# Move-Semantik und Performance

**Aufgabe 4.1** Erklären Sie, welcher der zwei Overloads der Funktion `f` jeweils aufgerufen wird und warum dies geschieht. Was sind die Wertekategorien der jeweiligen Funktionsargumente?

```
struct S {};
struct T { S s; };

void f(S&  s) { std::cout << "call f(S&  s)" << std::endl; }
void f(S&& s) { std::cout << "call f(S&& s)" << std::endl; }

int main() {
  S s; T t;

  f(s);
  f(t.s);
  f(S{});
  f(T{}.s);
  f(std::move(s));
  f(std::move(t).s);
  f(std::move(t.s));
}
```

**Aufgabe 4.2** Was ist die jeweilige enthaltene Zahlensequenz bei den vier Vektoren `v1`, `v2`, `v3` und `v4`, nachdem die folgenden Codezeilen fertig ausgeführt worden sind? Kann man diese Frage überhaupt bei jedem Vektor mit Sicherheit beantworten?

```
std::vector<int> v1 = {1,2,3,4,5};
std::vector<int> v2 = v1;
std::vector<int> v3 = std::move(v1);
std::vector<int> v4 = {7,8,9};
v2 = std::move(v4);
std::move(v3);
```

**Aufgabe 4.3** Erweitern Sie die folgende Klasse `UniqueIntPtr` indem Sie den Verschiebekonstruktor und den Verschiebeoperator implementieren. Die Klasse soll garantieren, dass auf dem verwalteten Heap-allozierten `int` genau einmal `delete` aufgerufen wird. Da sich diese Klasse nicht sinnvoll kopieren lässt, soll zusätzlich der Kopierkonstruktor und der Zuweisungsoperator von `UniqueIntPtr` entfernt werden.

Hierbei handelt es sich um unseren eigenen Versuch, die bekannte Klasse `std::unique_ptr<int>` aus der Standardbibliothek nachzuprogrammieren. Eine generische Version `UniquePtr<T>` für beliebige Typen `T` würde sogenannte *Templates* erfordern, die wir erst später im Semester kennen lernen werden.

```
class UniqueIntPtr {
private:
  int* m_ptr;
public:
  UniqueIntPtr(int* ptr) : m_ptr(ptr) {}
  ~UniqueIntPtr() {
    if(m_ptr != nullptr) {
      delete m_ptr;
    }
  }
};
```