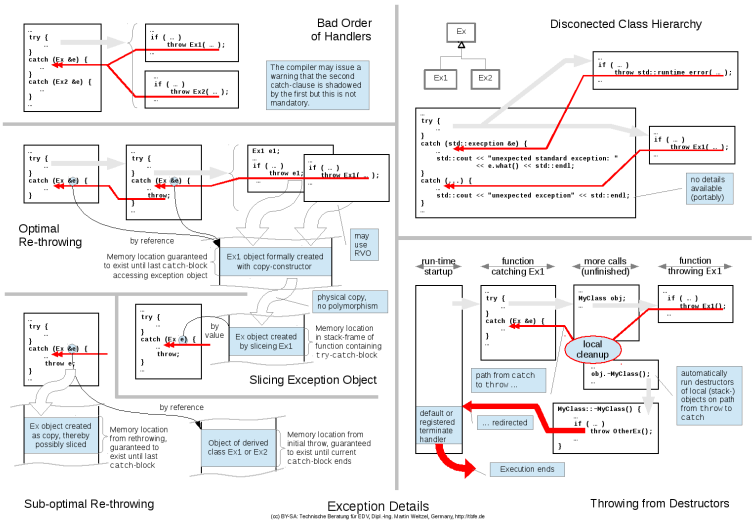
C++ FOR
(Thursday Afternoon)

Kürzere Pausen werden jeweils nach Bedarf eingelegt.

Die Besprechung der Musterlösung(en) erfolgt zu Beginn des folgenden Vormittags.

Richtlinien für den Exception-Einsatz

- (Keine) isolierte Klassenhierarchie
- (Falsche) Behandlungsblock-Abfolge
- Sub-optimale Weiterleitung
- Exceptions in Destruktoren vermeiden



(Keine) isolierte Klassenhierarchie

Bilden eigene Exceptions eine isolierte Klassenhierarchie, so kann (sehr allgemeiner) Code diese **nicht** über Standard-Exceptions abfangen:

```
int main() {
    using namespace std;
    try {
        …
        … // ordinary application
        …
        return EXIT_SUCCESS; // macro in <cstdlib>
    }
    catch (exception &e) {
        cerr << "terminated by standard exception: "
             << e.what() << endl;
    }
    catch (...) {
        cerr << "terminated by unknown exception" << endl;
    }
    return EXIT_FAILURE; // macro in <cstdlib>
}
```

(Falsche) Behandlungsblock-Abfolge

Werden die `catch`-Blöcke für Exceptions falsch angeordnet, wird

- ein weiter oben stehender **allgemeiner** Block
- einen nachfolgenden **spezifischeren** Block unwirksam machen.

Sub-optimale Weiterleitung

Sub-optimal ist grundsätzlich jede Weiterleitung, bei der das Exception-Objekt unnötig kopiert wird.

Durch Kopieren des Exception-Objekts werden geworfene Exceptions einer abgeleiteten Klasse per Slicing auf die Basisklasse reduziert.

Dies ist im Allgemeinen unerwünscht.

Daher sollte ein Kopieren vermieden werden:

- Argumente von `catch`-Blöcken als Referenz spezifizieren.
- Bei nur teilweiser Auflösung der Fehlersituation das beim Eintritt in den `catch`-Block vorliegende Exception-Objekt mit `throw;` weiterwerfen.

Optimale Weiterleitung

Nur mit in der folgenden Art und Weise wird das ursprünglich vorliegende Exception-Objekt **ohne Kopieren** erneut geworfen.

```
try {
    …
    … // code that may cause an exception
    …
}
catch (SomeException &ex) {
    …
    // only partial recovery
    throw;
}
```

Eine `throw`-Anweisung ohne nachfolgendes Argument kann nicht nur in einem `catch`-Block verwendet werden sondern auch in Funktionen, die aus einem `catch`-Block direkt oder indirekt aufgerufen werden.*

*: Gemeinsamkeiten mehrerer `catch`-Blöcke können damit auch dann problemlos in eine Hilfsfunktion ausgelagert werden, wenn diese eine komplexe Ablaufsteuerung benötigt und die gefangene Exception nicht nur am Ende weitergeworfen wird.

