C++: Die Regel der Drei / Die Regel der Fünf

Zusammenfassung

Copy Constructor, Copy Assignment Operator & Destructor in C++, Transkript aus stackoverflow Thread[1] und Wikipedia[2]

Die Regel der Drei

...wenn **Copy Constructor**, **Destructor** oder **Copy Assignment Operator** als explicit deklariert wird, muss man mit hoher Sicherheit alle Drei als explicit deklarieren.

Original: The Rule of Three claims that if one of these had to be defined by the programmer, it means that the compiler-generated version does not fit the needs of the class in one case and it will probably not fit in the other cases either.

Die Regel der Fünf

Durch die C++ 11 Erweiterung in Bezug auf die „move-semantic“ wird die Regel der Drei erweitert auf:

- destructor
- copy constructor
- move constructor
- copy assignment operator
- move assignment operator

[1] https://stackoverflow.com/questions/4172722/what-is-the-rule-of-three
[2] https://en.wikipedia.org/wiki/Rule_of_three_%28C%2B%2B_programming%29

1 Einführung / Problemstellung

```
1  class Person
2  {
3          private:
4                  string name;
5                  int age;
6          public:
7                  person(const string& name, int age) : name(name),
                       age(age)
8                  {
9                  }
10 };
11
12 int main()
13 {
14     Person a("Bjarne Stroustrup", 60);
15     Person b(a);    // What happens here?
16     b = a;          // And here?
17 }
```

Die Grundfrage die sich stellt ist, was heißt es eine "Kopie" eines Objektes anzulegen, die vom Compiler generierte „Kopier-Funktionalität" kopiert einen Pointer / eine Referenz in dem sie einfach die Speicheradresse kopiert, dies führt dazu das die Kopie nur eine Referenz auf das selbe Objekt ist - was wir aber in diesem Fall wollen ist eine autarke Kopie des Objektes - genauer der Datenkomponenten, das oben angeführte Beispiel zeigt 2 "Kopier-Szenarien" einerseits, den Kopier-Konstruktor **Person b(a)**, dieser hat die Aufgabe eine neues Objekt aus den Daten des bestehenden Objektes zu konstruieren andererseits den Kopier-Zuweisungsoperator **b = a**, dieser ist schwieriger zu realisieren da sich das Objekt auf der linken Seite bereits in einem gültigen Zustand befindet.

Nachdem wir weder Kopier-Konstruktor noch den Assignment Operator definiert haben werden diese implizit vom Compiler gestellt:

Auszug aus dem C++ Standard (Section 12 §1)

The [...] copy constructor and copy assignment operator, [...] and destructor are special member functions. [Note: **The implementation will implicitly declare these member functions for some class types when the program does not explicitly declare them**. The implementation will implicitly define them if they are used ...

2 Implizite Definition

```
1  // 1. copy ctor
2  Person(const person& that) : name(that.name), age(that.age)
3  {
4  }
5
6  // 2. copy assignment operator
7  Person& operator=(const person& that)
8  {
9      name = that.name;
10     age = that.age;
11     return *this;
12 }
13
14 // 3. destructor
15 ~Person()
16 {
17 }
```

Die „Drei“ implizit definiert. Die Kopie klont unsere Datenkomponenten und legt uns eine neue unabhängige Person an, der Destruktor bleibt leer. Das ist das „Geschenk des Compilers“ - das passiert wenn man diese „Special Member Functions“ nicht selbst angibt. Alles „Fun and Games“ bis zu dem Moment wo eine Datenkomponente nicht ein einfaches „Primitive“ bzw. Skalar ist. Und wie wir inzwischen wissen: Regel der Drei/Fünf, sobald wir einen anlegen müssen - müssen (oder zumindest sollten :P) wir die anderen auch anlegen.

Auszug aus dem C++ Standard (Section 12.8)

§16 The implicitly-defined copy constructor for a non-union class X performs a memberwise copy of its subobjects.

