C++ FOR (Freitagvormittag)

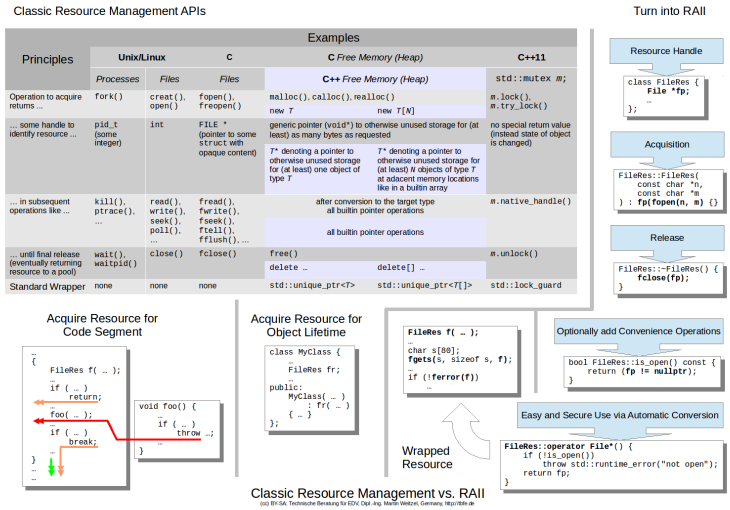
Kürzere Pausen werden jeweils nach Bedarf eingelegt.

> Die Besprechung der Musterlösung(en) erfolgt nach direkt nach der Übung, da am letzten Kurstag der Nachmittag 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 in aller Regel einer (zu) komplexen Programmlogik anzulasten. Die vorwiegend genutzten und gut getesteten Ausführungspfade sind dann ohne Probleme, diese treten nur relativ selteneren Konstellationen mit oft unklarer Vorgeschichte auf.

Um welche Ressourcen geht es?

Grundsätzlich werden unter dem Begriff hier alle "knappen Betriebsmittel" verstanden, über die ein Programm nicht während seiner gesamten Ausfhrungszeit 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-Resource 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 Resource-Wrappers wird an der Stelle eines Code-Blocks (als Stack-Objekt) angelegt, ab dem die Ressource benötigt wird.
- Der Destruktor des Resource-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 im Block, `return`, `break`, `continue` oder `throw` (letzteres auch in aufgerufener Funktion).

RAII-Resource für Objekt-Lebensspanne belegen

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

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

Damit kann ein Resource-Wrapper die Belegungsdauer einer Resource auch zuverlässig mit der Lebensspanne eines bestimmten (anderen) Objekts verknüpfen.

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 Resource-Leaks.

Beispiel-Code für Klasse - kein RAII

Angenommen eine Klasse `Choice` benötigt zwei Sorten von Ressourcen:

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

```
class Choice {
 …
 Window *w;
 MenuItem *m;
public:
 Choice( … );
 ~Choice();
 …
};
```

Explizite Anforderung mehrerer Ressourcen ohne RAII (2)

Beispiel-Code Konstruktor für Klasse - kein RAII

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

```
Choice::Choice( … )
 : w(nullptr), m(nullptr) {
 try {
  w = new Window( … );
  m = new MenuItem[…];
 }
 catch( … ) {
  delete w;
  delete[] m;
  throw;
 }
};
```

Explizite Anforderung mehrerer Ressourcen ohne RAII (3)

Beispiel-Code Destruktor für Klasse - kein RAII

Nach erfolgreichem Durchlaufen des Konstruktors wird der Destruktor aktiviert. Dieser muss dann so aussehen:

```
Choice::~Choice() {
 delete w;
 delete[] m;
}
```

[!] 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]; }
};
```

Anforderung mehrerer Ressourcen mit RAII (2)

Beispiel-Code Vereinfachung Choice-Klasse - mit RAI

