Inhaltsverzeichnis

Abstract

Folgendes Dokument erläutert die Move-Semantik in der C++-Programmiersprache, welche seit dem C++11-Programmiersprachenstandard Teil der C++-Programmiersprache ist.

Dabei kommen Text, Codebeispiele, sowie Diagramme und Tabellen zur Erläuterung zum Einsatz.

Begonnen wird mit einer Einleitung in die Thematik der Move-Semantik.

An die Einleitung schließt sich der Hauptteil an, welcher mit dem Kapitel „Was ist ein Move?“ beginnt.

Darauf folgen die Kapitel

„Gefährliche und harmlose Moves“,

„Value categories“,

„Rvalue-Referenzen“,

„Implizite Konvertierungen“,

„Move-Konstruktoren“,

„Move-Zuweisungsoperatoren“,

„Von Lvalues moven“,

„Xvalues“,

„Aus Funktionen hinausmoven“,

„In Datenmember hineinmoven“,

„Spezielle Mitgliedsfunktionen“,

„Forwarding references“,

„Implementation von std::move“,

„Perfect forwarding“,

„Move-Sicherheit“

und

„Interaktion mit Standardbibliothekscontainern“

Anschließend folgt der Teil „Anwendung“, der die Anwendung der Move-Semantik mit Beispielen zeigt, darauf folgt das Fazit.

Das Dokument beendet sich mit Quellenangaben.

Einleitung

Die Move-Semantik erlaubt es einem Objekt, unter bestimmten Bedingungen Besitz über externe Ressourcen eines anderen Objekts zu übernehmen. Dies ist auf zweierlei Weise von Signifikanz:

1. Günstigere Move-Operationen statt teurere Kopieroperationen zu verwenden.
   Beispiel:
   Es sei folgende Klasse gegeben:

```
#include <cstring>

class MyClass {
public:
    explicit MyClass(const char* string)
        : string_(new char[std::strlen(string) + 1]) {
        std::strcpy(string_, string);
    }

    MyClass(const MyClass& other)
        : string_(new char[std::strlen(other.string_) + 1]) {
        std::strcpy(string_, other.string_);
    }

    MyClass& operator=(const MyClass& other) {
        char* tmp = new char[std::strlen(other.string_) + 1];
        std::strcpy(tmp, other.string_);
        delete[] string_;
        string_ = tmp;
        return *this;
    }

    ~MyClass() {
        delete[] string_;
    }

private:
    char* string_;
};
```

Folgende Funktion sei deklariert und definiert:

```
void f(MyClass v);
```

Die main-Funktion sehe wie folgt aus:

```
int main() {
    f(MyClass(“Hallo”));
    return 0;
}
```

Es wird ein temporäres Objekt vom Typ MyClass mit dem Stringliteral “Hallo“ konstruiert, entsprechend des Konstruktors.

Dieses Objekt wird nun in den Parameter v der Funktion f mit dem Kopierkonstruktor kopiert.

Da das temporäre Objekt nicht länger gebraucht wird, wird die Zeichenfolge “Hallo“ einmal unnötigerweise kopiert.

In C++11 kann man mit einem Move-Konstruktor und einem Move-Zuweisungsoperator durch Move-Semantik diese unnötige Kopie entfernen.
Die Klasse wäre dann wie folgt definiert:

```
#include <cstring>

class MyClass final {
public:
    explicit MyClass(const char* string)
        : string_{ new char[std::strlen(string) + 1] } {
        std::strcpy(string_, string);
    }

    MyClass(const MyClass& other)
        : string_{ new char[std::strlen(other.string_) + 1] } {
        std::strcpy(string_, other.string_);
    }

    MyClass(MyClass&& other) noexcept
        : string_{ other.string_ } {
        other.string_ = nullptr;
    }

    MyClass& operator=(const MyClass& other) {
        auto tmp = new char[std::strlen(other.string_) + 1];
        std::strcpy(tmp, other.string_);
        delete[] string_;
        string_ = tmp;
        return *this;
    }

    MyClass& operator=(MyClass&& other) noexcept {
        auto tmp = other.string_;
        other.string_ = nullptr;
        delete[] string_;
        string_ = tmp;
        return *this;
    }

    ~MyClass() {
        delete[] string_;
    }

private:
```

```
    char* string_;
};
```

Nun wird statt des Kopierkonstruktors der Move-Konstruktor aufgerufen, was dazu führt, dass die Zeichenkette nicht kopiert wird, sondern das neue Objekt v in f einfach Besitz über die Zeichenkette des temporären Objekts übernimmt.

2. Das Implementieren vom Move-Only-Typen.
   Es gibt Typen, die logisch nicht kopierbar sind.
   Dazu zählen zum Beispiel Netzwerksockets, Dateihandles, Locks, etc.
   Auch der neue smart pointer std::unique_ptr zählt zu den Move-Only-Typen, da dieser das Konzept eines einzelnen Besitzers über eine Ressource implementiert.

Hauptteil

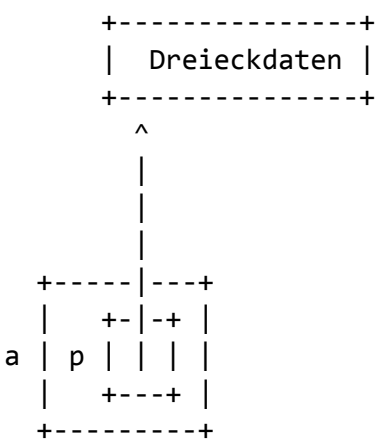
Was ist ein Move?