Mögliche Optimierungen durch den Compiler

Beim Eintritt in einen `catch`-Block führt

```
catch (SomeException ex) …
```

und beim Verlassen

```
    throw ex;
```

technisch gesehen zum Kopieren des Exception-Objekts:

- Der Compiler darf Optimierungen anwenden – unter Einhaltung der für diese Fälle im C++-Standard spezifizierten Semantik.
- Dazu gehört Slicing und die Beachtung des Zugriffsschutzes für den Kopier-Konstruktors, **nicht** aber dessen tatsächliche Ausführung.

Insbesondere ist daher beim Werfen einer Exception RVO üblich.*

*: Auch NRVO käme in Betracht, dürfte in der Praxis aber selten von Bedeutung sein, da Exception-Objekte meist beim ersten Werfen einer Exception als Argument der `throw`-Anweisung direkt erzeugt werden.

(Keine) Exceptions aus Destruktoren werfen

[!] Exceptions, die von Destruktoren geworfen werden, können ein Programm abbrechen, wenn der Destruktor im Rahmen eines Exception-Handlings ausgeführt wird.

Falls ein Destruktor Operationen benutzen muss, die u.U. eine Exception auslösen, sollte der Code sicherheitshalber in einen `try`-Block verpackt werden, der alle Exceptions fängt und ignoriert:*

```
class MyClass {
    …
    ~MyClass() {
        try {
            … // whatever needs be done
        }
        catch (...) {
            if (!std::uncaught_exception()) throw;
        }
    }
};
```

*: Ob es tatsächlich sinnvoll ist, die Exception wie gezeigt weiterzuwerfen, falls der Destruktor **nicht** im Rahmen eines Exceptions-Handlings abläuft, muss je nach Sachlage individuell entschieden werden.

C++98 `throw`-Spezifikationen

Seit vielen Jahre empfehlen von C++-Experten praktisch einhellig, die seinerzeit von C++98 eingeführten `throw`-Spezifikationen zu vermeiden.

Mit C++11 wurden `throw`-Spezifikationen zum *Deprecated Feature*.*

*: Dies bedeutet zwar, dass sie in einem künftigen C++-Standard ganz entfallen könnten, erfahrungsgemäß bieten Compiler aber selbst dann Aufrufoptionen oder `#pragma`-s, mit deren Hilfe sich Rückwärtskompatibilität erzielen lässt. Andererseits ist es aber auch denkbar, dass ein C++11 Compiler für jede Verwendung einer `throw`-Spezifikationen eine Warnung ausgibt.

C++11 noexcept-Spezifikation

Diese bewirkt folgendes:

- Wirft eine noexcept-Funktion direkt oder indirekt eine Exception, wird das Programm **unwiderruflich beendet** – wenn auch ggf. erst nach Durchlaufen eines frei festlegbaren, zentralen Handlers.
- In anderen Funktionen, welche wiederum noexcept-Funktion aufrufen, gelten letztere als **aufrufbar ohne Risiko einer Exception**.*

Insgesamt liegen mit der Umsetzung von noexcept durch gängige Compiler noch zu wenige Erfahrungen vor, um eindeutige Richtlinien und Empfehlungen zur Benutzung dieses Features geben zu können.

*: Im Kontext des Aufrufers kann das Auftreten von Exceptions bei noexcept-Funktionen in der Tat unberücksichtigt bleiben … aber nicht, weil Exceptions in noexcept-Funktionen ausgeschlossen sind, sondern weil es ggf. **statt zur Rückkehr** einer solchen Funktion zum Programmabbruch kommt.

Bedingte Exception-Freiheit

Über das auf der vorhergenden Seite Dargestellte hinaus ist `noxecept` auch eine Compilezeit-Funktion, mit welcher insbesondere Templates auch die **bedingte Freiheit von Exceptions** ankündigen können.