```
class Choice {
 …
 WindowRes wr;
 MenuItemResr mr;
public:
 Choice( … ) : WindowRes( … ), MenuItemRes( … ) {}
 // (eigener) Destruktor entfaellt
 …
};
```

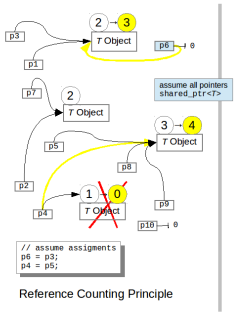
Smart-Pointer

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

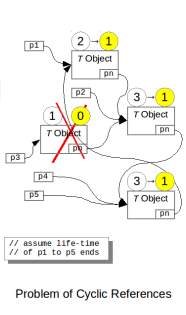
- Prinzip der Referenzzählung

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

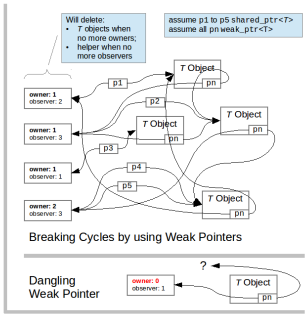
- Varianten der Implementierung



Reference Counting Principle



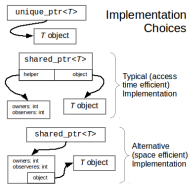
Problem of Cyclic References



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 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 Objekt zu auf dem Heap angelegten Speicherbereich haben kann:

- Exklusive Eigentümerschaft
- Geteilte Eigentümerschaft

`std::unique_ptr`

Die Verwendung dieser Art von Smart-Pointer hilft dabei, die exklusive Eigentümerschaft zu garantieren:

- Sie verfügt über einen *Move-Konstruktor*,
- aber über **keinen** *Copy-Konstruktor*!
- Sie verfügt über ein *Move-Assignment*,
- aber über **kein** Copy-Assignment.

`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 refernzierte Objekt,
- der andere auf einen Referenzzähler.

In Default-Konstruktor, Copy- und Move-Konstruktor, Copy- und Move-Assignment sowie Destruktor der `std::shared_ptr`-Klasse werden die Referenzzähler gemäß der Anzahl der Referenzierer eines Objekts verwaltet.

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

*: Dies gilt für die typische Implementierung - in einer alternativen Implementierung ist nur ein Zeiger erforderlich, zum Zugriff muss dann aber eine zusätzliche Dereferenzierung erfolgen - also der klassiche Austauch von (weniger) Speicherplatz gegen (mehr) Laufzeit.

Problem zyklischer Referenzierung

Mitunter kann es notwendig werden, dass

- dass ein Objekt, auf das per `std::shared_ptr` verwiesen wird,
- selbst wiederum einen `std::shared_ptr` enthält, welcher auf Objekte der eigenen Art zeigen kann.*

Dies vorausgesetzt, kann es zyklische Ketten von Referenzen geben, womit irgendwann ein isolierter, unerreichbar gewordenen Verbund von Objekten im Speicher verbleiben könnte, welche aller über `std::shared_ptr` miteinander verbunden aber von außen unerreichbar sind.

*: Es müssen nicht zwingend Objekte der eigenen Art sein, auch in der Konstellation "A zeigt auf B und B auf A" oder "A zeigt auf B, B zeigt auf C und C zeigt auf A" usw. kann das Problem auftreten.

Lediglich "beobachtende" Zeiger als Ausweg

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

Dabei ist hinsichtlich aller Referenzierer zu prüfen, ob es sich

- um (echte) Ressource-Eigentümer handelt
- oder lediglich um Resource-Beobachter?

Letzteren kann die Resource "entzogen" werden, wenn es keine (echten) Ressource-Eigentümer mehr gibt.

"Verlorene" Resourcen

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` des `std::weak_ptr` 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 diese ergibt, dass der Zeiger ein `nullptr` ist, wurde das beobachtete Objekt bereits weggeräumt.

Zweiten Vorgehensweise

- man erhält eine gültige Referenz, die sofort und ohne weitere Prüfungen nutzbar ist,
- oder es wird eine Exception geworfen, wenn das beobachtete Objekt bereits weggeräumt wurde.

Varianten der Implementierung

In der alternativen Implementierung zeigen die Objekte der Klasse `std::shared_ptr` auf ein Helfer-Objekt, das den Referenzzähler sowie einen Zeiger auf das eigentlich referenzierte Objekt enthält.

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.