Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Von der Dreierregel zur Fünferregel in C++11

Oliver Niebsch

Deeply Embedded

Inhaltsverzeichnis

1 Einleitung 3

2 Die Dreierregel 4

2.1 Objektkopien 5

3 Die Fünferregel 7

3.1 Move Semantik 7

3.2 lvalue und rvalue Referenzen 9

3.2.1 std::move 9

3.2.2 std::forward für Funktionstemplates 12

4 Fazit 14

5 Quellenverzeichnis 15

1 Einleitung

Die Dreierregel ist ein elementarer Bestandteil der Programmiersprache C++. Durch die Einhaltung dieser einfachen Regel werden typische Fehler im Umgang mit Objekten und Speicherverwaltung vermieden. In C++11 kommt ein neues Feature hinzu, wodurch aus der Dreierregel die Fünferregel wird. Diese Arbeit erklärt die Bestandteile und die Notwendigkeit der Dreierregel, sowie Neuerung der Fünferregel mit Hilfe von kleinen Code-Beispiele rund um eine Klasse für Binärzahlen.

2 Die Dreierregel

Die Dreierregel in C++ besagt, dass der Destruktor, der Kopierkonstruktor und der Kopierzuweisungsoperator immer zusammen in einer Klasse auftreten. Gibt es die Notwendigkeit eine der drei Methoden zu definieren, müssen auch die anderen Beiden definiert werden (vgl. Brown 2014, S. 2). Wieso dies so ist, wird schon an einem kleinen Beispiel ersichtlich. Wenn man eine Klasse für Binärzahlen hat, die die einzelnen Stellen intern in einem int-Array verwaltet, könnte das so aussehen:

```
class Binary {
private:
	int* num;
	int size;
	string name;
	Binary(int* nums, int sz, string name);  // normaler Konstruktor

public:
	~Binary();                                // Destruktor
	string to_s();
	static Binary createBinary(int n);       // Factory-Methode
};
```

Der String *name* und die Methode *to_s* dienen der besseren Lesbarkeit der Ausgaben. Die Factory-Methode *createBinary* übernimmt das Erstellen einer neuen Binärzahl anhand einer Ganzzahl im Dezimalsystem. Die Implementierung der Klasse sieht entsprechend so aus:

```
Binary Binary::createBinary(int n) {
	int size = (int)log2(n * 1.0) + 1;
	int* num = new int[size];
	string name = "b" + to_string(n);

	for (int i = size - 1; i >= 0; i--) {
		num[i] = n % 2;
		n /= 2;
	}

	return Binary(num, size, name);
}
```

```
Binary::Binary(int* nums, int sz, string name)
	:num{nums}, size{sz}, name{name} {}

Binary::~Binary() {
	delete[] num;
}

string Binary::to_s() {
	string s = name + ": ";
	for (int i = 0; i < size; i++) {
		s += to_string(num[i]) + " ";
	}
	return s;
}
```

Da innerhalb der Factory-Methode das Array für die Binärziffern mit Hilfe von *new* erzeugt wird, muss der dafür allokierte Speicher auch wieder frei gegeben werden. Dies passiert innerhalb des Destruktors der Klasse *Binary*. Erzeugt man nun eine Binärzahl mit Hilfe der Factory-Methode und gibt sie auf der Konsole aus, erhält man z.B. folgende Ausgabe:

```
b6: 9828612 0 -33686019
```

Neben dieser falschen Ausgabe stürzt das Programm auch vor der Beendung ab. Das alles kommt zu Stande, weil die Dreierregel nicht eingehalten wurde. Da die Notwendigkeit besteht einen Destruktor zu definieren, müssen auch der Kopierkonstruktor und der Kopierzuweisungsoperator definiert werden.

2.1 Objektkopien

Ist in einer Klasse keine konkrete Implementierung des Kopierkonstruktors oder des Kopierzuweisungsoperators vorhanden, wird zum Kopieren von Objekten die Standardimplementierung von C++ benutzt. Diese erzeugt eine Objektkopie durch Kopieren der einzelnen Felder des Objektes (vgl. Stroustrup 2013, S.48). In der ersten Implementierung der Binärzahlen hat dieses Standardverhalten zu den Problemen geführt.

Die Factory-Methode gibt als Ergebnis ein Objekt der Klasse *Binary* zurück, welches innerhalb der Methode erzeugt wurde. Dieses Objekt verliert nach der Ausführung der Methode seine Gültigkeit und wird zerstört. Um trotzdem das neu erstellte Objekt der Factory-Methode zurück geben zu können, wird vor dem Zerstören des lokalen Objektes eine Kopie im Kontext des Aufrufers der Factory-Methode erzeugt.