- Auf diese Weise kann eine Funktion zum Ausdruck bringen, dass Exceptions nur dann auftreten, wenn bestimmte von ihr aufgerufene Funktionen diese werfen.
- Dadurch kann zur Compilezeit bei der Instanziierung von Templates mit konkreten Typen entschieden werden, ob bei einer bestimmten Operation das Risiko von Exceptions besteht.
- Dies eröffnet letztlich die Möglichkeit, dass Bibliotheksfunktionen mittels Techniken der Meta-Programmierung unter verschiedenen Implementierungen wählen.

Beispiele zu noexcept

Eine einfache Berechnung – Exceptions sind völlig ausgeschlossen:

```
float fahrenheit_to_centigrade(float temperature) noexcept {
    return 9.0*temperature/5.0 + 32.0;
}
```

Wenn es zu einer Exception kommt ist der Programmabruch jeglichem Versuch einer irgendwie gearteten Fortsetzung vorzuziehen:

```
extern void may_fail_catastrophically(int = 42) noexcept;
```

Eine Exception wird von `bar` nur geworfen, wenn `T::foo` eine wirft:*

```
template<class T>
void bar(const T &arg) noexcept(noexcept(arg.foo())) {
    … // an exception can neither occur here ...
    arg.foo();
    … // ... nor can an exception occur here
}
```

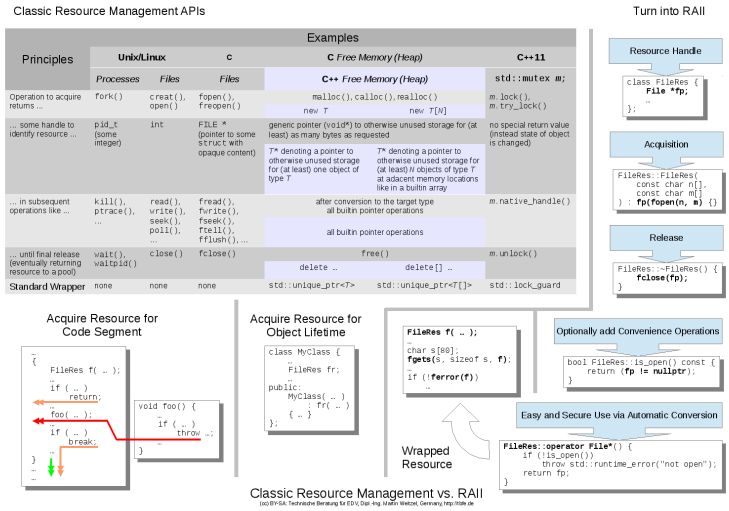
---

*: The C++11-syntax is that ugly and requires nested `noexcept`-s; for C++14 some polishing is expected.

Ressourcen ohne und mit RAII

- Klassische APIs

- Mit RAII verwaltete Ressourcen ...
- ... für Ausführung einer Anweisungsfolge ...
- ... oder Lebensdauer eines Objekts belegen



Klassische APIs

Klassisches Ressource-Management beruht auf zwei getrennten Operationen:

- Belegen der Ressource (Acquire, Allocate, Open, ...)
- Freigeben der Ressource (Release, Free, Close, ...)

Bei beiden Operationen besteht das Problem, dass sie vergessen werden könnten, bei der zweiten zusätzlich das Problem, dass sie zu früh stattfinden könnte.*

*: Zum falschen Zeitpunkt ausgeführte Operationen sind oft einer zu komplexen Programmlogik anzulasten. Die vorwiegend genutzten und daher relativ gut getesteten Ausführungspfade sind dann zwar ohne Probleme, Schwierigkeiten treten aber bei selteneren Konstellationen auf, deren Vorgeschichte – zwecks Nachvollziehung des Fehlers – zudem oft schwer reproduzierbar ist.

Um welche Ressourcen geht es?

