floor ["
         << next->getNumber() << "]"
         << endl;
  return next;
}

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]
```

Hängende Zeiger
Nochmalige Zerstörung von Objekten

TECHNISCHE UNIVERSITÄT DARMSTADT

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

  delete floor;
  delete floor;
}
```

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

Extrem gefährlich!

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

  delete floor;

  floor = 0;

  delete floor;
}
```

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

Nach dem Löschen immer auf „null“ setzen!

Speicherlecks

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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

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?

Smart Pointer: Boost to the rescue

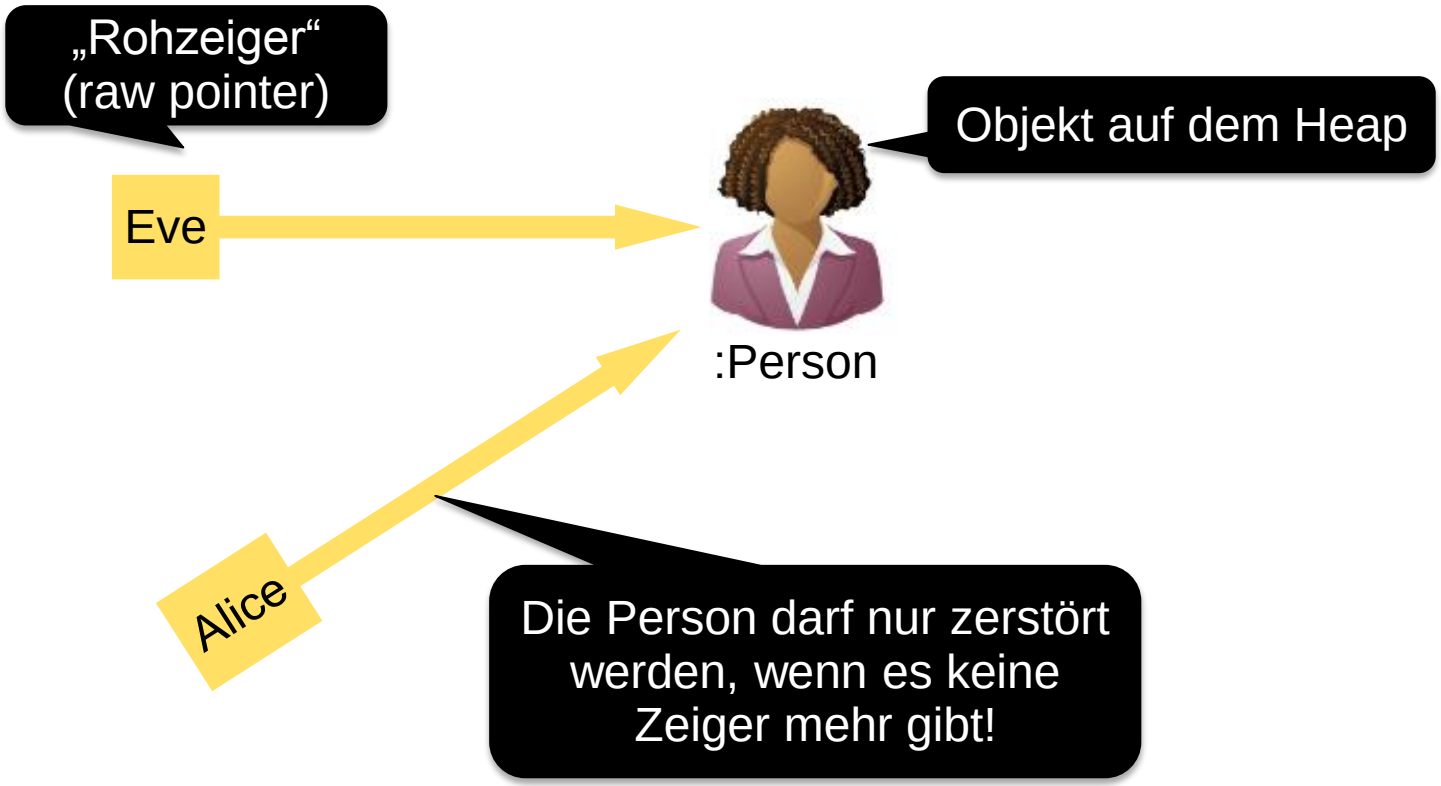
"...one of the most highly regarded and expertly designed C++ library projects in the world."

Herb Sutter, Andrei Alexandrescu, C++ Coding Standards

http://www.boost.org/

Array	Filesystem	Lambda	Odeint
Chrono	Function(al)	Math (advanced)	Smart Ptr
Date Time	Graph	MPI	System

Ohne Smart Pointer



Intermezzo

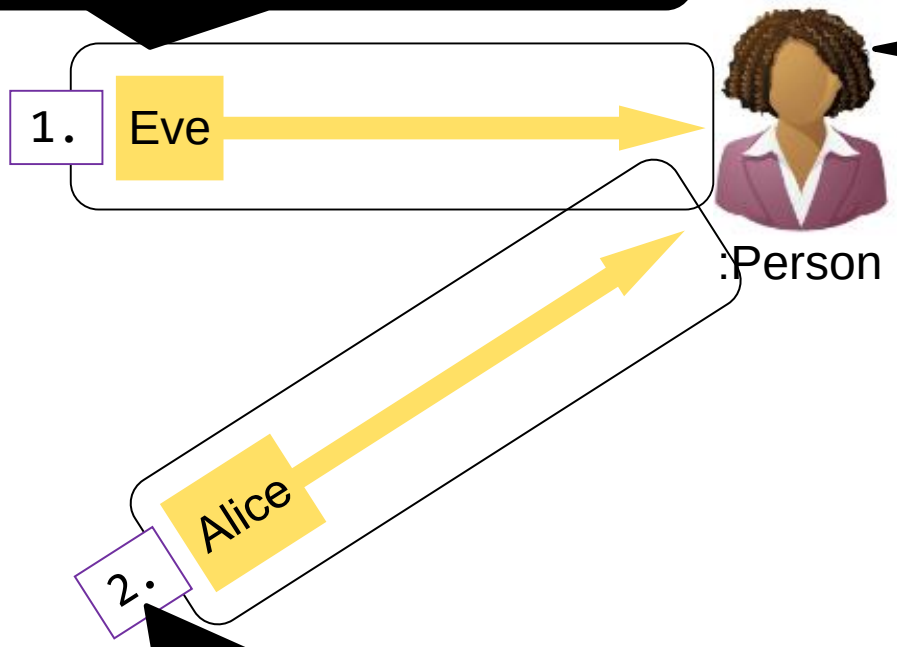
TECHNISCHE UNIVERSITÄT DARMSTADT

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

Mit boost::shared_ptr

TECHNISCHE UNIVERSITÄT DARMSTADT

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



Objekt auf dem Heap

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!

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

Ohne Smart Pointer

TECHNISCHE UNIVERSITÄT DARMSTADT

Person.h