Das bedeutet, wenn der Aufruf der Factory-Methode wie folgt aussieht, befindet sich in der Variable *b6* eine Kopie des innerhalb der Methode erzeugten Objektes.

```
Binary b6 = Binary::createBinary(6);
```

Beim Kopieren des Objektes werde alle Felder kopiert und somit auch der Pointer auf das int-Array. Da das lokale Objekt der Factory-Methode beim Verlassen gelöscht wird, wird der Destruktor aufgerufen, der das int-Array löscht. Das Löschen ist korrekt, da der Speicher des Arrays irgendwann wieder frei gegeben werden muss. Jedoch zeigt der kopierte Zeiger danach auf ein gelöschtes Array, was die falsche Ausgabe der Binärzahl und den Absturz des Programms zur Folge hat. Um dieses Problem zu lösen und die Dreierregel zu erfüllen, muss man der Binary-Klasse noch den Kopierkonstruktor und Kopierzuweisungsoperator hinzufügen. So wird beim Kopieren von Binärzahlen auch vom Array eine korrekte Kopie erstellt. Der Zusatz „_Copy“ wird zum Namen hinzugefügt, um die Ausgaben besser nachvollziehen zu können.

```
Binary::Binary(const Binary& other)
	: num{ new int[other.size] }, size{ other.size },
	  name{ other.name + "_Copy" } {
	for (int i = 0; i < size; i++) {
		num[i] = other.num[i];
	}
}

Binary& Binary::operator=(const Binary& other) {
	delete[] num;

	num = new int[other.size];
	size = other.size;
	name = other.name + "_Copy";

	for (int i = 0; i < size; i++) {
		num[i] = other.num[i];
	}

	return *this;
}
```

Die Ausgabe einer erzeugten Binärzahl sieht korrekt nun wie folgt aus. An dem Namen „b6_Copy“ erkennt man, dass zum Kopieren des Objektes der selbst definierte Kopierzuweisungsoperator verwendet wird.

```
b6_Copy: 1 1 0
```

3 Die Fünferregel

Die Fünferregel erweitert die Dreierregel um den Movekonstruktor und den Move-zuweisungsoperator. Das bedeutet, dass für keine Kopiermethoden, keine Movemethoden oder den Destruktor einer Klasse die Standardimplementierung von C++ genutzt werden sollte, wenn mindestens eine dieser Methoden innerhalb der Klasse spezifiziert wird. (vgl. Brown 2014, S.2).

3.1 Move Semantik

Das Kopieren von Objektstrukturen, wie z.B. Objekte der Klasse Binary, nimmt mehr Rechenzeit in Anspruch, je größer die Objektstruktur ist. Beim Beispiel der Binärzahl bleibend, wird beim Kopieren einer Binärzahl ein neues int-Array angelegt und die einzelnen Stellen der zu kopierenden Binärzahl in das neue Array übertragen. Somit wird Rechenzeit zum Allokieren des Speichers für das neue Array und zum Kopieren jedes einzelnen Eintrags benötigt. Um das Kopieren von großen Objektstrukturen zu vermeiden, kann man zur Übergabe an eine Methode Zeiger benutzen. So wird nur ein relativ kleiner Zeiger anstatt einer großen Objektstruktur kopiert. Beim Zurückgeben einer Objektstruktur als Ergebnis einer Methode, wie es beispielsweise bei der Factory-Methode für Binärzahlen der Fall ist, kann kein Zeiger genutzt werden, da dieser nach Abschluss der Methode auf ein gelöschtes Objekt zeigen würde (vgl. Stroustrup 2013, S. 50). Um dieses Problem ohne Kopieren von Objekten zu lösen, gibt es seit C++11 die Move Semantik.

Im Gegensatz zum Kopieren von Objekten, ermöglicht die Move Semantik den Besitzer eines Objektes zu ändern. Im Falle der Factory-Methode bedeutet dies, dass die Eigentümerschaft des in der Methode erstellten Objektes an den Aufrufer der Factory-Methode übertragen wird. Dabei muss gewährleistet werden, dass der alte Besitzer des Objektes dieses nicht mehr modifizieren kann. Eine ordentliche Implementierung der beiden Movemethoden in die Binary kann zum Beispiel so aussehen:

```
// Movekonstruktor
Binary::Binary(Binary&& other)
       : num{ other.num }, size{ other.size },
         name{ other.name + "_Move" } {
       other.num = nullptr;
       other.size = 0;
}

// Movezuweisung
Binary& Binary::operator=(Binary&& other) {
       delete[] num;

       num = other.num;
       size = other.size;
       name = other.name + "_Move";

       other.num = nullptr;
       other.size = 0;

       return *this;
}
```

