

Programmierskurs C++ für Fortgeschrittene

SS 2012

Teil 3

Sandro Andreotti andreott@inf.fu-berlin.de

THEMA 1

SPEICHERARTEN

C++ Speicherarten

Speicherarten für Variablen:

1. **static**

- leben so lange das Programm läuft
- globale Variablen, statische lokale Variablen, externe Variablen, statische Member-Variablen

2. **auto** ⇒ Stack

- leben bis { }-Block verlassen wird
- lokale Variablen

3. **dynamic** ⇒ Heap

- mit **new** erzeugt
- leben bis zum **delete**

Stack

- Verwendet CPU-Stack \Rightarrow schnell (*stack pointer*)
- Speicherreservierung bei jedem Funktionsaufruf
- C'tor bei Definition
- D'tor beim Verlassen des umgebenden { }-Blocks

```
void f()
{
    int i;
    for (i=0; i<10; ++i)
    {
        myObject obj;
        //...
    }
}
```

\Leftarrow Stack-Platz reserviert

\Leftarrow c'tor `myObject::myObject()`

\Leftarrow d'tor `obj.~myObject()`

\Leftarrow Stack-Platz freigegeben

Die „Lebenszeit“ ist wie gezeigt vom Kontrollfluss abhängig. Soll die Lebenszeit einer Variablen dynamisch bestimmt werden, muss sie auf dem Heap erzeugt werden.

Die maximal Stackgröße wird beim Linken des Programmes festgelegt (z.B. auf 1 MByte).

„C'tor“ = „Constructor“

„D'tor“ = „Destructor“

Heap-Manager

Operationen:

1. **new**: Speicherblock der Größe $\geq x$ reservieren.
2. **delete**: Speicherblock freigeben

Randbedingungen und Wünsche:

- Speicherblöcke nicht mehrfach vergeben
- Speicherblöcke nicht verschieben
- Minimiere Fragmentierung
- Minimiere Laufzeit von **new** und **delete**

Beobachtung: das geht im *worst case* nicht!

[Literatur:

Paul R. Wilson, Mark S. Johnstone, Michael Neely and David Boles: **Dynamic Storage Allocation: A Survey and Critical Review**. Proc. 1995 Int'l Workshop on Memory Management. Springer LNCS. (<ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps>)]

Jede Strategie zur Verteilung des aktuellen Speichers scheitert im *worst case* an der Bedingung, die Fragmentierung zu minimieren.

Dass Speicherblöcke nicht verschoben werden können, folgt aus der Konstruktion der Sprache: Es ist i.A. nicht feststellbar, wo überall ein versteckter Pointer auf einen Speicherblock gehalten wird. Aus diesem Grund muss gewährleistet sein, dass einmal allozierte Speicherbereiche nicht von selbst verschoben oder gar gelöscht werden.

Bei Sprachen ohne „Pointer-Magie“ ist dies jedoch möglich: Bei diesen Sprachen kann anstelle eines Heap-Managers ein Garbage-Collector die Speicher-Verwaltung übernehmen.

Heap: Probleme und Lösungshilfen

- Langsamkeit, Fragmentierung, Overhead:
Vorschlag: Pool Allocator
- Memory Leaks:
Vorschlag: Auto Pointer, Smart Pointer



(memory leak)

Pool Allocator

- Situation: Große (vorher nicht bekannte) Anzahl von immer gleich großen Objekten speichern.
- Ansatz:
 - Reserviere auf dem Heap Blöcke, die gleich für mehrere Objekte reichen.
 - Reserviere und konstruiere nach Bedarf Objekte in nicht-vollem Block.
 - Verwalte freigegebene Objekte in Liste.
 - Reserviere neue Blöcke, wenn alle bisherigen belegt.

[Literatur: Stroustup, „The C++ Programming Language“, Chapter 19.4.2: „A User-Defined Allocator“]


new und delete Operatoren

Die Operatoren **new** und **delete** wenden die Funktionen **operator new** und **operator delete** an.

```
int * x = new int;
```

ist das gleiche wie

```
int * x = (int *) operator new (4);
```



sizeof (int)

Rückgabe von **operator-new**:

void * auf reservierten Speicherbereich.