```
#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

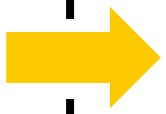
```
#include "Person.h"
#include <iostream>
using namespace std;

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;
}
```

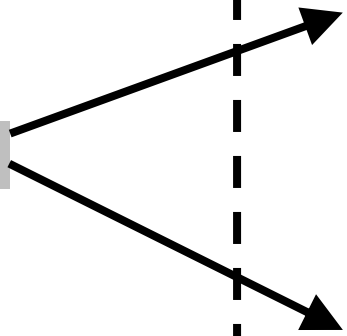
Ohne SmartPointer

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;
}
```

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 boost::shared_ptr

TECHNISCHE UNIVERSITÄT DARMSTADT

Person.h

```
#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;
```

Person.cpp

```
#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

main.cpp

```
#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);
}
```

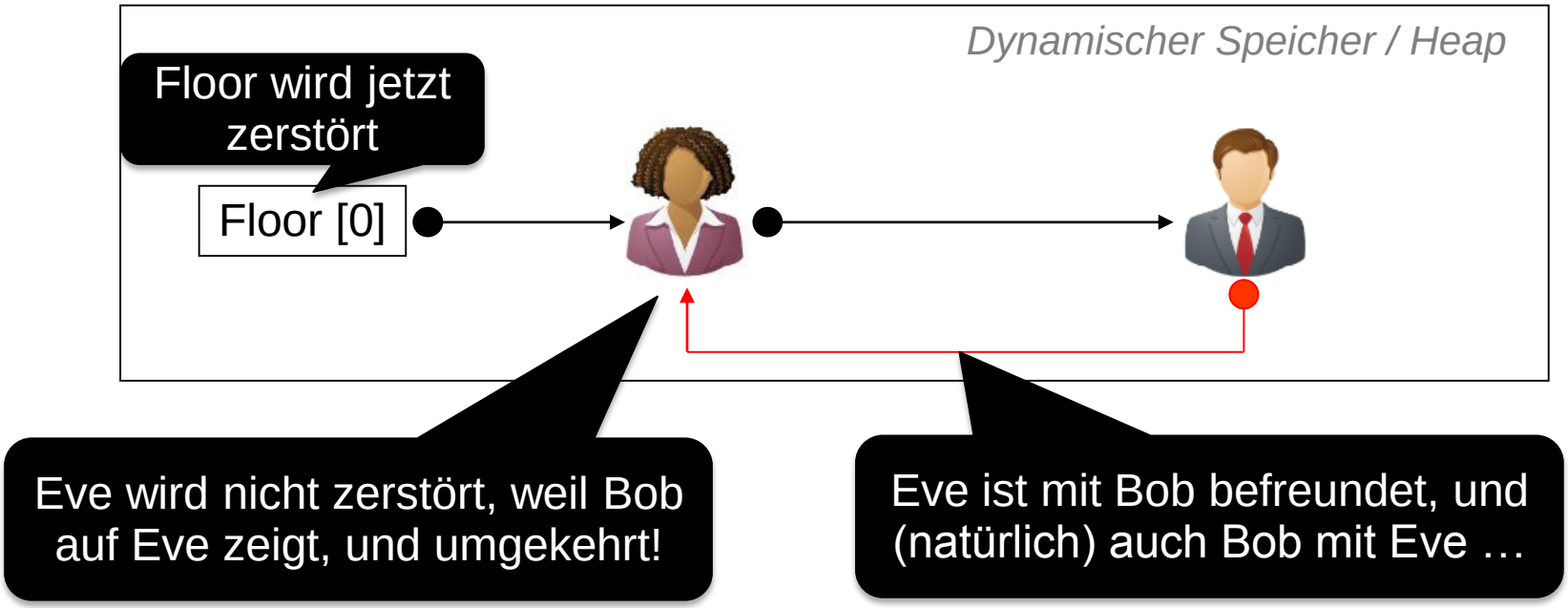
```
Created Eve

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

TECHNISCHE UNIVERSITÄT DARMSTADT

boost::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?

Mögliche Lösung für zyklische Zeiger

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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

TECHNISCHE UNIVERSITÄT DARMSTADT

Variable **const**

Zeiger &

Referenz *

delete

T(const T&)

new

1. Eve

:Person