Die Implementierung der beiden Movemethoden ist sehr ähnlich. Die Werte der Instanzvariablen werden von einem Quell-Objekt in ein anderes bzw. ein neu erstelltes Ziel-Objekt kopiert. Anschließend werden die Instanzvariablen des Quell-Objektes soweit überschrieben, dass dieses nicht mehr das Zielobjekt modifizieren kann. Beim Beispiel der Binärzahlen bedeutet dies, dass das int-Array des Quellobjektes mit einem *nullptr* überschrieben wird, da nur der Zeiger kopiert wird und somit ansonsten das in 2.1 beschriebene Problem wieder auftaucht. Zudem wird sicherheitshalber auch die interne Angabe zur Größe passend zum nullptr auf 0 geändert, sodass Zugriffe auf das Array an Hand der gespeicherten Größe nicht zu Fehlern führen. Der einzige Unterschied der beiden Methoden besteht darin, dass beim Movezuweisungsoperator ein eventuell vorhandenes Array im Ziel-Objekt zunächst gelöscht und so der dafür allokierte Speicher wieder ordentlich freigegeben wird. Beim Movekonstruktor ist das nicht nötig, da das Ziel-Objekt neu erstellt wird.

Auffällig in beiden Methoden ist der Typ des Parameter *Binary&&*. Dabei handelt es sich um eine sogenannte „rvalue Referenz“, welche eng mit der Move Semantik in Verbindung steht und im nachfolgenden Kapitel 3.2 näher erklärt wird. Der Parametertyp ist auch das Einzige, worin sich jeweils die Signaturen der Kopier- und Movemethoden unterscheiden. Dementsprechend handelt es sich um überladene

Methoden und der Compiler entscheidet an welcher Stelle welche Methode zum Einsatz kommt.

3.2 lvalue und rvalue Referenzen

Type& ist eine Referenz auf einen lvalue vom Typ *Type*, *Type&&* ist eine Referenz auf einen rvalue von Typ *Type*. Eine grundlegenden Unterscheidung der beiden Arten von Werte kann an Hand von vier Eigenschaften erreicht werden:

Eigenschaft	lvalue	rvalue
Kann auf der linken Seite einer Zuweisung stehen.	Ja	Nein
Kann auf der rechten Seite einer Zuweisung stehen.	Ja	Ja
Kann als Quell-Objekt für eine Movemethode dienen.	Nein	Ja
Kann als Quell-Objekt für eine Kopiermethode dienen.	Ja	Ja

Grundsätzlich handelt es sich bei lvalues um Objekte, die man in irgendeiner Art ansprechen kann, z.B. über einen Namen oder durch Folgen eines Pointers. Wohingegen rvalues in der Regel temporäre Objekte sind, wie z.B. Rückgabewerte von Funktionen, auf die niemand anderes zugreifen kann. Deswegen ist es erlaubt, rvalues als Quell-Objekt für eine Movemethode zu verwenden und ihnen somit die Werte zu „stehlen“ (vlg. Meyers 2015, S. 2 und Stroustrup 2013, S. 51).

3.2.1 std::move

Möchte man ein neues Objekt auf Basis eines anderen Objektes gleichen Typs erstellen oder ein vorhandenes Objekt überschreiben, entscheidet der Compiler an dieser Stelle, ob eine Kopier- oder einer Movemethode verwendet wird. Im folgenden Code wird zuerst der Movezuweisungsoperator benutzt, da das Quell-Objekt das Ergebnis der Factory-Methode ist und es sich somit um einen rvalue handelt. In der zweiten Zeile kommt der Kopierkonstruktor zum Einsatz, da das Quell-Objekt ein lvalue ist, denn man kann es über den Namen *bin* ansprechen.

```
Binary bin = Binary::createBinary(1263); // Movezuweisung
Binary copy(bin);                        // Kopierkonstruktor
```

Weiß man jedoch, dass nach der Erstellung von *copy* auf die Variable *bin* nie mehr zugegriffen wird, wäre es in der zweiten Zeile sinnvoller den Movekonstruktor zu verwenden. Um den Compiler dazu zu bringen, gibt es mit Hilfe der Methode *std::*move die Möglichkeit ein beliebiges Objekt in einen rvalue umzuwandeln. Diese