Die C++98 Standardbibliothek bietet einen smart pointer, welcher die Semantik eines einzelnen Besitzers über eine Ressource implementiert, welcher std::auto_ptr heißt.
Der Sinn und Zweck von std::auto_ptr war es zu garantieren, dass ein dynamisch allokiertes Objekt immer freigegeben wird, unabhängig davon ob Ausnahmen geworfen werden.
Beispiel:

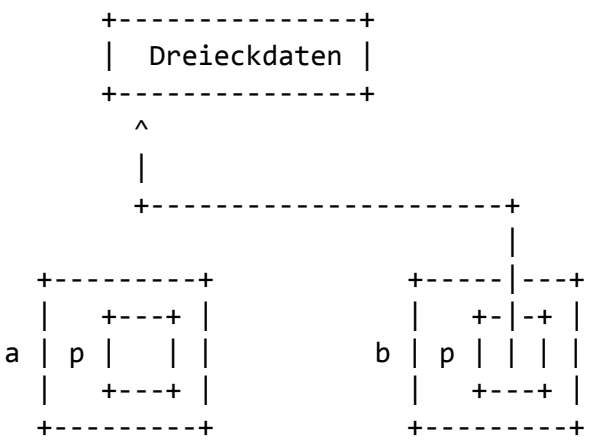
```
{
    std::auto_ptr<Form> a(new Dreieck);
    // …
    // beliebiger Code, könnte Ausnahmen werfen
    // …
}   // ← wenn a seinen Gültigkeitsbereich verlässt wird das Dreieck
    // automatisch zerstört.
```

Was an dem auto_ptr besonders ist, ist wie er kopiert:

```
std::auto_ptr<Form> a(new Dreieck);
```



```
std::auto_ptr<Form> b(a);
```



Achten Sie darauf, wie die Initialisierung von b mit a nicht das Dreieck kopiert.
Stattdessen wird der Besitz über das Dreieck von a nach b transferiert.

Ein Objekt zu Moven bedeutet den Besitz der Ressourcen, welche es verwaltet an ein anderes Objekt zu übergeben.

Der Kopierkonstruktor von auto_ptr könnte in etwa so aussehen:

```
auto_ptr(auto_ptr& quelle) // kein const, da quelle modifiziert wird
{
    p = quelle.p;
    quelle.p = 0; // die quelle besitzt das Objekt nicht mehr.
}
```

Gefährliche und harmlose Moves

Das Gefährliche an dem auto_ptr ist, dass das was syntaktisch wie ein Kopieren aussieht in

Wirklichkeit ein Move ist. Ein Versuch eine Funktion über den überladenen member of pointer-Operator von std::auto_ptr aufzurufen, nachdem von diesem der Besitz über das zu verwaltende Objekt durch den Kopierkonstruktor oder Kopierzuweisungsoperator hinfort transferiert wurde resultiert in undefiniertem Verhalten. Somit ist der Umgang mit auto_ptr gefährlich, da man ihn leicht falsch benutzen kann.

Beispiel:

```
auto_ptr<Form> a(new Dreieck); // Dreieck erstellen

auto_ptr<Form> b(a);           // a nach b moven

double flaeche = a->flaeche();  // undefiniertes Verhalten
```

Bei einer Fabrikfunktion tritt diese Gefahr nicht auf.

Beispiel:

```
auto_ptr<Form> erstelle_dreieck()
```

```
{
    return auto_ptr<Form>(new Dreieck);
}
auto_ptr<Form> c(erstelle_dreieck()); // move das Temporäre in c
double flaeche = erstelle_dreieck()->flaeche(); // sicher
```

Achten Sie darauf, wie beide Beispiele demselben syntaktischen Muster folgen:

```
auto_ptr<Form> variable(ausdruck);
double flaeche = ausdruck->flaeche();
```

Trotzdem führt ein Beispiel zu undefiniertem Verhalten, während das andere dieses nicht tut. a und erstelle_dreieck() sind ausdrücke vom selben Typ haben aber eine andere value category.

Value categories

Es gibt einen signifikanten Unterschied zwischen den Ausdrücken a, welcher eine auto_ptr Variable bezeichnet und dem Ausdruck erstelle_dreieck(), welcher einen Funktionsaufruf, welcher einen auto_ptr per Wert zurückgibt, bezeichnet. erstelle_dreieck() erstellt bei jedem neuen Aufruf immer ein neues temporäres auto_ptr Objekt. a ist eine lvalue, während erstelle_dreieck() ein Beispiel für eine rvalue ist.

Von lvalues wie a zu moven ist gefährlich, denn man könnte später versuchen beispielsweise eine Mitgliedsfunktion über a aufzurufen, welches in undefiniertem Verhalten resultieren würde.

Von rvalues wie erstelle_dreieck() zu moven ist sicher, denn man kann dieses temporäre Objekt nicht noch einmal referenzieren. Wenn man erneut erstelle_dreieck() schreibt, so bekommt man ein anderen temporäres Objekt. Das temporäre Objekt, von welchem moved wird verlässt seinen Gültigkeitsbereich nach dem Ausdruck.

```
auto_ptr<Form> c(erstelle_dreieck());
                                    ^ das Temporäre wird zerstört
```

Die Buchstaben l und r bei lvalues und rvalues haben historische Herkunft. Das l steht dabei für die linke Seite einer Zuweisung und das r für die rechte Seite einer Zuweisung. In C++ gilt dies allerdings nicht mehr, da es lvalues gibt, die nicht auf der linken Seite einer Zuweisung auftreten können, wie zum Beispiel arrays, oder benutzerdefinierte Type ohne Zuweisungsoperator. Außerdem gibt es auch rvalues, die auf der linken Seite einer Zuweisung auftreten können, dazu zählen alle rvalues von Klassentypen mit Zuweisungsoperator.

