

MODERNES C++

LEAK-FREI "BY DEFAULT"

by
TOBIAS LANGNER

ETAS GmbH
Software Architect

ERFAHRUNG

- 4 Jahre Delphi
- 6 Jahre C++



Jason Turner

@lefticus

 Follow



C++ has what very few other languages have:
well defined object life cycle.

You **must** appreciate this if you want to
understand [#cplusplus](#)

RETWEETS

43

LIKES

114



11:56 AM - 6 Feb 2017



5



43



114

IST ENTHALTEN

was ist der Standard für eine "ist enthalten" Beziehung?

Member by value

IST ENTHALTEN

```
class Foo {  
    Bar member_  
};
```

IST ENTHALTEN

```
class Foo {  
    Bar member_;  
public:  
    Foo() = default; //default constructor  
    Foo(const Foo&) = default; //default copy constructor  
    Foo(Foo&&) = default; //default move constructor  
  
    Foo& operator=(const Foo&) = default; //default copy assignment operator  
    Foo& operator=(Foo&&) = default; //default move assignment operator  
};
```

MEMBER BY VALUE

VORTEILE:

- einfach
- schnell

NACHTEILE:

- Größe des Objektes muss bekannt sein
- keine "polymorphen" Member
- Stackgröße limitiert

GRÖSSE DES OBJEKTES MUSS BEKANNT SEIN

da der Compiler die Größe jeder class/struct berechnen können muss, darf eine class/struct nur Elemente als member enthalten, deren Größe bekannt ist.

KEINE "POLYMORPHEN" MEMBER

Laufzeit-Polymorphismus funktioniert nur über Zeiger

STACK

```
void foo() {  
  Bar b;  
}
```

legt die Variable b vom Typ Bar auf dem Stack an

STACK

- sehr schnell (Speicher auf dem Stack ist nur eine Subtraktion)
- sehr Cache-Effizient (die Spitze des Stacks liegt eigentlich immer im Cache)
- sehr limitiert (Stack ist normalerweise 1 Mbyte groß)

MEMBER BY VALUE

- Member "by value" sollte der "Normalfall" sein (80%).
- Wenn er nicht ausreicht, muss man mit Zeigern und dem Heap arbeiten

EINSCHUB: VALUE TYPES

VALUE TYPES

sind dazu gedacht "by value" verwendet zu werden.

```
class Foo {  
    std::string name_; // "by value" member, no reference, no pointer  
}  
  
int myFunc(std::vector<int> data) { // parameter "by value"  
    int retval=0; // also usage "by value"  
    std::accumulate(data.cbegin(), data.cend(), retval);  
    return retval;  
}
```

VALUE TYPES

sollten sich wie "int" verhalten

- kopierbar
- nicht polymorph
- Default-Constructible wo möglich

VALUE TYPES

- kleine Objekte (z.B. Point) haben typischerweise ihre Member auch "by value"
- große bzw. dynamische Objekte (z.B. String) haben nur Verwaltungsinformationen (z.B. Pointer auf den Stack und Größe) im intern und lagern die eigentlichen Daten auf den Heap aus

VALUE TYPES

Es gibt spezielle Optimierungen von denen Value-Typen besonders profitieren bzw. die erst dadurch möglich sind:

- Return-Value-Optimization (Named & Unnamed)
- Immutable objects können die Daten über alle Instanzen teilen
- Copy-On-Write (z.B. `std::string` illegalerweise in alten glibc Versionen)

EINSCHUB: RAI

**WAS UNTERSCHIEDET C++ VON ALLEN
PROGRAMMIERSPRACHEN?**

}

wenn ein Block mit einer } geschlossen wird, werden die **Destruktoren** der **lokalen** Variablen des Blocks automatisch ausgeführt

RAII

RESOURCE ACQUISITION IS INITIALIZATION

Ein Programmier-Muster in C++ bei dem eine Resource direkt einem Verwaltungsobjekt mit Wert-Semantik übergeben wird.

RAII

- RAII ist ein **eleganter** weg um **Ressourcen** in C++ zu verwalten.
- Es ist der **einzigste wartbare** Weg wenn mit Exceptions gearbeitet wird.

BEISPIEL MUTEX

```
std::mutex g_mutex;  
  
void foo() {  
    std::lock_guard lock(g_mutex); // now locked until the }  
    ...  
} //lock will be unlocked regardless how the block is left
```


STD::UNIQUE_PTR<T>

STD::UNIQUE_PTR<T>

```
#include <memory>

class Foo {
    std::unique_ptr<Bar> member_;
};
```

- Smart Pointer
- Nicht kopierbar
- aber "moveable"
- ruft bei seiner Zerstörung "delete" für die Klasse auf

STD::UNIQUE_PTR<T>

- funktioniert ähnlich wie ein "member by value"
- zeigt üblicherweise auf ein Objekt auf den Heap
- funktioniert in STL containern
- lässt sich mit std::move "übergeben"

```
class Foo {  
    std::unique_ptr<Bar> data_;  
public:  
    void SetData(std::unique_ptr<Bar> data) {  
        data_ = std::move(data);  
    }  
};
```

STD::UNIQUE_PTR<T>

Durch das Design als nicht kopierbares Objekt ist sichergestellt, dass jeweils nur ein `std::unique_ptr<T>` die Resource besitzt. Dadurch kann die Resource durch den `std::unique_ptr<T>` bei dessen Zerstörung freigegeben werden.