Grundsätzlich werden unter dem Begriff hier alle

- **"knappen" Betriebsmittel**

verstanden, über die ein Programm nicht beliebig während seiner gesamten Ausführungszeit verfügen kann oder sollte:

Die folgende Liste ist nur beispielhaft zu sehen und keineswegs erschöpfend:

- Hauptspeicher
- Mutexe
- Dateien
- Prozesse
- Datenbank-Verbindungen
- ...

Wie werden Ressourcen repräsentiert?

In C/C++ gibt es zwei besonders häufig verwendete Abstraktionen, die eine Ressource repräsentieren:

- **Zeiger** – eine Speicheradresse,
  - an der wesentliche Informationen zur Ressource stehen,
  - oftmals repräsentiert durch eine Struktur,
  - deren Inhalt im Detail aber nicht von Interesse ist.
- **Handles** – in der Regel eine Ganzzahl,
  - die einer Service-Schnittstelle zu übergeben ist,
  - der gegenüber sie die Ressource repräsentiert.*

*: Mitunter – aber nicht grundsätzlich – sind Handles Indizes, mit welchen Einträge einer hinter der Service-Schnittstelle angesiedelten Tabelle ausgewählt werden.

Mit RAII verwaltete Ressourcen

Bei RAII handelt es sich um die Abkürzung der von Bjarne Stroustrup empfohlenen Technik, klassische Ressourcen zu verpacken:

Ressource **A**cquisition **I**s **I**nitialisation

In der Regel ist dafür eine – meist sehr einfache – Hilfsklasse erforderlich, welche

- die Ressource-Anforderung im Konstruktor und
- die Ressource-Freigabe im Destruktor

vornimmt.

RAII-Ressource für Anweisungsfolge belegen

Da in C++ überall neue (geschachtelte) Blöcke beginnen können, lässt sich die Anweisungssequenz, während der die Ressource belegt ist, nach Belieben festlegen.

- Eine Variable vom Typ des Ressource-Wrappers wird an der Stelle eines Code-Blocks (als Stack-Objekt) angelegt, ab dem die Ressource benötigt wird.
- Der Destruktor des Ressource-Wrappers wird automatisch ausgeführt, wenn der betreffende Code-Block verlassen wird.

> Durch die automatische Ausführung des Destruktors wird die Ressource in jedem Fall zuverlässig freigegeben.

Insbesondere spielt es keine Rolle, wie und warum der Kontrollfluss den betreffenden Code-Block verlässt, also egal ob

- nach Ausführung der letzen Anweisung ...
- ... durch vorzeitiges `return`, `break`, `continue` ...
- ... oder `throw` – auch wenn indirekt in einer aufgerufenen Funktion.

RAII-Ressource für Objekt-Lebensspanne belegen

Objekte die (per Komposition) als Teil anderer Objekte existieren, werden

- während der Erzeugung des umfassenden Objekts automatisch mit angelegt und
- während dessen Zerstörung automatisch mit zerstört.

Damit kann ein Ressource-Wrapper die Belegungsdauer einer Ressource zuverlässig an die Lebensspanne eines bestimmten (anderen) Objekts binden.

Explizite Anforderung mehrerer Ressourcen ohne RAII

Fordert eine Klasse in ihrem Konstruktor mehr als eine Ressource explizit an – **ohne Verwendung von RAII** -, besteht nur bei einer besonders sorgfältigen Vorgehensweise Sicherheit vor Ressource-Leaks.

Beispiel-Code für Klasse – kein RAII

```
class Choice {
    …
    Window *w;   // optional, if needed allocated on heap with new
    MenuItem *m; // acually an array, allocated on heap with new[]
public:
    Choice( … );
    ~Choice();
    …
};
```

Offensichtlich benötigt die Klasse `Choice` zwei Sorten von Ressourcen:

- ein `Window` (optional, also Multiplizität 0..1);
- eine mehr oder weniger große Zahl von `MenuItem`-s (Multiplizität 1..*).

