C++ FOR (Freitagvormittag)

---

1. Ressourcen ohne und mit RAII verwalten
2. Smart-Pointer
3. Übung

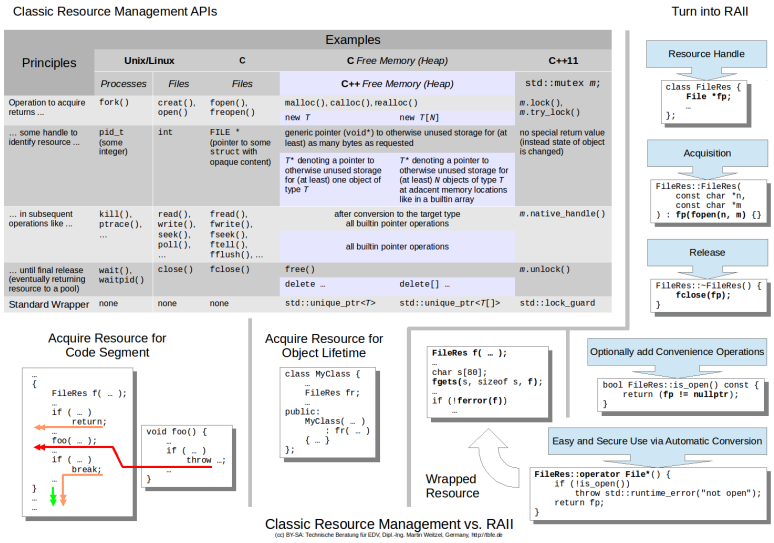
Kürzere Pausen werden jeweils nach Bedarf eingelegt.

> Die Besprechung der Musterlösung(en) erfolgt direkt nach der Übung, da am letzten Kurstag der Nachmittagsteil entfällt.

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 seltenen 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 während seiner gesamten Ausführungszeit verfügen kann.

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 und dieser gegenüber 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 Ablürzung der von Bjarne Stroustrup empfohlenen Technik, klassische Ressourcen zu verpacken:

> Ressource Acquisition Is Initialisation

In der Regel ist dazu eine kleine 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, vorzeitiges `return`, `break`, `continue` oder `throw`, letzteres auch in 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.

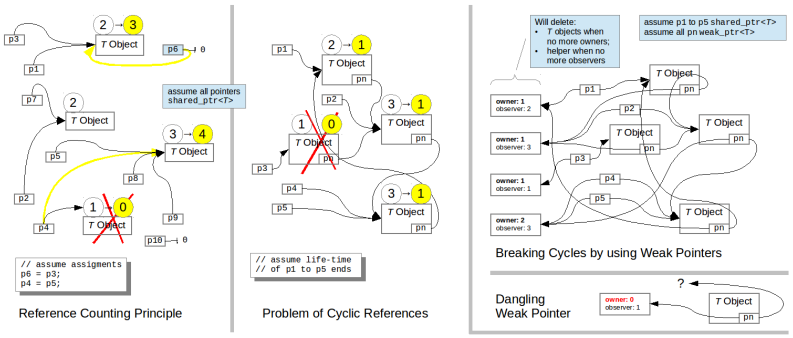
Smart-Pointer

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

- Prinzip der Referenzzählung

- Problem zyklischer Referenzierung mit ...
- ... "nur beobachtenden" Zeiger als Ausweg und ...
- ... "verlorene" Ressourcen

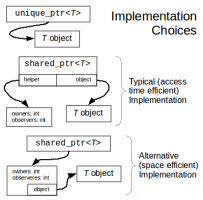
- Varianten der Implementierung



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 last 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 a possible work-around

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 die beiden wichtigsten Beziehungen, die ein Zeiger (evtl. als Bestandteil eines Objekt) zu auf dem Heap angelegten (anderen) Objekt haben kann:

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

Den mit C++98 eingeführte `std::auto_ptr` gibt es in C++11 zwar weiterhin, er sollte in neuen Programmen aber nicht mehr verwendet werden.*

*: Substituting an `std::auto_ptr` with an `std::unique_ptr` should cause no major pains … if any problems occur they may rather indicate sleeping bugs caused by the special behavior of an `std::auto_ptr` and will now only become more obvious with an `std::unique_ptr`.

