Move-Semantik und Performance

☑ **Lösung zu Aufgabe 4.1** Nur die ersten zwei Funktionsargumente `s` und `t.s` führen zum Aufruf von `f(S& s)` mit der Lvalue-Referenz, während alle anderen zum Aufruf von `f(S&& s)` mit der Rvalue-Referenz führen. Entprechend sind die Wertekategorien der zwei ersten Ausdrücke jeweils *Lvalue* und die der restlichen Ausdrücke alle *Rvalue*. Die Kategorie Rvalue kann dann noch feiner unterteilt werden in *Prvalue* (`S{}`) und *Xvalue* (`T{}.s`, `std::move(s)`, `std::move(t).s` und `std::move(t.s)`).

☑ **Lösung zu Aufgabe 4.2** Auf den Vektoren `v1` und `v4` wird als letzte Anweisung jeweils eine Verschiebung ausgeführt. Die allgemeine Regel dazu sagt nur, dass verschobene Objekte in einem gültigen Zustand sein müssen; die Regel gibt jedoch keine weitere Garantie über den Inhalt dieser Objekte. Zusätzlich zu dieser allgemeinen Regel gibt der konkrete Typ `std::vector<int>` laut Spezifikation aber die Garantie, dass verschobene Vektoren immer leer sind. Deshalb kann man hier tatsächlich mit Sicherheit sagen, dass `v1` und `v4` am Schluss beide die leere Zahlensequenz enthalten.

Der Vektor `v2` erhält von `v4` mittels Verschiebeoperator auf der zweitletzten Zeile die Zahlensequenz `7,8,9`, die er dann bis am Schluss behält.

Der Vektor `v3` erhält von `v1` mittels Verschiebekonstruktor auf der dritten Zeile die Zahlensequenz `1,2,3,4,5`, die er ebenfalls bis am Schluss behält. Der zusätzliche Aufruf `std::move(v3)` hat keinen nennenswerten Effekt, weil es sich dabei nur um einen Cast von einer Lvalue zu einer Rvalue handelt.

☑ **Lösung zu Aufgabe 4.3** Wir vereinbaren, dass bei einem verschobenen `UniqueIntPtr` die Member-Variable `m_ptr` stets auf den Nullzeiger `nullptr` gesetzt ist. Diese Vereinbarung ist natürlich und hat den Vorteil, dass sich der bereits implementierte Destruktor auch für verschobene Objekte korrekt verhält.

Weiter ist anzumerken, dass die eigene Implementation von Verschiebekonstruktor und Verschiebeoperator dazu führt, dass der Kopierkonstruktor und der Zuweisungsoperator nicht mehr automatisch vom Compiler erstellt werden. Trotzdem ist es eine gute Idee, diese zwei speziellen Member-Funktionen explizit (mittels Schlüsselwort `delete`) zu entfernen.

```
class UniqueIntPtr {
private:
  int* m_ptr;
public:
  UniqueIntPtr(int* ptr) : m_ptr(ptr) {}
  ~UniqueIntPtr() {
    if(m_ptr != nullptr) {
      delete m_ptr;
    }
  }
  // Verschiebeoperationen implementieren:
  UniqueIntPtr(UniqueIntPtr&& other) : m_ptr(other.m_ptr) {
    other.m_ptr = nullptr;
  }
  UniqueIntPtr& operator=(UniqueIntPtr&& other) {
    if(this != &other) {
      if(m_ptr != nullptr) {
        delete m_ptr;
      }
      m_ptr = other.m_ptr;
      other.m_ptr = nullptr;
    }
    return *this;
  }
  // Kopieroperationen entfernen:
  UniqueIntPtr(const UniqueIntPtr& other) = delete;
  UniqueIntPtr& operator=(const UniqueIntPtr& other) = delete;
};
```