Die Move-Semantik ist ein seit dem C++11-Standard standardisiertes fundamentales Konzept der C++-Programmiersprache.

Die Move-Semantik bietet primär eine Laufzeitperformanceoptimierung, da sie erlaubt unnötiges kopieren von Objekten zu ersparen.

Für Objekte, welche eine oder mehrere Ressourcen durch z.B. rohe Zeiger managen, sodass der compilergenerierte Kopierkonstruktor und Kopierzuweisungsoperator eine flache Kopie (shallow copy) vornehmen, wird in C++03 häufig ein Kopierkonstruktor, Kopierzuweisungsoperator und Destruktor vom Programmierer definiert, wenn tiefe Kopien gewünscht sind. Dies ist als die „Dreierregel“ (rule of three) bekannt. Diese Art von Objekten kann von der Move-Semantik profitieren, indem zusätzlich ein Move-Konstruktor und Move-Zuweisungsoperator definiert wird, dies ist seit C++11 als die „Fünferregel“ (rule of five) bekannnt.

Der Move-Konstruktor und Move-Zuweisungsoperator stehlen dabei die Ressource vom Quellobjekt, bei einem rohen Zeiger bedeutet das typischerweise, dass der Zeiger in Zielobjekt den Wert des Zeigers des Quellobjektes annimmt und der Zeiger im Quellobjekt genullt wird. Anmerkung: Der Move-Zuweisungsoperator sollte dabei, wie auch der Kopierzuweisungsoperator so designend sein, dass er selbstzuweisungssicher ist.

Ein Move kann also, anders als eine Kopie, das Quellobjekt verändern. Das bedeutet auch, dass es illegal ist von einem const-Objekt zu moven. In welchem Zustand das Quellobjekt nach einem Move ist, ist von der Implementierung des Move-Konstruktors und Move-Zuweisungsoperators abhängig. Typen aus der Standardbibliothek sind nach einem Move mindestens in einem validen, aber undefinierten Zustand. Somit mag der „Wert“ undefiniert sein, jedoch halten die Invarianten, eigene Typen sollten diesem Modell aus Konsistenzgründen folgen und mindestens dieselben Garantien bieten. Nach einem Move ist es nicht unüblich, dass das aufrufen von Memberfunktionen undefiniertes Verhalten auslöst. Lediglich der Destruktor und die Zuweisungsoperatoren (sofern definiert) müssen valide Operationen auf solchen Objekten sein. Solche Objekte werden „Zombie-Objekte“ genannt. Ob nach einem Move noch weitere Memberfunktionen valide sind, ist der Dokumentation des jeweiligen Typs zu entnehmen. Generell sollten Zombie-Objekte eliminiert werden, indem ihnen ein definierter Wert zugewiesen wird (sofern dies günstig möglich ist), oder indem sie so bald wie möglich zerstört werden. Zombie-Objekte können entstehen, wenn ein Programmierer z.B. ein lokales Objekt zu einer rvalue casted, z.B. mit der Standardbibliotheksfunktion std::move. Anzumerken ist, dass std::move selbst keine Move-Operation vornimmt, sondern lediglich eine rvalue-Referenz gebunden an das Argument zurückgibt, ist z.B. kein Move-Konstruktor definiert, jedoch ein Kopierkonstruktor, so wird dieser „trotz“ std::move aufgerufen, da eine konstante lvalue-Referenz an eine rvalue gebunden werden kann.

Die Einführung der Move-Semantik hat keine Auswirkungen auf POD (plain old data) -Typen, das heißt, dass Move und Kopieren von z.B. einem int identisch sind (der Compiler wird den gleichen Code erzeugen).

Der signifikanteste Performancegewinn erfahren die meisten Codebases dank der Move-Semantik beim Moven von temporären Objekten, welche in C++03 noch unnötigerweise kopiert werden mussten. Somit kann ein einfacher Neukompilieren einer C++03 Codebase mit einem C++11-Compiler bereits einen signifikanten Performancegewinn ohne Codeanpassungen mit sich ziehen, durch die Präsenz von Move-Konstruktoren und Move-Zuweisungsoperatoren in der Standardbibliothek, sowie durch Compilergenerierte.

Um die Move-Semantik in C++ zu unterstützen, wurden mit C++11 rvalue-Referenzen eingeführt, welche auch bei Move-Konstruktoren und Move-Zuweisungsoperatoren eingesetzt werden. In der Overloadresolution werden Overloads mit rvalue-Referenzparametern für rvalue-Argumente über Overloads mit konstanten lvalue-Referenzparametern bevorzugt. Somit kann man sicher sein, dass eine rvalue-Referenz auch an eine rvalue gebunden ist, außer der Programmierer „casted“ eine lvalue zu einer rvalue mit z.B. std::move. Darüber hinaus können rvalue-Referenzen das Objekt an welches sie gebunden sind modifizieren.