Eine rvalue eines Klassentyps ist ein Ausdruck, dessen Evaluation ein temporäres Objekt erzeugt. Unter normalen Umständen kann kein anderer Ausdruck im selben Gültigkeitsbereich des selbe temporäre Objekt bezeichnen.

Rvalue-Referenzen

Nun ist uns klar, dass das Moven von lvalues potenziell gefährlich sein kann und dass das Moven von rvalues harmlos ist. Wenn man lvalue-Argumente von rvalue-Argumenten unterscheiden kann, dann kann man das Moven von lvalues unterbinden, oder es wenigstens explizit machen, sodass man nicht mehr aus Versehen Moven kann.

Die Antwort auf das Problem sind in C++11 die Rvalue-Referenzen.

Eine Rvalue-Referenz ist ein neuer Referenztypmodifizierer, welcher eine Referenz deklariert, die ausschließlich an rvalues gebunden werden kann.

Der Syntax ist X&&. Die aus C++98 und C++03 bekannte Referenz im Stil X&, wird Lvalue-Referenz genannt.

X&& ist keine Referenz auf eine Referenz, so etwas ist in C++ nicht möglich, da Referenzen keine Objekte sind.

Mit dem const-Typmodifizierer haben wir bereits vier verschiedene Referenztypen.

Der folgenden Tabelle ist zu entnehmen an welche Art von Ausdruck welcher Referenztyp gebunden werden kann:

	lvalue	const lvalue	rvalue	const rvalue
X&	Ja	Nein	Nein	Nein
const X&	Ja	Ja	Ja	Ja
X&&	Nein	Nein	Ja	Nein
const X&&	Nein	Nein	Ja	Ja

Implizite Konvertierungen

Eine Rvalue-Referenz X&& kann auch an alle value categories eines anderen Typs Y gebunden werden, gegeben, dass es eine implizite Konvertierung von Y zu X gibt. In dem Fall wird ein temporäres Objekt vom Typ X erstellt, an welches die Rvalue-Referenz gebunden wird.

```
void eine_funktion(std::string&& r);

eine_funktion(“Hallo Welt“);
```

In dem obigen Beispiel ist “Hallo Welt“ eine lvalue vom Ty const char[11]. Da es eine implizite Konvertierung von const char[11] durch const char* zu std::string gibt, wird ein temporäres Objekt vom Typ std::string kreiert und r wird an dieses gebunden.

Move-Konstruktoren

Ein nützliches Beispiel für eine Funktion mit einem X&&-Parameter ist der Move-Konstruktor X::X(X&& quelle). Der Sinn des Move-Konstruktors ist es den Besitz von einer verwalteten Ressource von einem Quellobjekt zum momentanen Objekt zu transferieren.

In C++11 wurde die Klasse std::auto_ptr mit std::unique_ptr effektiv erstetzt.

std::unique_ptr nutzt Rvalue-Referenzen.

Im Folgenden soll eine vereinfachte Version von unique_ptr erstellt werden. Zuerst wird ein roher Zeiger gekapselt und die Operatoren -> und * werden überladen.

```
template<typename T>
class unique_ptr
{
    T* ptr;

public:
    T* operator->() const
    {
        return ptr;
    }

    T& operator*() const
    {
        return *ptr;
    }
// Der Konstruktor übernimmt den Besitz über das Objekt und
// der Destruktor löscht es.
    explicit unique_ptr(T* p = nullptr)
    {
        ptr = p;
    }

    ~unique_ptr()
    {
        delete ptr;
    }
// Nun der Move-Konstruktor
    unique_ptr(unique_ptr&& source)
```

```
    {
        ptr = source.ptr;
        source.ptr = nullptr;
    }
// Dieser Move-Konstruktor tut das Gleiche, wie
// der Kopierkonstruktor von auto_ptr,
// doch kann er nur mit Rvalues beliefert werden.
unique_ptr<Form> a(new Dreieck);
unique_ptr<Form> b(a);                  // Fehler
unique_ptr<Form> c(erstelle_dreieck()); // okay
```

Die zweite Zeile wird einen Kompilierfehler verursachen, da a eine lvalue ist, aber der parameter unique_ptr&& source nur an rvalues gebunden werden kann. So können gefährliche Moven von lvalues nicht mehr implizit geschehen.

Die dritte Zeile kompiliert fehlerfrei, da erstelle_dreieck() eine rvalue ist. Der Move-Konstruktor wird den Besitz von dem temporären Objekt zu c transferieren.

Move-Zuweisungsoperatoren

Der Move-Zuweisungsoperator gibt die alte Ressource frei und bekommt eine neue Ressource vom Argument.

```
unique_ptr& operator=(unique_ptr&& source)
{
    auto temp = other.ptr;
    other.ptr = nullptr;
    delete ptr;
    ptr = temp;

    return *this;
}
```

Wie auch ein Kopierzuweisungsoperator sollte der Move-Zuweisungsoperator selbstzuweisungssicher sein. Das obige Beispiel zeigt eine selbstzuweisungssichere Implementation.

Von Lvalues moven

Manchmal möchte man von Lvalues moven. Also möchte man, dass der Compiler ein Lvalue-Objekt behandelt, als wäre es ein Rvalue-Objekt, sodass der Move-Konstruktor aufgerufen werden kann, auch wenn dies potenziell unsicher sein kann. Dafür bietet die C++11-Standardbibliothek ein Funktionstemplate std::move im header <utility>. std::move casted eine Lvalue zu einer Rvalue, es moved aber nichts selbst. Es ermöglicht lediglich das Moven von einer Lvalue. Ist zum Beispiel kein Move-Konstruktor vorhanden, wird nicht gemoved sondern der Kopierkonstruktor aufgerufen.