§30 The implicitly-defined copy assignment operator for a non-union class X performs memberwise copy assignment of its subobjects

3 Dynamische Datenobjekte

Hier fängt jetzt die Krux an, nehmen wir das selbe Beispiel - nur anstatt eines Strings nehmen wir jetzt, der „Einfachheit“ halber, ein char-Array.

```
1  class Person
2  {
3          char* name;
4          int age;
5          public:
6          // the constructor acquires a resource:
7          // in this case, dynamic memory obtained via new[]
8          Person(const char* the_name, int the_age)
9          {
10                 name = new char[strlen(the_name) + 1];
11                 strcpy(name, the_name);
12                 age = the_age;
13         }
14
15         // the destructor must release this resource via delete[]
16         ~Person()
17         {
18                 delete[] name;
19         }
20 };
```

Problem Hier wird wieder „member-wise“ kopiert, das heißt aber bei dem Char-Array, dass der Skalar-Wert kopiert wird und dieser beinhaltet nicht die Daten auf der Adresse sondern die Adresse selbst. Nun teilt sich die Kopie die entsteht einen Speicherbereich mit dem Original und wir bekommen unerwünschtes Verhalten.

1. Alle Änderungen des Namens von Person a wird ist automatisch auch in dem kopierten Objekt Person b.
2. Sobald b zerstört wird, ist der Pointer in a ein ungültiger Pointer
3. Einen ungültigen Pointer erneut löschen erzeugt undefinierbares Verhalten.
4. Memory Leaks über Zeit

4 Explizite Definition

Nachdem wir nun das Problem der implicit Kopie („member-wise cloning“) kennen und eben dieses Verhalten nicht wollen müssen wir einen Kopier-Konstruktor und einen „Copy-Assignment-Operator“ bzw Kopier-Zuweisungsoperator selbst anlegen um dieses Problem zu beheben.

```
1  // 1. copy constructor
2  Person(const person& that)
3  {
4          name = new char[strlen(that.name) + 1];
5          strcpy(name, that.name);
6          age = that.age;
7  }
8
9  // 2. copy assignment operator
10 Person& operator=(const person& that)
11 {
12         if (this != &that)
13         {
14                 delete[] name;
15                 // This is a dangerous point in the flow of
                      execution!
16                 // We have temporarily invalidated the class
                      invariants,
17                 // and the next statement might throw an exception
                      ,
18                 // leaving the object in an invalid state
19                 name = new char[strlen(that.name) + 1];
20                 strcpy(name, that.name);
21                 age = that.age;
22         }
23         return *this;
24 }
```

Achtung Hier wird nun auch der oben erwähnte Komplexitätsunterschied zwischen Kopier-Konstruktor und Kopier-Zuweisungsoperator sichtbar. Wir müssen im Kopier-Zuweisungsoperator prüfen ob es sich nicht um das selbe Objekt handelt, da wir sonst das Quell-Array zerstören würden und nicht wieder herstellen könnten. Außerdem müssen wir das Char-Array löschen um keine Memory Leaks zu produzieren, da dieses, anders als beim Kopier-Konstrukt bereits existiert. Was an dem oberen Beispiel immer noch ein Problem darstellen kann, ist wenn der Speicher zum erneuten Reservieren des Arrays nicht mehr ausreicht, hier kann es durch das fangen der geworfenen Exception passieren, dass das Objekt in einem ungültigen Zustand hinterlassen wird.

5 Kopierschutz

Wenn man nun nicht möchte, dass das Objekt kopiert werden kann so setzt man den Kopier-Zuweisungsoperator und den Kopierkonstruktor einfach private ohne Implmentierung:

```
1 private:
2         Person(const person& that);
3         Person& operator=(const person& that);
```

C++ 11 aufwärts unterstütz eine eigene Semantik um diesen Kopierschutz umzusetzen:

```
1 public:
2         Person(const person& that) = delete
3         Person& operator=(const person& that) = delete
```

6 Erweiterung „move-semantics“

Ab C++11 sind die so genannten „move-semantics“ dazu gekommen, diese ermöglichen es einem anderen Objekt einem anderen Daten zu „stehlen“. Zusammen mit den Drei vorher genannten „Special Member Functions“ bilden die zwei Weiteren, Move-Konstruktor und Move-Zuweisungsoperator, die **Regel der Fünf**.

```
// 1. Move Ctor
Person&(Person&& that)
{
        //if its unsure that that.name is initialized
        //also add a check for nullptr
        name = that.name;
        age = that.age;
        //'stolen' data
        that.name = nullptr;
        //'stolen' scalar value gets its 'default'
        that.age = 0;
}

// 2. Move-Assignment Operator for Person
Person& operator=(Person&& that)
{
        if (this != &that)
        {
                delete[] name;
                name = that.name;
                age = that.age;
                // we have 'stolen' the data
                that.name = nullptr;
                that.age = 0;
        }
        return *this;
}
```

Inhaltsverzeichnis