[Literatur: ISO/IEC 14882 – Standard, 3.7.3 „Dynamic storage duration“, 5.3.4 „New“, 5.3.5 „Delete“]

Beim erzeugen von Objekten mit **new** macht **new** jedoch mehr als **operator new**:
new ruft den C‘tor des erzeugten Objekts auf.

new und **delete** Operatoren (II)

new-Funktionen:


```
void * operator new (size_t size, ...);  
void * operator new [] (size_t size, ...);
```

optional weitere Argumente



delete-Funktionen:

```
void operator delete (void * buf, ...);  
void operator delete [] (void * buf, ...);
```



Als erstes Argument (**size**) wird **operator new** und **operator new []** die Größe des zu bereitzustellenden Speicherbereiches in Bytes übergeben. Ein Zeiger auf den bereitgestellte Speicherbereich wird zurückgeliefert.

Als erstes Argument (**buf**) wird **operator delete** und **operator delete []** der Zeiger auf den freigegebenen Speicherbereich (der früher mit **new** oder **new []** bereitgestellt worden ist) übergeben.

new und **delete** Operatoren (III)

Regeln:

- **new** sucht Version von **operator new** oder **operator new []**, deren Argumente passen.

```
int * x = new (17, "hallo") int;
```

matched mit

```
void * operator new (size_t size, int i, char * s);
```



The diagram shows two arrows originating from the boxed code above. One arrow points from the integer '17' in the new expression to the 'int i' parameter in the operator new function signature. The other arrow points from the string literal '"hallo"' in the new expression to the 'char * s' parameter in the operator new function signature.

- Wird mit **new** ein Array erzeugt, so wird wenn möglich **operator new []** aufgerufen.

Wenn es keine geeignete **operator new []**-Funktion gibt, aber eine passende **operator new**-Funktion, so wird letztere aufgerufen.

new und **delete** Operatoren (IV)

- Bei Objekten ruft **new** einen C'tor auf:

```
MyClass * obj = new MyClass(Par1, Par2);
```

- Bei Arrays von Objekten ruft **new** den default C'tor auf:

```
MyClass * arr = new MyClass [20];
```

auto_ptr - Template

Auto Pointer:

- Klasse,
- verhält sich wie Pointer auf Zielobjekt,
- ruft im eigenen D'tor **delete** auf Zielobjekt auf.

```
#include <memory>
void f()
{
    std::auto_ptr<MyClass> foo(new MyClass);
}
```

Aufruf C'tor von MyClass

D'tor von foo ruft D'tor von MyClass auf

Anders ausgedrückt: Wird ein Auto Pointer zerstört, so bringt er zuvor noch sein Zielobjekt um.

„Verhält sich wie ein Pointer auf Zielobjekt“ bedeutet: Alle wichtigen Operatoren sind so überladen, dass man alle für Pointer üblichen Operationen durchführen kann.

Die Template Klasse **auto_ptr** ist Teil des C++ Standards. Definiert wird sie im Header **<memory>**.

[Literatur: ISO/IEC 14882 – Standard, 20.4.5 „Template class auto_ptr“]

unique_ptr - Template

Unique Pointer (ersetzt auto_ptr):

C++11

- Klasse,
- verhält sich wie Pointer auf Zielobjekt,
- ruft im eigenen D'tor **delete** auf Zielobjekt auf.

```
#include <memory>
void f()
{
    std::unique_ptr<MyClass> foo(new MyClass);
    std::unique_ptr<MyClass[]> foo(new MyClass[20]);
}
```

Aufruf C'tor von MyClass

Im neuen Standard wird der auto_ptr als „deprecated“ behandelt und soll durch den unique_ptr ersetzt werden.

Der unique_ptr funktioniert auch für dynamic arrays.

Dieser bietet dieselbe Funktionalität ist allerdings sicherer (move semantics)

shared_ptr - Template

C++11

Shared Pointer (*reference counting*):

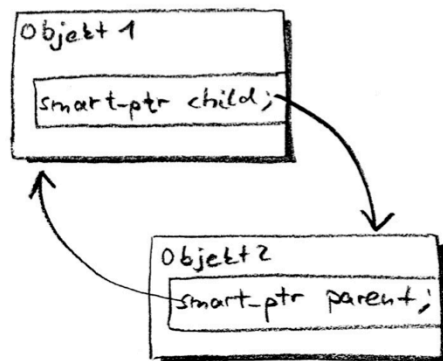
- Klasse,
- verhält sich wie Pointer auf Zielobjekt,
- Zielobjekt hat Referenz-Zähler,
- Mehrere shared pointer können auf dasselbe Objekt zeigen:
- Wenn alle shared_pointer die auf Zielobjekt zeigen zerstört oder von Zielobjekt weggerichtet werden
 - Zielobjekt wird zerstört

Die Idee hinter den Smart Pointern ist: Im Referenz-Zähler wird gezählt, wie viele Smart Pointer noch auf das Objekt zeigen. Sobald dieser Zähler auf 0 fällt, gibt es keinen Pointer mehr, der auf das Objekt zeigt, das Objekt ist somit nicht mehr benutzbar und kann zerstört werden. Voraussetzung: Auf das Zielobjekt wird ausschließlich über Smart Pointer (und nicht noch zusätzlich über andere „normale“ Pointer) zugegriffen.