Methode nimmt eine beliebige Referenz und castet diese in eine rvalue Referenz. Das Codebeispiel sieht mit *std::move* wie folgt aus:

```
Binary bin = Binary::createBinary(1263); // Movezuweisung
Binary mv(std::move(bin));               // Movekonstruktor
```

Zudem kann *std::move* hilfreich sein, wenn man eine rvalue Referenz in geschachtelten Methoden weitergeben möchte.

```
void foo(Binary&& bz) {
	cout << bz.to_s();
}

void bar(Binary&& bin) {
	foo(bin);	// Fehler
}
```

```
void foo(Binary&& bz) {
	cout << bz.to_s();
}

void bar(Binary&& bin) {
	foo(std::move(bin));
}
```

Bei dem Code links führt die markierte Zeile *foo(bin)* zu einem Fehler, da die Methode nicht mit einem rvalue aufgerufen wird. Zwar ist *bin* eine Referenz auf einen rvalue, aber der Wert der Variable selbst ist ein lvalue, denn schließlich kann man den Wert über einen Namen ansprechen (vgl. Allain). Möchte man eine rvalue Referenz aus einem Parameter als rvalue Referenz an eine andere Methode weitergeben, hilft einem *std::move* wie im Beispiel rechts dargestellt.

Die Methode *std::move* kann auch bei der Implementierung der Movemethoden zum Einsatz kommen. Hat man beispielsweise eine Klasse *IntArray*, die ein int-Array verwaltet und benutzt diese anstatt direkt einen Pointer in der Binary Klasse zu speichern, könnte das so aussehen:

```
class IntArray {
private:
	int* num;
	int size;

public:
	IntArray(int* nums, int sz);		// Konstruktor
	~IntArray();				// Destruktor

	IntArray(const IntArray&);		// Kopierkonstruktor
	IntArray& operator=(const IntArray&);	// Kopierzuweisung

	IntArray(IntArray&&);			// Movekonstruktor
	IntArray& operator=(IntArray&&);	// Movezuweisung
};
```

```
class Binary {
private:
	IntArray nums;
	string name;
	Binary(IntArray nums, string name);	// Konstruktor

public:
	~Binary();				// Destruktor

	Binary(const Binary&);			// Kopierkonstruktor
	Binary& operator=(const Binary&);	// Kopierzuweisung

	Binary(Binary&&);			// Movekonstruktor
	Binary& operator=(Binary&&);		// Movezuweisung
};
```

Der Movekonstruktor der Klasse Binary lässt sich nun sehr kurz schreiben:

```
Binary::Binary(Binary&& other)
	: nums{ other.nums }, name{ other.name + "_Move" } {}
```

Jedoch wird so zum Initialisieren der Instanzvariable *nums* der Kopierkonstruktor der Klasse *IntArray* verwendet, da es sich bei *other.nums* um einen lvalue handelt. Es muss ein lvalue sein, denn dieses Objekt wird über den Namen *other.num* angesprochen. Der Kopierkonstruktor ist an dieser Stelle aber gar nicht nötig, da es sich bei *other* um eine rvalue Referenz handelt und somit klar ist, dass auf das Objekt und damit auch auf dessen Instanzvariablen nicht mehr zugegriffen wird. Die Lösung für dieses Problem liefert die Methode *std::move*:

```
Binary::Binary(Binary&& other)
	: nums{ std::move(other.nums) }, name{ other.name + "_Move" } {}
```

Man könnte nun meinen mit Hilfe von *std::move* ist es immer möglich die Verwendung der Movemethoden zu erzwingen. Leider ist dem nicht so. Der nachfolgende Code zeigt an einem Beispiel, wann dies nicht geht:

```
void foo(const Binary bin) {
	Binary copy(std::move(bin));
}
```

Auch in diesem Beispiel macht *std::move* genau das, was es machen soll: Es gibt eine rvalue Referenz zurück. Jedoch wird dabei die Angabe *const* nicht entfernt und das Ergebnis ist eine *const* rvalue Referenz. Die Angabe *const* sagt aus, dass das Objekt hinter der Referenz nicht verändert wird. Wenn aber der Movekonstruktor

zum Einsatz käme, würden die Werte aus dem Objekt in der Variable *bin* geändert werden und die Angabe *const* wäre somit nicht eingehalten. Dementsprechend hat der Compiler an dieser Stelle keine andere Wahl, als den Kopierkonstruktor zu benutzen.