Exklusive Eigentümerschaft: `std::unique_ptr`

Die Verwendung dieser Art von Smart-Pointer hilft dabei, die exklusive Eigentümerschaft für das referenzierte Objekt zu garantieren.

- 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 dabei, die geteilte Eigentümerschaft zu garantieren. Ihre Objekte enhalten jeweils *zwei* Zeiger:*

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

Von

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

der `std::shared_ptr`-Klasse wird der Referenzzähler gemäß der Anzahl der Referenzierer des über den Zeiger erreichbaren Objekts verwaltet.

> Fällt der Stand des Referenzzählers im Hilfsobjekt von 1 auf 0, wird der Destruktur des referenzierten Objekts ausgeführt.

*: Dies gilt für die typische Implementierung, eine etwa sparsamere Implementierung erfordert nur einer Zeiger, hat aber eine schlechtere Performance zur Laufzeit.

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 auf Objekte der eigenen Art zeigen kann.*

[!] 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.

*: Es müssen nicht zwingend Objekte der eigenen Art sein, 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 klarkommen, dass ihnen die Ressource "entzogen" wird, wenn es dafür keine anderen ("echten") Eigentümer mehr gibt.

"Verlorene" Ressourcen

Da die `std::weak_ptr` lediglich Beobachter sind, unterstützen sie keinen direkten Zugriff zur referenzierten Ressource (mit `*` oder `->`).

Vielmehr muss durch Aufruf der Member-Funktion `lock` zunächst ein `std::shared_ptr`* geholt werden und ähnlich wie beim `dynamic_cast` werden die zwei häufigsten Verwendungen gezielt unterstützt:

- Erste Vorgehensweise:
  - Rückgabe eines Zeigers und nachfolgend erforderlicher Test,
  - wenn dieser ergibt, dass der Zeiger ein `nullptr` ist, existiert das ursprünglich beobachtete Objekt nicht mehr.
- Zweiten Vorgehensweise:
  - man erhält eine gültige Referenz (die ohne weitere Prüfungen nutzbar ist),
  - wenn das ursprünglich beobachtete Objekt nicht mehr existiert, wird eine Exception geworfen.

---

*: Dieser `std::shared_ptr` ist es dann, der das Objekt ggf. am Leben hält, wenn die anderen Eigentümer zwischenzeitlich 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 das Helfer-Objekt zeigt.

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

- Der erste Zähler entscheidet daraüber, wann das referenzierte Objekt freigegeben werden kann,
- der zweite Zähler entscheidet darüber, wann das Hilfsobjekt freigegeben werden kann.

---

*: 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, welches außer dem Referenzzählern einen Zeiger auf das referenzierte Objekt enthält.

`std::weak_ptr` - typische Implementierung

Die Implementierung entspricht im Wesentlichen dem der `std::smart_ptr`, also zwei Zweiger (auf referenziertes Objekt und Helfer-Objekt) oder nur ein Zeiger (auf Helfer-Objekt und von dort weiter zum referenzierten Objekt).

`std::unique_ptr` - typische Implementierung

Hierfür ist in der Regel ein klassischer Zeiger ausreichend.*

---

*: If the `std::unique_ptr` implementation supports a "stateful custom deleter" another pointer will be required.

Übung

Ziel der Aufgabe:

Entwicklung eines Ressource-Wrappers im RAII-Stil zum C `FILE`-API.

> Weitere Details werden vom Dozenten anhand des Aufgabenblatts sowie der vorbereiteten Eclipse-Projekte erläutert.