Aufgabe: Programmieren Sie einen Smart-Pointer für beliebige Zielobjekte (Klasse des Zielobjekts spezifizieren über Template-Argument!), sowie eine Klasse **RefCount**, die den Referenzzähler enthält. **RefCount** wird der Klasse des Zielobjekts als Basisklasse gegeben.

Probleme bei Shared Pointern

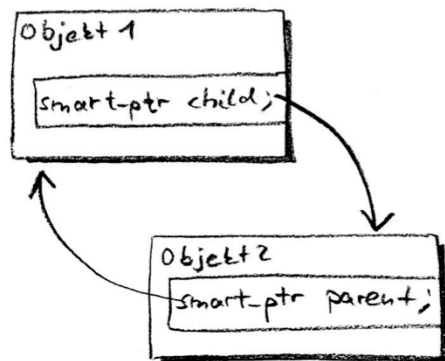
Bei nicht sachgemäßer Benutzung: Objekte halten sich gegenseitig am Leben \Rightarrow Leak!



(stupid pointers)

Probleme bei Shared Pointern

Bei nicht sachgemäßer Benutzung: Objekte halten sich gegenseitig am Leben \Rightarrow Leak!



C++11
Solution: weak_ptr

(stupid pointers)

Das weak_ptr template im neuen Standard implementiert eine „non owning“ oder „weak“ reference auf ein Zielobjekt.

Wenn über einen weak_ptr auf das Zielobjekt zugegriffen werden soll muss er zunächst in einen shared_ptr umgewandelt werden (member function lock())

THEMA 2

DELEGATION

Das Delegation-Problem

Das Problem: sei z.B. folgendes Programm gegeben:

```
struct Auto
{
    void tanken() { std::cout << "Benzin"; ...}
    void starten()
    {
        if (Tankfuellung == 0) tanken();
        ...
    }
};

struct Solarmobil: Auto
{
    void tanken() { std::cout << "Sonne"; ... }
};
```

Das Delegation-Problem (II)

```
...  
Solarmobil mob;  
mob.starten();
```

Ausgabe: **Benzin** (!)

Grund: Die Basisklasse **Auto** kennt die abgeleitete Klasse **Solarmobil** nicht.

```
void starten()  
{ ...  
    tanken();  
}
```

⇐ this-Pointer hat Typ **Auto ***

Virtuelle Funktionen

Lösungsmöglichkeit: **virtual**

```
struct Auto
{
    virtual void tanken() { ... }
};

...

Solarmobil mob;
mob.starten(); //Ausgabe: "Sonne"
```

Nur nichtstatische Member-Funktionen können virtuell sein.

Eine Member-Funktion ist virtuell, wenn sie mit **virtual** deklariert wird, oder wenn es eine gleichnamige virtuelle Funktion in einer Basisklasse gibt.

Klassen mit virtuellen Funktionen heißen auch „polymorph“.

Dieses Problem könnte z.B. auch mit **tag dispatching** gelöst werden. Anders liegt der Fall jedoch, wenn z.B. eine Liste von Autos nacheinander betankt werden sollte, und jeweils die richtige Funktion aufzurufen ist. In einem solchen Fall wäre die Anwendung von virtuellen Funktionen durchaus angemessen.

final und override

final: verbietet overriding in abgeleiteter Klasse

```
struct Auto
{
    virtual void tanken(int) final { ... }
};

struct Solarmobil : Auto
{
    virtual void tanken(int) { ... }
    ...
}
```

C++11

←
Error: overriding verboten!