Des Weiteren wurden die sogenannten forwarding references eingeführt, welche zu einem anderen Referenztyp bei template type deduction anhand der reference collapsing rules zerfallen. Diese werden mit der gleichen Syntax, wie rvalue-Referenzen deklariert. Bei Deklarationen von Variablen von Template-Typ-Parameter-Typen und bei Deklaration mit auto ist eine solche Deklaration wie T&& oder auto&& eine Deklaration einer forwarding reference. Der Typ wird anhand der template type deduction abgeleitet, was darin resultieren kann, dass ein invalider Referenztyp zur Kompilierzeit entsteht, indem der Typ T z.B. zu int& abgeleitet wird, was zu einem Parametertyp von int& && führen würde. Offensichtlich ist solch eine Typdeklaration nicht legal, da man nicht eine Referenz an eine Referenz binden kann, da Referenzen keine Objekte in C++ sind. Darum werden die reference collapsing rules angewandt, welche besagen das & & zu & kollabiert, && & zu &, & && zu & und && && zu &&. Wie man sieht ist der resultierende Typ lediglich eine rvalue-Referenz, wenn sowohl der abgeleitete Typ einen rvalue-Referenz-Typmodifizierer hat, als auch die Deklaration der Variable. Somit findet reference collapsing strenggenommen auch bei T& und auto& statt, welche aber immer in lvalue-Referenzen resultieren, und somit nicht weiter interessant sind. Andere Typmodifizierer, wie const oder volatile bleiben erhalten (im Gegensatz zu nur auto, welche const und ref-Modifizierer fallen lässt (was decltype(auto) wiederrum nicht tut)). auto verhält sich in diesem Zusammenhang gleich wie Template-Typ-Parameter, da auto Typableitung auch durch template type deduction vornimmt. Mit Hilfe von forwarding references und der assoziierten Standardbibliotheksfunktion std::forward<T> wird das perfect forwarding-Problem gelöst. Das heißt man kann so ein Objekt immer dann wenn es eine rvalue ist Moven und ansonsten Kopieren, somit bekommt man wann immer möglich die Performancegewinne durch Move-Semantik, moved aber nicht versehentlich von lvalues, ohne dass man in generischen Code l- und rvalues speziell behandeln muss.

Für den korrekten Einsatz der Move-Semantik ist es wichtig die assoziierten Standardbibliotheksfunktionen std::move und std::forward<T> zu verstehen und zu wissen wann sie anzuwenden sind.

std::move (unäre Funktion aus dem Header <utility>) casted ihr Argument bedingungslos zu einer rvalue. Dies geschieht, indem das Typsystem durch Casting umgangen wird und eine rvalue-Referenz an das Argument gebunden wird, unabhängig davon, ob das Argument eine rvalue ist oder nicht. std::move implementiert selbst nicht die Move-Semantik und hätte besser rvalue_cast genannt werden sollen. std::move kann genutzt werden um lokale Variablen, die nicht mehr gebraucht werden in eine konsumierende Funktion zu Moven, dass wirklich ein Move vorgenommen wird ist nicht garantiert, sondern hängt von der Implementation ab, so bieten z.B. Standardbibliothekscontainer Overloads für l- und rvalues von einfügenden Funktionen, wie z.B. push_back an, bei welchen die rvalue-Referenz-Overloads aus dem overload resolution set per SFINAE entfernt werden, sofern der Move-Konstruktor/Zuweisungsoperator des Elementtyps nicht noexcept ist, also wenn der Typ nicht nothrow move constructible/assignable ist. Dies geschieht, um die strong exception safety guarantee aus C++03 weiterhin bieten zu können. Darum sollten eigene Typen noexcept Move-Konstruktoren/Zuweisungsoperatoren bekommen, wenn möglich, da

ansonsten Standardbibliothekscontainer von ihnen keinen Gebrauch machen werden. std::move wird am häufigsten mit rvalue-Referenzen verwendet, z.B. wenn ein Objekt weitergeleitet werden soll, beispielsweise soll in einem Move-Konstruktor einer Klasse in einer Vererbungshierarchie der Basisteil des Objektes an den Konstruktor in der Basisklasse weitergeleitet werden. Durch die rvalue-Referenz wird die rvalue sozusagen als lvalue betrachtet, welches mit std::move beim Weiterleiten an eine andere Funktion „rückgängig“ gemacht wird, ohne std::move würde der Kopierkonstruktor in der Basisklasse aufgerugen werden. Es ist wichtig zwischen rvalue-Referenzen und forwarding references unterscheiden zu können um nicht versehentlich std::move auf eine forwarding reference anzuwenden, da diese an eine lvalue gebunden sein kann.

std::forward<T> ist ein bedingter Cast zu einer rvalue. Wenn der Typ im Templatetypparameter einen rvalue-Referenz Typmodifizierer hat, resultiert std::forward<T> effektiv in std::move, ansonsten verhält es sich wie ein pass by value. std::forward<T> wird mit forwarding references für des perfect forwarding benutzt.

Ein guter Test, ob man std::move und std::forward<T>, l/rvalueness und die verschiedenen Referenztypen verstanden hat ist Frage #116 von CppQuiz.org (http://cppquiz.org/quiz/question/116). Nach der Beantwortung wird eine gute Erklärung gegeben. Anzumerken ist, dass lediglich das Funktionstemplate „g“ gefährlich ist, da es potenziell ungewollt von einer lvalue moved.