Das Standardverhalten ist es `delete` aufzurufen.

STD::UNIQUE_PTR<T>

Typische Datenstrukturen für einen `std::unique_ptr<T>` sind:

- Liste
- Baum

VORSICHT:

Bei langen Listen / tiefen Bäumen kann es zu einem Stackoverflow kommen wenn die Destruktoren rekursiv aufgerufen werden.

Empfehlung: in diesem Fall die Ressourcen aus dem unique_ptr manuell löschen

```
data_ = nullptr;
```

AZYKLISCHE GRAPHEN?

funktionieren nicht mit `std::unique_ptr<T>!`

STD::SHARED_PTR<T>

STD::SHARED_PTR<T>

```
#include <memory>

class Foo {
    std::shared_ptr<Bar> member_;
};
```

- Smart Pointer
- **kopierbar**
- implementiert thread-safe Referenzzählung
- Zugriffsoperator nicht thread-safe
- wenn die Referenzzahl auf 0 geht, wird das Objekt zerstört

STD::WEAK_PTR<T>

- ist ein Verweis auf einen `std::shared_ptr<T>`
- erhöht die Referenzählung nicht
- kann mittels "lock()" den `std::shared_ptr<T>` wieder erzeugen

```
std::weak_ptr<Bar> weak = mySharedPtr;  
...  
//tmpShared is a shared_ptr  
//and points either to:  
//1. the object mySharedPtr pointed to  
//2. or nullptr if the object was freed  
auto tmpShared = weak.lock();
```

STD::SHARED_PTR<T>

Der `std::shared_ptr<Bar>` allokiert für den Referenzzähler einen sogenannten "Kontrollblock". Dieser enthält die 2 Referenzzähler (Anzahl `shared_ptr` & Anzahl `weak_ptr`).

UNIQUE_PTR VS. SHARED_PTR

UNIQUE_PTR

- 0 overhead gegenüber Zeiger
- Genau 1 Besitzer
- nicht kopierbar

SHARED_PTR

- Extra Speicher für Kontrollblock
- üblicherweise 2 Pointer groß
- Mehrere "Besitzer" möglich
- kann kopiert werden
- kopieren ist langsam (thread-safe)

STD::SHARED_PTR<T>

Typische Datenstrukturen:

- DAGs (directed acyclic graph)

ALLES GUT?

STD::SHARED_PTR<T>

der shared_ptr braucht 2 Allokationen (2x new) beim Erstellen. Einmal für das Objekt, einmal für den Kontrollblock. Geht das nicht besser?

Ja! std::make_shared

```
class IntHolder {
    int data_;
public:
    IntHolder(int data);
};

void foo() {
    ...
    //creates a std::shared_ptr<IntHolder>
    //by calling IntHolder's constructor with parameter 5
    auto shared = make_shared<IntHolder>(5);
    ...
}
```

STD::MAKE_SHARED

Vorsicht: Weil `make_shared` den Kontrollblock mit dem Objekt anlegt, kann der Speicher für den Kontrollblock auch nur mit dem Objekt freigegeben werden!
D.h. erst wenn die letzten `shared_ptr` & `weak_ptr` weg sind!

ALLES GUT?

WO IST DAS PROBLEM?

```
...  
myFunc(unique_ptr(new Foo()), unique_ptr(new Bar()));
```

EIN POTENTIELLES LEAK!

Der Code enthält ein potentiell Leak!. Dem Compiler ist es gestattet die "new-Aufrufe" von den Konstruktor-Aufrufen zu trennen. Eine Mögliche Aufrufreihenfolge ist:

1. Speicher für Foo anfordern
2. Konstruktor von Foo aufrufen
3. Speicher für Bar anfordern
4. Konstruktor für Bar anfordern
5. Konstruktor für unique_ptr von Foo aufrufen
6. Konstruktor für unique_ptr von Bar aufrufen
7. myFunc aufrufen

EIN POTENTIELLES LEAK!

1. Speicher für Foo anfordern
2. Constuktor von Foo aufrufen
3. Speicher für Bar anfordern
4. **Construktor für Bar anfordern** <-- Exception
5. Construktor für unique_ptr von Foo aufrufen
6. Construktor für unique_ptr von Bar aufrufen
7. myFunc aufrufen

EIN POTENTIELLES LEAK!

Wenn Schritt 4 eine Exception auslöst, wird das in Schritt 2 erstellte Objekt leaked!