So kann man von einer Lvalue moven:

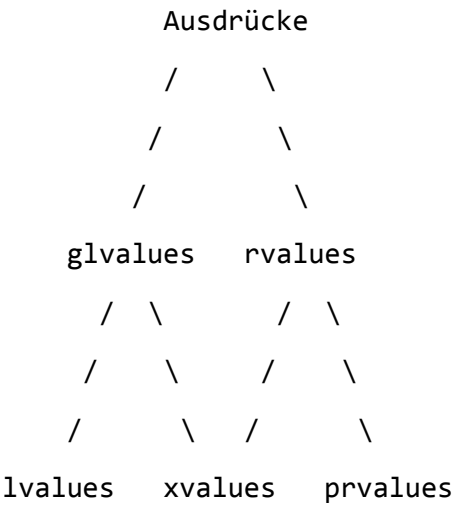
```
unique_ptr<Form> a(new Dreieck);
unique_ptr<Form> b(a);            // Fehler
unique_ptr<Form> c(std::move(a)); // okay
```

Nach der dritten Zeile besitzt a nicht mehr das Dreieck.

Xvalues

Trotz dass std::move eine rvalue ist, kreiert dessen Evaluation kein temporäres Objekt. Dieses Rätsel zwang das Standardkomitee eine dritte value category einzuführen. Etwas, welches an eine Rvalue-Referenz gebunden werden kann, aber nicht eine Rvalue im traditionellem Sinne ist, wird xvalue genannt (expiring value). Die traditionellen Rvalues wurden zu prvalues umbenannt (pure rvalues).

prvalues und xvalues sind rvalues. Xvalues und Lvalues sind beide glvalues (generalized lvalues). Der Zusammenhang sei durch folgendes Diagramm dargestellt:



C++98 rvalues heißen prvalues in C++11. Somit kann der Leser nun nach Einführung der prvalues jedes Auftreten von rvalue in den voherigen Kapiteln mit prvalue ersetzen.

Aus Funktionen hinausmoven

Man kann auch aus Funktionen hinausmoven.

Wenn eine Funktion etwas per Wert zurückgibt, so kann man mit dem Rückgabewert eine Variable an der Aufrufsstelle initialisieren. Dabei wird der Ausdruck nach dem return Schlüsselwort als Argument an den Move-Konstruktor genutzt.

```
unique_ptr<Form> erstelle_dreieck()
{
    return unique_ptr<Form>(new Dreieck);
}          \----------------------------/
                  |
                  | Temporäres Objekt wird in c hineingemoved
                  |
                  v
unique_ptr<Form> c(erstelle_dreieck());
```

Automatische Objekte (lokale Variablen, welche nicht als static deklariert sind) können implizit aus Funktionen hinausgemoved werden:

```
unique_ptr<Form> erstelle_quadrat()
{
    unique_ptr<Form> result(new Quadrat);
    return result;   // kein std::move
}
```

Am Ende der Funktion endet der Gültigkeitsbereich von result. Nach dem zurückkehren der Funktion existiert result nicht mehr, darum kann das Lvalue-Objekt ohne std::move implizit aus der Funktion hinausgemoved werden.

In Datenmember hineinmoven

Folgender Code kompiliert nicht:

```
class Foo
{
    unique_ptr<Form> member;

public:
    Foo(unique_ptr<Form>&& parameter)
```

```
    : member(parameter)   // error
    {}
};
```

Das liegt daran, dass unique_ptr ein move-only type ist und parameter eine Lvalue ist.

Eine Referenz ist immer eine Lvalue. Das gilt auch für Rvalue-Referenzen. Eine Rvalue-Referenz ist lediglich eine Referenz, die nur an Rvalues gebunden werden kann.

Die Lösung ist es explizit mit std::move zu einer Rvalue zu casten.

```
class Foo
{
    unique_ptr<Form> member;

public:
    Foo(unique_ptr<Form>&& parameter)
    : member(std::move(parameter))
    {}
};
```

Spezielle Mitgliedsfunktionen

In C++98 werden implizit drei spezielle Mitgliedsfunktionen generiert, wenn sie benötigt werden. Diese sind Kopierkonstruktor, Kopierzuweisungsoperator und Destruktor.

```
X::X(const X&);            // Kopierkonstruktor
X& X::operator=(const X&); // Kopierzuweisungsoperator
X::~X();                   // Destruktor
```

In C++11 werden zwei weitere spezielle Mitgliedsfunktionen generiert.

Diese sind Move-Konstruktor und Move-Zuweisungsoperator.

```
X::X(X&&);            // Move-Konstruktor
X& X::operator=(X&&); // Move-Zuweisungsoperator
```

Diese zwei neuen speziellen Mitgliedsfunktionen werden nur dann implizit generiert, wenn keine speziellen Mitgliedsfunktionen manuell deklariert wurden. Des Weiteren werden Kopierkonstruktor und Kopierzuweisungsoperator nicht mehr generiert, wenn man seinen eigenen Move-Konstruktor oder Move-Zuweisungsoperator deklariert.