Eine Funktion die in der Basisklasse als final gekennzeichnet wird darf in abgeleiteten Klassen nicht überschrieben werden (kein overriding)

Es kann auch die gesamte Klasse als final gekennzeichnet werden. Es darf dann keine andere Klasse von ihr abgeleitet werden.

final und override

override: verhindert unabsichtliche Neudefinition

```
struct Auto
{
    virtual void tanken(int) { ...}
};

struct Solarmobil : Auto
{
    virtual void tanken(float) override { ...}
    ...
}
```

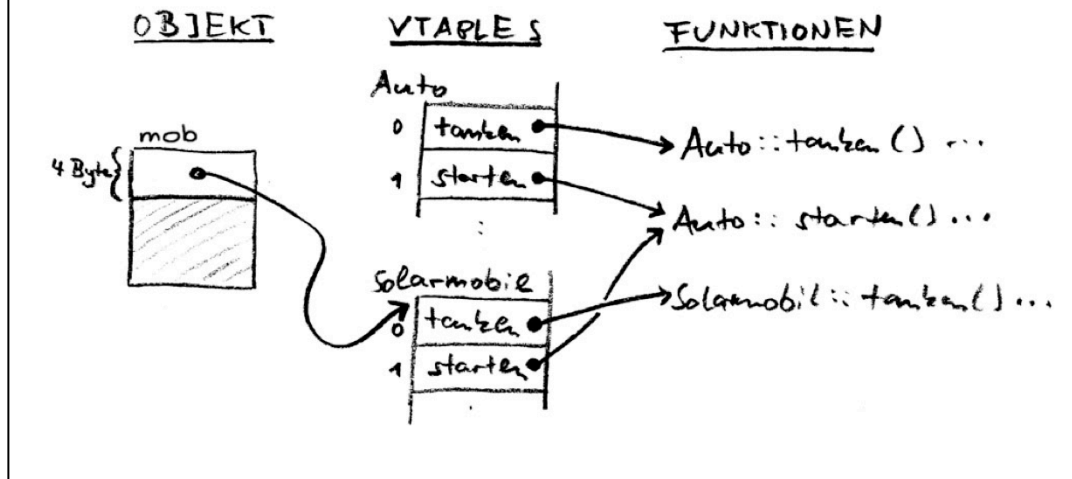
C++11

←
Error: Da kein overriding!

Da die Signatur in Basisklasse und abgeleiteter Klasse nicht übereinstimmen (unterschiedliche Parameter Typen) wird nicht die Basisklassenfunktion überschrieben (kein overriding), sondern eine neue tanken Funktion definiert.

Wie funktioniert **virtual**?

Prinzip: Objekt hält Zeiger auf eine Tabelle mit Zeigern auf die richtigen Funktionen.



Jede Klasse hat eine eigene VTABLE (bzw. eine VTABLE für jede ihrer Basisklassen, siehe unten). Es entstehen gewisse Probleme dabei, zu entscheiden, wo diese VTABLE abgespeichert werden soll, insbesondere bei Bibliotheksfunktionen (siehe dazu: Meyers, More Effective C++, Item 24).

Wie funktioniert **virtual**? (II)

Aufruf einer virtuellen Funktion:

C++:

```
mob.starten();
```

Assembler:

```
mov    edx,dword ptr [mob]  ← Lade Zeiger auf VTABLE  
call   dword ptr [edx+4]    ← Rufe zweite Funktion in VTABLE auf
```

Anmerkung: In diesem Beispiel haben wir die Tatsache unterschlagen, dass der Funktion **starten()** selbstverständlich auch ein Zeiger auf die aktuelle Objektinstanz **mob** (der **this**-Pointer) übergeben werden muss. (Mehr dazu: Siehe unten)

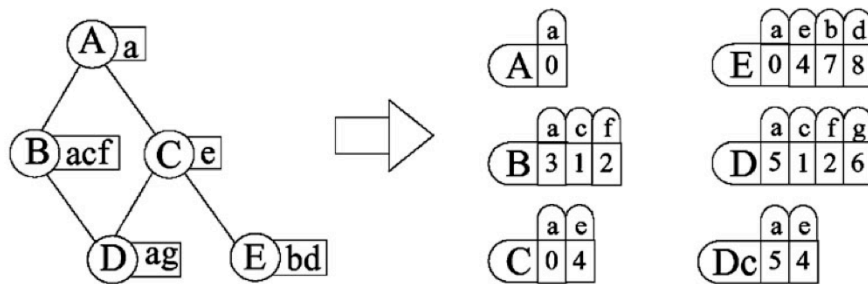
virt. Funk. und Mehrfachvererbung

Ein VTABLE-Pointer für jede Basisklasse:

```
struct D: B, C { };    D hält 2 Zeiger auf VTABLEs:
```

Table D: für Funktionen über **B**

Table Dc: für Funktionen über **C**



Anmerkung zur Grafik: Große Buchstaben bezeichnen Klassen bzw. die entsprechenden VTABLEs. Kleine Buchstaben sind Funktionen. Ziffern bezeichnen die Adressen der aufzurufenden Funktionen.