LÖSUNG

```
myFunc(make_unique<Foo>(), make_unique<Bar>());
```

analog natürlich auch für `shared_ptr` & `make_shared`. Falls man `make_shared` nicht verwenden kann (z.B. wegen der `weak_ptr`), dann sollte man `make_unique` nehmen und den `unique_ptr` in einen `shared_ptr` umwandeln.

WAS GEHT NICHT MIT DIESEN WERKZEUGEN?

Graphen mit Zyklen

GRAPHEN MIT ZYKLEN

- In diesem Fall gibt es (aktuell) keine einfache Lösung
- Normalerweise erzeugt man ein externes Verwaltungsobjekt das die Knoten besitzt und regelmäßig prüft welche Knoten noch erreichbar sind und welche freigegeben werden können.
- Es gibt einen theoretischen Ansatz das in eine Library zu packen

WIE SIEHT MODERNES C++ AUS?

- RAII für alle Ressourcen
 - Smartpointer (z.B. `unique_ptr` / `shared_ptr`) für die Speicherverwaltung
- Rule of 0
- kein `new` / `delete`



Peter Sommerlad

@PeterSommerlad

Fo

@rainer_grimm @meetingcpp i would say
user-level C++ code new and delete are

🌐 Original (Englisch) übersetzen

RETWEET

1

GEFÄLLT

3



22:35 - 7. Dez. 2016





Antwort an @PeterSommerlad @rainer_grimm @meetingcpp



Ito @mt19937_64 · 8. Dez. 2016

@PeterSommerlad @rainer_grimm @meetingcpp make_shared can be a good choice if working with weak_ptr though.

🌐 Original (Englisch) übersetzen



1



Peter Sommerlad @PeterSommerlad · 8. Dez. 2016

@mt19937_64 @rainer_grimm @meetingcpp make_unique<T>().sh

🌐 Original (Englisch) übersetzen



1



Leak freedom: In order of preference

| Strategy | Natural examples | Cost | Rough frequency |
|--|--|---|-------------------|
| Defer scoped lifetime by default (e.g., members) | Local and member objects – directly owned | Zero: Tied directly to another lifetime | O(80% of objects) |
| We prefer make_unique & weak_ptr or a container , if the object must have its own lifetime (i.e., heap) and ownership can be unique w/o owning cycles | Implementations of trees, lists | Same as new/delete & malloc/free Automates simple heap object use in a library | |
| We prefer make_shared & std_ptr , if the object must have its own lifetime (i.e., heap) and shared ownership w/o owning cycles | Node-based DAGs, incl. trees that share out references | Same as manual reference counting (RC) Automates shared object use in a library | O(20% of objects) |
| We defer destruction and have to trace unreachable objects (e.g., eventually automate in a library?) | Graphs, real-time code, limited stacks | Deferred destructors and tracing logic | |

CREDITS

Herb Sutter's "Modern C++ Series"

1. Back To The Basics!
2. Writing Good C++14 ... By Default
3. Leak Freedom in C++ ... By Default

DANKE

BONUS: MODERNES C++ - PARAMETERÜBERGABE

VALUE-TYPEN

- sollten der default sein
- Können dank
 - Move Semantik
 - besserer RVO

deutlich häufiger zum Einsatz kommen

RAW-POINTER & REFERENZEN

- haben trotz Smartpointer weiter ihre Berechtigung
- wenn ein Objekt benutzt aber nicht behalten werden darf
- und für legacy APIs

SMARTPOINTER

- werden verwendet um die Besitzrechte zu übergeben
- `std::unique_ptr<T>` wenn ein Objekt übergeben wird
- `std::shared_ptr<T>` wenn eine Funktion die geteilten Rechte übernimmt
- `(const) std::shared_ptr<T>&` wenn die Funktion geteilte Rechte übernehmen kann

MODERNES C++ - PARAMETERÜBERGABE

Durch die Möglichkeiten von C++14/11 hat C++ mehr Standardmittel bekommen um schon in der API auszudrücken, wie die Objekte verwendet werden können.

BONUS: QT-UNIVERSE

SMART-POINTER IN QT

1. QPointer
2. QScopedPointer
3. QSharedPointer
4. QWeakPointer
5. QSharedDataPointer
6. QExplicitlySharedDataPointer

QPOINTER

- "provides guarded pointers to QObject"
- is automatically set to 0 in case the referenced object is destroyed
- works internally with the QObject reference counts & a QWeakPointer

QSCOPEDPOINTER

- deletes the allocated object upon its own destruction
- non-copyable
- takes alternate destruction method via additional type
- multiple other deletion strategies are predefined (e.g. `QScopedPointerObjectDeleteLater`)
- roughly equivalent to a `std::unique_ptr<T>`
- does **not** provide move semantics

QSHAREDPOINTER & QWEAKPOINTER

- provides atomic reference counting
- QWeakPointer can be used to break reference cycles
- roughly equivalent to a `std::shared_ptr<T>`

Q-EXPLICITLY-SHAREDDATAPOINTER

- reference counting for classes derived from QSharedData
- provides *intrusive* reference counting
- thread-safe
- the explicit class detaches only if detach() is called explicitly