3.2.2 std::forward für Funktionstemplates

Neben *std::move* gibt es noch eine Methode, die einen Cast in ein rvalue ermöglicht: die Methode *std::forward*. Der Unterschied dieser beiden Methoden besteht darin, dass *std::move* immer eine rvalue Referenz zurück gibt und *std::forward* nur dann eine rvalue Referenz zurück gibt, wenn das übergebene Objekt eine rvalue Referenz beinhaltet. Ansonsten liefert *std::forward* eine lvalue Referenz zurück (vgl. Meyers 2015, S. 161). Typischerweise braucht man diesen bedingten Cast in Verbindung mit Funktionstemplates. Beispielsweise bei einer überladenen Methode, die ausgibt, ob es sich bei dem Parameter um eine lvalue oder eine rvalue Referenz handelt:

```
template<typename T>
void test(T& t) {
	printf("Parameter ist eine lvalue Referenz\n");
}

template<typename T>
void test(T&& t) {
	cout << "Parameter ist eine rvalue Referenz\n";
}
```

```
Binary bin = Binary::createBinary(1);
test(bin);    // Fehler
```

Der Aufruf der Methode mit einem lvalue führt jedoch zu einem Fehler. Der Compiler behauptet, dass beide Methodensignaturen von *test* zum Aufruf *test(bin)* passen würde, obwohl das Objekt in *bin* eindeutig ein lvalue ist. Neben der ersten Implementierung von *test*, kann auch die zweite Implementierung mit einem lvalue als Parameter aufgerufen werden. Der Grund ist, dass der Parameter vom Typ *T&&* nicht nur für eine rvalue Referenz steht, sondern ebenfalls mit einer lvalue Referenz befüllt werden kann, da *T* ein Typ-Parameter ist. Scott Meyers spricht hierbei von „universellen Referenzen“ (vgl. Meyers 2015, S. 164). Die Lösung des Problems ist das Hinzufügen und Benutzen einer Hilfsmethode, die ebenfalls eine universelle Referenz als Parameter besitzt und den Parameter an die Methode *test* weiter leitet:

```
template<typename T>
void check(T&& val) {
	test(val);
}
```

Ruft man die Methode *check* nun jeweils mit einem lvalue und einem rvalue auf, erscheint zweimal die Ausgabe „Parameter ist eine lvalue Referenz“ auf der Konsole. Das Problem hierbei ist wieder, dass der Wert von *val* immer ein lvalue ist, egal welche Art von Referenz in *val* steht. Das Benutzen von *std::move* führt ebenfalls zu einer falschen Aussage, da *test* dann immer mit einer rvalue Referenz aufgerufen werden würde. Abhilfe schafft in einer solchen Situation die Methode *std::forward*, da diese nur einen Cast in eine rvalue Referenz durchführt, wenn *val* eine rvalue Referenz beinhaltet. So wird die Implementierung von *test* mit einer universellen Referenz als Parameter nur aufgerufen, wenn auch *check* mit einer rvalue Referenz als Parameter aufgerufen wurde.

```
template<typename T>
void check(T&& val) {
	test(std::forward(b));
}
```

4 Fazit

Mit der Einwicklung der Dreierregel zur Fünferregel hat sich eine der elementarsten Grundsätze beim Erstellen eigener Klassen in C++ an die Neuerungen der Programmiersprache angepasst. Die Vorgabe einen Movekonstruktor und einen Movezuweisungsoperator zu definieren bildet dabei eine gute Grundlage im Umgang mit der neuen Move Semantik. Durch die Ähnlichkeit der beiden Methoden zu ihren Kopier-Pendants in den meisten Anwendungsfällen, ist es einfach auch bereits vorhandenen Code an die neuen Standards anzupassen und so durch kleine Änderungen die Effizienz zu steigern.

5 Quellenverzeichnis

Brown 2014
BROWN, Walter E. : *Proposing the Rule of Five, v2.* ISO/IEC JTC1/SC22/WG21 document N3839, 2014. – online verfügbar unter: http://open-std.org/JTC1/SC22/WG21/docs/papers/2014/n3839.pdf Abruf: 2016-06-16

Stroustrup 2013
STROUSTRUP, Bjarne : *A Tour of C++*. Addison-Wesley, 2013. – 978-0-321-958310 ISBN

Meyers 2015
MEYERS, Scott : *Effective Modern C++*. O'Reilly Media Inc., 2015. – 978-1-491-90399-5 ISBN – online verfügbar unter: http://file.allitebooks.com/20150510/Effective%20Modern%20C++.pdf

Allain
ALLAIN, ALEX : *Move semantivs and rvalue references in C++11.* http://www.cprogramming.com/c++11/rvalue-references-and-move-semantics-in-c++11.html Abruf: 2016-07-22