Der Grund für dieses Verfahren: Die VTABLE einer abgeleiteten Klasse ist eine Erweiterung der VTABLE ihrer Basisklasse. In B steht der Eintrag für Funktion c an zweiter Stelle, dies muss für die VTABLE D ebenfalls gelten. Nun steht bei C an zweiter Stelle jedoch der Eintrag für Funktion e und nicht für c (eine Funktion c gibt es in C nicht). Das Problem wird durch die Einführung getrennter VTABLES für jede Basisklasse (d.h. für jede Basisklasse, die mindestens eine virtuelle Funktionen enthält) gelöst.

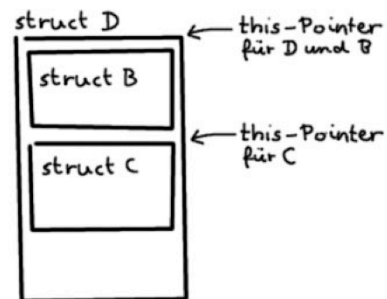
Das Problem mit dem **this**-Pointer

Aufruf von Basisklassen-Funktion:

```
static_cast<Base*>(this);
```

Mehrfachvererbung:

this-Pointer verschieb sich!



Problem bei virtuellen Funktionen:

Basisklasse steht erst zur Laufzeit fest.

Lösung: Speichere **this**-Verschiebung in VTABLE

In der VTABLE wird also ein Wert **delta_this** gespeichert, der die Zahl angibt, um die der **this**-Pointer bei Aufruf einer Funktion der Basisklasse verändert werden muss. Für die erste Basisklasse ist dies eine 0.

Diese Komplikation ist u.A. der Grund dafür, warum es in Sprachen wie Java keine Mehrfachvererbung gibt. Aus diesem Grund kann ein Funktionsaufruf in Java schneller ausgeführt werden, als der Aufruf einer virtuellen Funktion in C++.

Wie funktioniert **virtual**? (III)

Vollständiger Algorithmus:

Pseudocode:

```
vtable = [this + Offset_vtable_Pointer]
function = [vtable + Offset_function]
delta_this = [vtable + Offset_delta_this]
this += delta_this
call function
```

Ausdrücke mit [] bedeuten: Es wird die in [] angegebene Speicheradresse geladen.

In vielen Fällen ist **delta_this == 0**. Der GNU C++-Compiler verwendet daher eine Optimierung, nämlich sogenannte **Thunks**: Beim Aufruf einer virtuellen Funktion wird zunächst **delta_this** nicht geladen noch **this** nicht verändert. Wenn **delta_this == 0**, so funktioniert dies problemlos. In allen anderen Fällen wird in **vtable** nicht **function**, sondern die Adresse einer zusätzlichen Funktion **thunk** eingetragen. Diese Funktion **thunk** nimmt dann die Anpassung von **this** vor und springt anschließend zu **function**.

abstrakte Basisklassen

Abstrakte Klasse = hat nur virtuelle Funktionen

```
struct Abstract
{
    virtual void f() = 0;
};

struct Derived: Abstract
{
    void f() { /* implemented */ }
};

Abstract obj;    // Error!
Derived obj;     // OK
```

Abstrakte Basisklassen werden vornehmlich für die Definition von Interfaces benutzt: Man definiert Klassen, die keine Members außer nur virtuelle Funktionen haben. Von diesen Klassen wird abgeleitet. In der abgeleiteten Klasse werden die Funktionen aus dem Interface implementiert. Solange eine der nur virtuellen Funktionen noch nicht implementiert ist, kann von dem Objekt keine Instanz erzeugt werden.

virtual vs. inline

Faustregel: »Wenn **virtual** auf **inline** trifft, gewinnt (fast immer) **virtual**.«

```
struct C
{
    virtual inline void f() { }
};

C c;
c.f();    //das geht vielleicht noch mit inline.

C * pc = & c;
pc->f();  //das vermutlich nicht mehr.
```

Aufgabe: Probieren Sie das mit Ihrem Compiler aus. Finden Sie die Grenzen, bis zu denen der Compiler virtuelle Funktionen noch inlinen kann.