Beispiel-Code Choice-Konstruktor – kein RAII

Zunächst werden beide Zeiger sicher initialisiert, dann werden die Ressourcen angefordert:

```
Choice::Choice( … )
    : w(nullptr), m(nullptr) {
    try {
        w = new Window( … );
        m = new MenuItem[…];
    }
    catch( … ) {
        delete w;   // delete is nullptr-safe, so no problem if
        delete[] m; // either of first or second new above threw
        throw;
    }
};
```

[!] Bevor der Konstruktor (fehlerfrei) beendet wurde, ist der Destruktur noch nicht aktiv – evtl. auftretende Probleme müssen daher in einem lokalen `catch`-Block behandelt werden.

Beispiel-Code Choice-Destruktor – kein RAII

Nach erfolgreichem Durchlaufen des Konstruktors wird der Destruktor aktiviert.

Dieser muss dann so aussehen:

```
Choice::~Choice() {
    delete w;    // matching kind of new easy to verify
    delete[] m;  // by quick comparison with c'tor code
}
```

[!] Genau diese beiden Anweisungen waren schon einmal nötig – im `catch`-Block des Konstruktors. Es liegt eine unschöne Duplizierung von Code vor (Verletzung des DRY-Principles).

Anforderung mehrerer Ressourcen mit RAII

Hierbei wandert die Operation zur Freigabe in den Wrapper der betreffenden Ressource:

```
class WindowRes {
    Window *res;
public:
    WindowRes( … ) : res(new Window( … )) {}
    ~WindowRes()  { … ; delete res; … }
    operator Window*() { return res; }
};


class MenuItemRes {
    MenuItem *res;
public:
    MenuItemRes( … ) : res(new MenuItem[…]) {}
    ~MenuItemRes() { … ; delete[] res; … }
    MenuItem &operator[](int i) { return res[i]; }
};
```

Beispiel-Code Vereinfachung Choice-Klasse – mit RAI

```
class Choice {
    …
    WindowRes wr;
    MenuItemResr mr;
public:
    Choice( … ) : WindowRes( … ), MenuItemRes( … ) {
        … // in case something special is necessary,
          // but NO try-catch required to handle
          // problems with ressource allocation
    }
    ~Choice() {
        … // in case something special is necessary
        // but NO d'tor required to return ressources
    }
    …
};
```

- Der Destruktor ist nicht mehr für die Ressourcen zuständig – diese werden eigenständig von ihrem jeweiligen Wrapper verwaltet.
- Er kann ganz entfallen, wenn ansonsten keine Aufräumungsarbeiten notwendig sind.

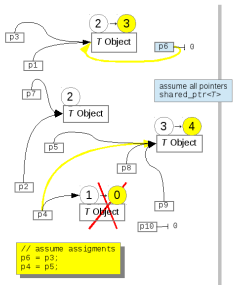
Smart-Pointer

- Smart-Pointer von C++11* im Vergleich

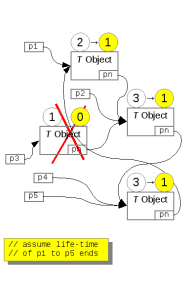
- Prinzip der Referenzzählung

- Problem der zyklischen Referenzierung
- Lediglich "beobachtende" Zeiger als Lösung
- "Verlorene" Ressourcen

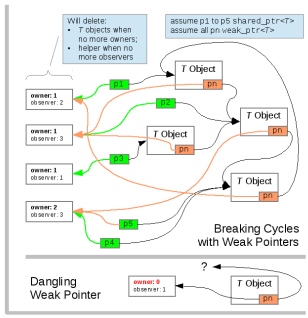
- Typische Smart-Pointer Implementierungen



Reference Counting Principle



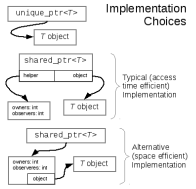
Problem of Cyclic References



Comparing ...	std::unique_ptr<T>	std::shared_ptr<T>	Remarks
Characteristic	refers to a single object of type T, uniquely owned	refers to a single object of type T, possibly shared with other referrers	may also refer to "no object" (like a nullptr)
Data Size	same as plain pointer	same as a plain pointer plus some extra space per referred-to object	
Copy Constructor	no*	yes	particularly efficient as only pointers are involved
Move Assignment	yes		
Copy Assignment	no*		a T destructor must also be called in an assignment if the current referrer is the only one referring to the object
Move Assignment	yes		
Destructor (when referrer life-time ends)	always called for referred-to object	called for referred-to object when referrer is the last (and only) one	