Folgende Tabelle stellt dar, welche speziellen Mitgliedsfunktionen der Compiler in welchen Fällen generiert:

Wird der Compiler generieren …	Wenn der Programmierer deklariert…					
	Nichts	dtor	cctor	cpy-op=	mctor	mv-op=
dtor	Ja		Ja	Ja	Ja	Ja
cctor	Ja	Ja		Ja	Nein	Nein
cpy-op=	Ja	Ja	Ja		Nein	Nein
mctor	Ja	Nein	Nein	Nein		Nein
mv-op=	Ja	Nein	Nein	Nein	Nein	

Wird ein Kopierkonstruktor oder ein Kopierzuweisungsoperator deklariert, so werden Move-Konstruktor und Move-Zuweisungsoperator nicht generiert, dennoch kann man Objekte des Typs von Rvalues erzeugen, diese werden dann den Kopierkonstruktor nutzen. Genauso wird bei Zuweisung mit einer Rvalue dann der Kopierzuweisungsoperator verwendet.

Wenn man eine Klasse schreibt, welche keine unverwalteten Ressourcen hat, so braucht man keine der fünf speziellen Mitgliedsfunktionen selbst schreiben. Der Compiler wird die korrekten Kopier- und Move-Semantiken generieren. Ansonsten muss man die speziellen Mitgliedsfunktionen selbst schreiben. Wenn eine Klasse durch Move-Semantik, keine Vorteile erfährt, so braucht man den Move-Konstruktor und Move-Zuweisungsoperator auch nicht zu definieren.

Forwarding references

Folgendes Funktionstemplate sei gegeben:

```
template <typename T>
void foo(T&&);
```

T&& ist in diesem Fall nicht eine Rvalue-Referenz, sondern eine forwarding reference.

Es kann auch an lvalues gebunden werden.

```
foo(erstelle_dreieck()); // T ist unique_ptr<Form>&&,
                         // T&& ist unique_ptr<Form>&&
unique_ptr<Form> a(new Dreieck);
foo(a); // T ist unique_ptr<Form>&,
        // T&& ist unique_ptr<Form>&
```

Wird also eine Lvalue als Argument übergeben, so wird die forwarding reference zu einer entsprechenden Lvalue-Referenz. Wird eine Rvalue übergeben, so wird die forwarding reference zu der entsprecheneden Rvalue Referenz.

Der Typ T wird abgeleitet und so entstehen invalide Typen aus Referenzen zu Referenzen.

wie z.B. T & &, T& &&, T&& &, oder T&& &&. Diese werden anhand der reference collapsing rules zu validen Referenztypen umgewandelt:

T& & wird zu T&

T& && wird zu T&

T&& & wird zu T&

T&& && wird zu T&&

Möchte man ein Funktionstemplate, welches nur rvalues akzeptiert, kann man das zum Beispiel wie folgt deklarieren:

```
#include <type_traits>

template <typename T>

typename std::enable_if<std::is_rvalue_reference<T&&>::value, void>::type

foo(T&&);
```

Implementation von std::move

```
template<typename T>

typename std::remove_reference<T>::type&&

move(T&& t)

{

    return static_cast<typename std::remove_reference<T>::type&&>(t);

}
```

move akzupetiert jede Art von Parameter dank der forwarding reference T&&.

move gibt eine Rvalue-Referenz zurück.

Der Einsatz der Metafunktion std::remove_reference ist nötig, denn für Lvalues vom Typ X wäre der Rückgabetyp X& &&, welche zu X& zusammenfallen würde. Da t, wie jede Referenz, eine Lvalue ist, aber wir eine Rvalue-Referenz an t binden wollen, muss t explizit zum korrekten Rückgabetyp gecasted werden.

Der Aufruf einer Funktion, welche eine Rvalue-Referenz zurückgimt, wie zum Beispiel std::move, ist eine xvalue.

Perfect forwarding

Ein Problem welches forwarding references lösen ist das perfect forwarding-Problem.

Gegeben sei folgende simple Fabrikfunktion:

```
template <typename T, typename Arg>
std::shared_ptr<T> factory(Arg arg)
{
    return std::shared_ptr<T>(new T(arg));
}
```

Die Absicht ist es das Argument in arg an den Konstruktor von T weiterzuleiten.

Idealerweise, sollte es sich so verhalten, als wäre die Fabrikfunktion gar nicht da und es wäre so, als würde der Konstruktor direkt aufgerufen.

Der Code im obigen Beispiel versagt dabei, da er eine unnötige Kopie im Parameter vornimmt. Des Weiteren wird das Argument nicht nur unnötig kopiert, sondern wird es immer als Lvalue weitergeleitet, auch wenn die value category eine andere war.

Nimmt man nun eine Referenz statt eine Kopie zu machen, so ist das nicht viel besser.

```
template <typename T, typename Arg>
std::shared_ptr<T> factory(Arg& arg)
{
    return std::shared_ptr<T>(new T(arg));
}
```

Nun kann man die Funktion nicht mehr mit rvalues aufrufen.

```
factory<X>(foo()); // Fehler, wenn foo by value zurückgibt
factory<X>(41);    // Fehler
```

Man kann nun ein const Lvalue Referenz-Parameter stattdessen nehmen.

```
template <typename T, typename Arg>
std::shared_ptr<T> factory(const Arg& arg)
{
    return std::shared_ptr<T>(new T(arg));
}
```

Problematisch ist, dass die Fabrikfunktion nicht von Move-Semantik profitieren kann, da sie ihr Argument immer als Lvalue weitterreicht. Von dem Argument zu Moven ist nicht legal, da es const-qualifiziert ist und möglicherweise eine Lvalue ist.

Man möchte also das Argument als Lvalue weiterleiten, wenn es eine Lvalue ist und es als Rvalue weiterleites, wenn es eine Rvalue ist.