Fazit: virtuelle Funktionen sind aus mehreren Gründen langsamer als nicht virtuelle Funktionen: Zum einen wird der Funktionsaufruf relativ komplex (insbesondere bei Mehrfachvererbung), zum anderen wird das Inlinen der Funktion durch **virtual** unmöglich gemacht. Außerdem werden bei der Benutzung von virtuellen Funktionen die Objektinstanzen größer. Beides Grund genug, vorsichtig mit virtuellen Funktionen umzugehen, und genau zu prüfen, ob sich keine Alternative finden lässt, die diese Nachteile vermeidet.

Quiz: Was kostet ein **virtual**?

Frage: Wieviel Speicher verbrauchen die Klassen **A**, **B**, **C** und **D**? Wieviel würden sie verbrauchen, wenn **f** nicht virtuell wäre?

```
struct A
{
    virtual f();
    int a;
};
struct B: A { int b; }
struct C: A { int c; }
struct D: B, C { int d; }
```

Antwort:

A: 8 Bytes,
B und **C:** 12 Bytes
D: 28 Bytes

Ohne **virtual**:

A: 4 Bytes,
B und **C:** 8 Bytes
D: 20 Bytes

Anmerkung: in **D** stecken somit zwei **A**-Objekte.

Eigenschaften virtueller Funktionen

dynamic binding: "Die tatsächlich aufgerufene Funktion wird erst zur Laufzeit bestimmt".

Nachteile:

- zusätzlicher Speicherbedarf pro Objekt
- langsamer, indirekter Sprung
- kein Inlining

Beobachtung: oft braucht man gar kein *dynamic binding*.

Oft ist *dynamic binding* gar nicht nötig, wenn nämlich in Wahrheit bereits zur Compilezeit feststeht, welche Funktion aufgerufen werden wird.

Template Subclassing

Eine alternative Lösung des Delegation-Problems mit *static binding*.

bisher:

```
struct Auto { /*Auto*/ };  
struct Solarmobil: Auto { /*Solar*/ };
```


Template Subclassing

Schritt 1: "Template Spezialisierung statt Ableitung"

```
template <typename T>
struct Auto { /*Auto*/ };

struct Solarmobil;
template <>
struct Auto <Solarmobil> { /*Solar*/ };
```

Solarmobil ist lediglich ein Tag, das noch nicht einmal definiert (sondern nur deklariert) werden muss. Definiert wird hingegen wie hier gezeigt eine Spezialisierung von `Auto`.

Template Subclassing (II)

Schritt 2: "Globale Funktionen statt Member Funktionen"

```
template <typename T>
void tanken (Auto<T> & obj)
{ std::cout << "Benzin"; ... }

void tanken (Auto<Solarmobil> & obj)
{ std::cout << "Sonne"; ... }

template <typename T>
void starten (Auto<T> & obj)
{ tanken(obj); ... }
```

Template Subclassing (III)

Template Subclassing löst das Delegation Problem:

```
Auto<Solarmobil> mobi;  
starten(mobi);           //"Sonne"
```

Aufgaben zum Mittwoch:

1. Aufgabe: (PRÜFUNGSAUFGABE)

Ein Tierpark hält drei Arten von Tieren: Affen, Delphine und Haie. Die Pfleger können mit den Tieren zwei Aktivitäten ausüben: Spielen und Füttern. Delphine und Haie fressen Fisch, während Affen ausschließlich Bananen als Futter bekommen. Das Spielen mit Affen und Delphinen ist unproblematisch; beim Spielen mit Haien muss man sich jedoch vorsehen und eine Schutzweste tragen. Modellieren Sie das Problem, indem Sie Klassen für Affen, Delphine und Haie schreiben. Schreiben Sie außerdem eine Funktion **tierpflege**, bei der ein Tier zuerst gefüttert wird, um anschließend mit ihm zu spielen. Wie sähe die Lösung aus im klassischen OOP? Fällt Ihnen eine einfachere Lösung mit globalen Funktionen ein?

2. Aufgabe:

Programmieren Sie einen Shared-Pointer für beliebige Zielobjekte (Klasse des Zielobjekts spezifizieren über Template-Argument!), sowie eine Klasse **RefCount**, die den Referenzzähler enthält. **RefCount** wird der Klasse des Zielobjekts als Basisklasse gegeben (alternative Varianten willkommen!).