*: explcit use of std::move for argument is possible

Smart Pointer Comparison



*: Berücksichtigt sind hier nur die Smart-Pointer von C++11, da mit diesem Standard zugleich der mit C++98 eingeführte `std::auto_ptr` abgekündigt wurde.

Smart-Pointer von C++11 im Vergleich

Die von C++11 bereitgestellten Smart-Pointer repräsentieren (im Wesentlichen dem Vorbild von Boost.Smart_ptr folgend) die wichtigsten Beziehungen, die ein Zeiger zu einem anderen – i.d.R. aber nicht zwingend auf dem Heap angelegten – Objekt ausdrücken kann:

- Exklusive Eigentümerschaft: `std::unique_ptr`
- Geteilte Eigentümerschaft: `std::shared_ptr`
- Beobachter ohne Eigentümerschaft: `std::weak_ptr`

Den mit C++98 eingeführten `std::auto_ptr` gibt es weiterhin, langfristig und für neue Programme empfiehlt C++11 an dessen Stelle aber die Verwendung von `std::unique_ptr`.*

*: Substituting an `std::auto_ptr` with an `std::unique_ptr` should cause no major pains. If problems occur they often rather indicate sleeping bugs caused by the special semantics of an `std::auto_ptr`, that now will only become more obvious.

Exklusive Eigentümerschaft: `std::unique_ptr`

Die Verwendung dieser Art von Smart-Pointer hilft, die exklusive Eigentümerschaft für das referenzierte Objekt sicherzustellen:

- Es gibt einen *Move-Konstruktor*
- aber **keinen** *Copy-Konstruktor*.

- Es gibt ein *Move-Assignment*
- aber **kein** *Copy-Assignment*.

Geteilte Einegtümerschaft: `std::shared_ptr`

Die Verwendung dieser Art von Smart-Pointer hilft, bei geteilter Eigentümerschaft die Beseitigung referenzierter Objekte sicherzustellen.

Intern verendet die übliche Implementierung zwei Zeiger:*

- Einer zeigt auf das referenzierte Objekt,
- der andere auf ein Hilfsobjekt mit einem Referenzzähler.

Von der `std::shared_ptr`-Klasse wird in letzterem durch

- Default-Konstruktor,
- Copy- und Move-Konstruktor,
- Copy- und Move-Assignment
- sowie Destruktor

die Anzahl der Referenzierer eingetragen bzw. aktualisiert.

> Fällt der Referenzzähler auf 0, wird der Destruktur des referenzierten Objekts ausgeführt und dessen Speicherplatz freigegeben.

*: Eine alternative Implementierung benötigt nur einen Zeiger bei schlechterer Laufzeit-Performance.

Problem der zyklischen Referenzierung

Mitunter kann es notwendig werden, dass

- ein Objekt, auf das per `std::shared_ptr` verwiesen wird,
- selbst wiederum einen `std::shared_ptr` enthält, der direkt oder indirekt* auf Objekte der eigenen Art zeigt.

[!] Dies vorausgesetzt, kann es zyklische Ketten von Referenzen geben, die einen isolierten, unerreichbar gewordenen Verbund von Objekten im Speicher darstellen, dessen Teile nur noch untereinander über `std::shared_ptr` verbunden sind.

*: Die Kernproblematik besteht also nicht nur, wenn ein `std::shared_ptr` **exakt** auf die Art von Objekten zeigt, die ihn enhtalten. Auch bei der Konstellation "*A zeigt auf B und B auf A*" oder "*A zeigt auf B, B zeigt auf C und C zeigt auf A*" kann das beschriebene Problem auftreten.