Um dies zu erreichen werden im folgenden Beispiel forwarding references genutzt. Diese fallen aufgrund der reference collapsing rules zum korrekten Referenzparametertyp zusammen.

```
template <typename T, typename Arg>
std::shared_ptr<T> factory(Arg&& arg)
{
    return std::shared_ptr<T>(new T(std::forward<Arg>(arg)));
}
```

std::forward kann wie folgt definiert sein:

```
template <typename S>
S&& forward(typename std::remove_reference<S>::type& a) noexcept
{
    return static_cast<S&&>(a);
}
```

Es sei angenommen, dass factory<A> mit einer Lvalue vom Typ X aufgerufen wird.

```
X x;
factory<A>(x);
```

Aufgrund der reference collapsing rules wird das Template-Typ-Argument Arg in factory zu X&.

Der Compiler wird folgende Instanziierungen von factory und std::forward erstellen:

```
std::shared_ptr<A> factory(X& && arg)
{
    return std::shared_ptr<A>(new A(std::forward<X&>(arg)));
}
X& && forward(std::remove_reference<X&>::type& a) noexcept
{
    return static_cast<X& &&>(a);
}
```

Nach dem Auswerten von std::remove_reference und nach Anwenden der reference collapsing rules wird dies zu:

```
std::shared_ptr<A> factory(X& arg)
{
    return std::shared_ptr<A>(new A(std::forward<X&>(arg)));
}
```

```
X& std::forward(X& a)
{
    return static_cast<X&>(a);
}
```

Wie man sieht funktioniert das perfect forwarding für Lvalues. Die Lvalueness bleibt durch alle Funktionsaufrufe hindurch dank std::forward erhalten.

Als nächstes soll davon ausgegangen sein, dass factory<A> mit einer Rvalue vom Typ X aufgerufen wird:

```
X foo();
factory<A>(foo());
```

Dies wird zu folgenden Funktionstemplateinstanziierungen führen:

```
std::shared_ptr<A> factory(X&& && arg)
{
    return std::shared_ptr<A>(new A(std::forward<X&&>(arg)));
}
X&& && std::forward(typename std::remove_reference<X&&>::type& a) noexcept
{
    return static_cast<X&& &&>(a);
}
```

Nach dem Auswerten von std::remove_reference und nach Anwenden der reference collapsing rules wird dies zu:

```
std::shared_ptr<A> factory(X&& arg)
{
    return std::shared_ptr<A>(new A(std::forward<X&&>(arg)));
}
X&& std::forward(X& a) noexcept
{
    return static_cast<X&&>(a);
}
```

Somit wird die Rvalue letztendlich durch std::forward zu einer Rvalue zurückgecasted um die Rvalueness beim Weiterleiten beizubehalten.

So löst std::forward das perfect forwarding-Problem, Lvalues werden immer als Lvalues weitergeleitet und Rvalues immer als Rvalues, so werden die value categories beibehalten.

Move-Sicherheit

Wenn man von einer Lvalue moved, zum Beispiel mit Hilfe von std::move, so verbleibt ein move-from-Objekt. Diese Objekte werden auch Zombie-Objekte genannt und der inkorrekte Umgang mit diesen kann undefiniertes Verhalten auslösen. Der Implementierer eines Typs kann, sofern dieser Move-Semantik nutzt, den Move-Konstruktor und Move-Zuweisungsoperator so definieren, dass bestimmte Garantien für Objekte des Typs im moved-from-Zustand geboten werden können.

Dabei kann man folgende Arten von Move-Sicherheit unterscheiden (es sind auch weitere denkbar):

1. Kein-Move-Garantie (nur Kopieren erlaubt)

Ein Typ bietet Kein-Move-Garantie, wenn der Move-Konstruktor oder Move-Zuweisungsoperator keine Move-Operationen vornehmen. Ein Move dieser Typen ist äquivalent mit einem Kopieren. Wenn ein Typ diese Garantie bietet, hat er keine nutzerdefinierten Move-Operationen und hat nur Datenmitglieder, die die gleiche Garantie bieten. Solche Typen besitzen typischerweise keine Ressourcen, welche freigegeben werden müssen, darum benötigen sie keine speziellen Kopier-Operationen oder Destruktoren.
Jeder triviale Typ hat diese Garantie, sowie jeder Typ für welchen keine Move-Operationen implizit generiert werden.

2. Starke Move-Sicherheit (Wohldefinierter und valider Moved-From-Zustand)

Der Moved-From-Zustand eines Typs, welcher diese Move-Sicherheit bietet ist wohldefiniert. Alle Mitgliedsfunktionen, dessen Vorbedingungen in dem definierten Zustand erfüllt sind, können sicher aufgerufen werden. Zusätzlich werden diese Mitgliedsfunktionen deterministische Auswirkungen oder Resultate haben.
Ein Beispiel für einen Typ, der die starke Move-Sicherheit bietet ist std::unique_ptr.
Move-Konstruktion ist definiert als Transfer des Besitzes über das verwaltete Objekt.
Ein std::unique_ptr im Moved-From-Zustand besitzt immer nichts. operator bool() wird false zurückgeben und get() wird nullptr zurückgeben. Aufrufe von operator*() oder operator->() werden in undefiniertem Verhalten resultieren. Allerdings ist zu beachten, dass bei std::unique_ptrs mit einem custom deleter, der custom deleter natürlich auch moved wird, dieser kann natürlich eine andere Move-Sicherheit bieten.

3. Einfache Move-Sicherheit (Valider aber unspezifizierter Moved-From-Zustand)

Die Einfache Move-Sicherheit benötigt keinen wohldefinierten Moved-From-Zustand.
Sie benötigt lediglich, dass der Moved-From-Zustand valide ist, aber der genaue Zustand ist nicht spezifiziert. Man kann sicher alle Mitgliedsfunktionen aufrufen, die einen weiten Vertrag haben, das heißt, sie haben keine Vorbedingungen. Aber es wird nicht garantiert welche Resultate diese Funktionsaufrufe haben werden, sie sind nicht deterministisch wie bei der starken Move-Sicherheit.
Ein Beispiel für einen Typ mit einfacher Move-Sicherheit ist std::string.

```
std::string a(“Hello World“);
std::string b(std::move(a));
std::cout << a.c_str() << ‘\n‘;
```

Der Aufruf der Mitgliedsfunktion c_str auf dem Moved-From-Objekt a wird kein undefiniertes Verhalten auslösen, aber ist das Resultat nicht spezifiziert. Es wird vom Standard nicht garantiert welche Ausgabe erzeugt werden würde. std::string::c_str() hat keine Vorbedingung, somit wird

garantiert, dass der Aufruf kein undefiniertes Verhalten auslöst. Die Ausgabe ist von der Implementation abhängig.

4. Keine Move-Sicherheit

Die geringste Garantie bietet keine Move-Sicherheit. Das Moved-From-Objekt ist nach dem Move nicht mehr valide. Man darf lediglich den Destruktor aufrufen, oder dem Objekt einen neuen Wert zuweisen (sofern es einen Zuweisungsoperator gibt). Objekte müssen also immer zerstörbar sein, auch im Moved-From-Zustand.

Interaktion mit Standardbibliothekscontainern

Eigene Move-Konstruktoren und Move-Zuweisungsoperatoren sollten, wenn möglich als noexcept deklariert werden. noexcept garantiert, das eine Funktion nicht wirft. Sollte die Funktion doch werfen, so wird std::terminate() aufgerufen, was zum Abbruch des Programms führt. Ist es möglich den Move-Konstruktor und den Move-Zuweisungsoperator so zu definieren, dass sie nie werfen, so sollte dies getan werden. Dann sollte zusätzlich auch der Move-Konstruktor und der Move-Zuweisungsoperator als noexcept deklariert werden.
Die Standardbibliothekscontainer bieten die „Strong Exception Guarantee“, diese besagt, wenn eine Exception auftritt, so sind die Objekte immer noch im gleichen Zustand, als wäre die Mitgliedsfunktion des Standardbibliothekscontainers nicht aufgerufen worden.
Ein Move modifiziert allerdings das Quellobjekt. Das Quellobjekt wird im Moved-From-Zustand hinterlassen. Sollte dieser Move also werfen, kann die „Strong Exception Gurantee“ nicht mehr geboten werden. Darum überprüfen die Mitgliedsfunktionen von Standardbibliothekscontainern, ob ein Move von dem Quellobjekt werfen könnte (nicht als noexcept deklarierter Move-Konstruktor oder Move-Zuweisungsoperator). Ist der Move-Konstruktor und Move-Zuweisungsoperator also nicht noexcept, so werden Standardbibliothekscontainer nicht von der nutzerimplementierten Move-Semantik Gebrauch machen, sondern stattdessen kopieren.

<u>Anwendung</u>
Die Move-Semantik findet an vielen Stellen Anwendung.
Dazu ein paar Beispiele:

```
std::vector<int> f() {
    std::vector<int> v{ 1, 2, 3, 4 };
    return v;
}
```

Die Funktion f gibt einen std::vector<int> per value zurück.
Dabei wird in C++11 keine unnötige Kopie im Rückgabetyp erzeugt.
Stattdessen wird die lokale Variable v implizit aus der Funktion hinausgemoved.
Dies geschieht auch für andere Typen, die gemoved werden können.

In modernem C++-Code sieht man in Konstruktoren häufig folgendes Muster:

```
class MyClass {
public:
    explicit MyClass(SomeType v)
        : v_(std::move(v)) { }
private:
    SomeType v_;
};
```

Im Konstruktor wird ein Objekt vom Typ SomeType per Wert angenommen.
Anschließend wird von dem Parameter v in das Datenmitglied v_ gemoved.
Ist das Argument, welches an den Konstruktor übergeben wird eine Lvalue, so wird es in den Parameter v kopiert und dann in v_ gemoved. Dies ist nicht ineffizient, da Lvalues ohnehin kopiert werden müssten, die Lvalue zu modifizieren würde den Aufrufer sicher überraschen.
Ist das Argument, welches an den Konstruktor übergeben wird eine Rvalue, so wird es in den Parameter v gemoved und anschließend in v_ gemoved.
Der Vorteil dieses Musters ist es, dass man nicht zwischen L- und Rvalues unterscheiden muss und dennoch effizenter Code generiert wird.
Ansonsten müsste man für jede mögliche Kombination an L-Rvalueness der Argumente einen eigenen Konstruktor schreiben, was zu einer kombinatorischen Explosion führen würde.
Alternativ könnte man den Konstruktor auch als Template definieren, welches mit forwarding references arbeitet und somit unabhängig von L-Rvalueness der Argumente mit std::forward immer das richtige tut. Allerdings müsste man dann den Konstruktor in einer Headerdatei definieren, was zu erhöhter Kompilierzeit führt, sofern die Headerdatei von mehr als einer Quelldatei inkludiert wird. Daneben müsste man sich auch möglicherweise darüber Gedanken machen die Template-Typ-Parameter mit Hilfe von Techniken wie SFINAE (Substitution failure is not an error, siehe auch std::enable_if) zu beschränken, da C++ noch keine Concepts bietet (siehe Concepts Technical Specification). Ein Ansatz mit std::enable_if ist auch gerade bei Konstruktoren nicht so schön, da man enable_if nicht als Rückgabetyp nutzen kann, somit müsste man es als Standardtyp für einen Template-Typ-Parameter definieren, welches der Nutzer dann allerdings mit Unsinn explizit instanziieren kann und so das enable_if vollkommen aushebeln kann. Somit ist die Vorgehensweise aus obigen Codebeispiel vorzuziehen, da diese wesentlich simpler ist.
Es sei allerdings anzumerken, dass man für Mitgliedsfunktionen, wie zum Beispiel Settern es generell vorziehen sollte explizit mit L-Value und Rvalue-Referenzen zu überladen.
Für Lvalues würde das Argument in den Parameter kopiert werden, dies kann potenziell teurer sein, als es mit dem Kopierzuweisungsoperator direkt in das Datenmitglied zu kopieren. Ist der String im Datenmitglied zum Beispiel groß genug um den im Argument zu halten, so braucht man keine Allokation von dynamischen Speicher für das Kopieren. Der Parameter würde aber, wenn per Wert angenommen wird immer einen komplett neuen String allokieren, dies ist ineffizient. Bei Konstruktoren ist das nicht ineffizient, da das Objekt mit dem Konstruktor erstellt wird und es somit keine Puffer in den Datenmitgliedern geben kann, die wiederverwertet werden könnten (außer eines der Datenmitglieder ist eine Referenz auf ein Objekt außerhalb der zu erstellenden Instanz). Bei Settern ist eine kombinatorische Explosion, im Gegensatz zu Konstruktoren, typischerweise nicht gegeben, da nur ein einzelnes, oder wenige Argumente angenommen werden.