Nutzer ohne Eigentümerschaft: `std::weak_ptr`

Mittels `std::weak_ptr` lassen sich zyklische Referenzen vermeiden.

Bei einem Design unter Einbeziehung von `std::weak_ptr` ist hinsichtlich der Referenzierer eines Objekts zu prüfen, ob es sich

- um (echte) Ressource-Eigentümer handelt
- oder lediglich um Ressource-Beobachter.*

> **[!]** Letztere müssen damit rechnen, dass sie die beobachtete Ressource "verlieren".

Das geschieht genau dann, wenn es für die beobachtete Ressource keine wirklichen Eigentümer mehr gibt sondern nur noch Beobachter.

*: In der typischen Implementierung des `std::shared_ptr` enthält ein zweiter Zähler im Hilfsobjekt die Summe der Anzahl von Eigentümern und Beobachtern. Fällt diese auf 0, kann auch der Speicherplatz für das Hilfsobjekt freigegeben werden.

"Verlorene" Ressourcen

Da die `std::weak_ptr` die beobachtete Ressource verlieren können, unterstützen sie keinen direkten Zugriff darauf mit `*` oder `->`.

Zunächst ist aus dem `std::weak_ptr` ein `std::shared_ptr` zu erstellen.*

Analog zum `dynamic_cast` werden zwei Vorgehensweisen unterstützt:

- Erste Vorgehensweise:
  - Initialisieren eines `std::shared_ptr` und nachfolgender Test;
  - ergibt dieser, dass der Zeiger auf kein Objekt gültiges Objekt zeigt, existiert das beobachtete Objekt nicht mehr.
- Zweite Vorgehensweise:
  - Rückgabe eines ohne Prüfung benutzbaren `std::shared_ptr`;
  - wenn das ursprünglich beobachtete Objekt nicht mehr existiert, wird stattdessen eine Exception geworfen.

*: Genau dieser `std::shared_ptr` ist es dann, der das referenzierte Objekt ggf. am Leben hält, auch wenn die anderen Eigentümer nun verschwinden sollten.

Typische Smart-Pointer Implementierungen

`std::shared_ptr` – meist verwendete Variante

Hierbei enthält ein `std::shared_ptr` (mindestens*) **zwei Zeiger**, von denen einer auf das referenzierte Objekt und der andere auf ein Helfer-Objekt zeigt.

Das Helfer-Objekt wiederum enthält **zwei Zähler**, von denen einer die Eigentümer, der andere (typisch) beides, Eigentümer *plus* Beobachter.

Damit entscheidet der

- erste Zähler darüber, wann das **referenzierte Objekt** und
- der zweite Zähler darüber, wann das **Hilfsobjekt**

freigegeben wird.

---

*: Wenn ein `std::shared_ptr` über einen Basisklassen-Zeiger auf eine virtuelle Basis-Klasse eines Objekts zeigt, das von dieser Klasse abgeleitet ist, kann zum effizienten Zugriff auf das referenzierte Objekt noch ein weiterer Zeiger erforderlich werden.

`std::shared_ptr` – alternative Implementierung

In einer alternativen Implementierung enthält ein `std::shared_ptr` nur **einen** Zeiger auf ein Helfer-Objekt, das neben den Referenzzählern nun auch einen Zeiger auf das referenzierte Objekt enthält.

`std::weak_ptr` – typische Implementierung

Die Implementierung entspricht im Wesentlichen dem der `std::shared_ptr`, also

- zwei Zweiger – auf referenziertes Objekt und Helfer-Objekt, **oder**
- ein Zeiger – auf Helfer-Objekt und von dort weiter zum referenzierten Objekt.

`std::unique_ptr` – typische Implementierung

Solange die Implementierung keinen *Stateful Custom Deleter* unterstützt reicht **ein** klassischer Zeiger – sonst sind u.U. **zwei** Zeiger erforderlich.*

*: But then advanced meta-programming techniques can help to save that second pointer for any `std::unique_ptr` that **makes no use** of a custom deleter or when the one used **is not stateful**.

Praktikum