Fazit

Die C++11-Move-Semantik ist eines der Erkennungsmerkmale von modernem C++. Die Move-Semantik ist für eine Sprache, die, wie C++, bis zu sehr hohe Laufzeitperformance bieten will absolut essentiell. Das Anfertigen von unnötigen Kopien, wie es vor C++11 geschah, ist für eine Programmiersprache, die Ansprüche wie C++ hat, nicht akzeptabel. Die Notwendigkeit für Unterstützung von Move-Semantik durch die Kernsprache war schon lange erkennbar. Indizien dafür sind zum Beispiel Optimierungen, wie Copy Ellision, die alle bekannteren Compiler vornehmen. Dabei werden unnötige Kopien beim Zurückgeben von lokalen Variablen aus Funktionen wegoptimiert. Analog dazu werden die lokalen Variablen in C++11 aus der Funktion implizit herausgemoved, sofern sie movebar sind. Ist ein Typ nur kopierbar, so muss der Compiler allerdings nach wie vor keine Copy Ellision vornehmen, im C++17-Standard wird dies allerdings Pflicht werden. Ein weiteres Indiz für die Notwendigkeit von Kernsprachenunterstützung für Move-Semantik ist, man begann mit std::swap Move-Semantik zu emulieren.

```
void f(std::vector<std::string> v_&)
{
    std::vector<std::string> v;
    using std::swap;
    swap(v_, v);
    // do something with v
}
```

Im obigen Codebeispiel wird ein leerer vector v erstellt, um diesen mit dem an welchen die Referenz v_ gebunden ist zu swappen. So werden effektiv, wie bei C++11 Move-Semantik die Ressourcen von dem vector an den v_ gebunden ist gestohlen. Natürlich erfordert dies, dass die Ressourcen, die das Objekt besitzt an welches die Referenz gebunden ist, nicht mehr gebraucht werden. Aber wenn der Programmierer eine Funktion wie f aufruft, die eine non-const Referenz im Parameter hat, geht er ohnehin davon aus, dass die Funktion das Argument modifiziert.
Dank der C++11-Move-Semantik sind nun solche Workarounds, um unnötige Kopien zu vermeiden, nicht mehr nötig.
Es sei noch angemerkt, dass die Move-Semantik an sich kein neues Konzept ist, da der Typ std::auto_ptr bereits in C++98 Move-Semantik implementiert, lediglich die Unterstützung der Move-Semantik durch die Kernsprache ist in C++11 neu. Wie aus diesem Dokument hervorgeht ist der Umgang mit std::auto_ptr durchaus gefährlich, was ein weiteres Indiz für die Notwendigkeit der C++11-Move-Semantik ist. Der Typ std::auto_ptr ist seit C++11 deprecated und wird mit dem C++17-Sprachstandard aus der Standardbibliothek entfernt, stattdessen wird empfohlen std::unique_ptr zu nehmen. Somit ist die Einführung der C++11-Move-Semantik insgesamt als Gewinn zu betrachten, wobei sie auch nicht verzichtbar gewesen wäre.

Quellenangaben

http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/n4606.pdf

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm

https://meetingcpp.com/index.php/br/items/cpp-and-zombies-a-moving-question.html

https://foonathan.github.io/blog/2016/07/23/move-safety.html

https://akrzemi1.wordpress.com/2011/08/11/move-constructor/

https://akrzemi1.wordpress.com/2011/06/10/using-noexcept/

http://en.cppreference.com/w/cpp/utility/move

http://en.cppreference.com/w/cpp/utility/forward

http://en.cppreference.com/w/cpp/language/rule_of_three

http://en.cppreference.com/w/cpp/language/value_category

http://en.cppreference.com/w/cpp/language/reference

http://en.cppreference.com/w/cpp/language/move_constructor

http://en.cppreference.com/w/cpp/language/move_assignment

http://stackoverflow.com/questions/3106110/what-are-move-semantics/11540204#11540204

http://thbecker.net/articles/rvalue_references/section_07.html

http://thbecker.net/articles/rvalue_references/section_08.html

http://www.codingstandard.com/rule/12-5-4-declare-noexcept-the-move-constructor-and-move-assignment-operator/