Fachhochschule Nordwestschweiz
Hochschule für Technik

Programmieren in C++

Christian Lang (Lac)

27. September 2019

Move-Semantik

Inhalt

- Return-Value-Optimization
- Move-Semantik
- lvalue und rvalue Referenzen
- Perfect-Forwarding

Return-Value-Optimization (RVO)

- Aus Performance-Gründen kann der Compiler Kopien weg-optimieren, z.B. bei `return`
- Spezialfall der Copy Elision
- Compiler darf das Verhalten des Programms verändern

```
1  int CreateInt() {
2    int value;
3    if (...) {
4      value = 0;
5    } else {
6      value = 1;
7    }
8    return value;      // value wird nicht kopiert
9  }
```

Beispiel: RVO

```cpp
struct C {
  C() {}
  C(const C&) {
    std::cout << "Copy" << std::endl;
  }
};

C f() {
  return C();
}

int main() {
  std::cout << "Start" << std::endl;
  C obj = f();
  return EXIT_SUCCESS;
}
```

```
// möglicher
// Output
Start
Copy
Copy

// oder
Start
Copy

// oder
Start
```

Move-Semantik

- Ähnliches Konzept wie Copy Elision aber kontrollierbar
- verwendet rvalue- Referenzen
- Implementieren von Move-ctor und Move-assignment-Operator
  - genau wie bei Copy

```
1  Vector CreateVector() {
2    Vector v;
3    v.add(Punkt(1,2,3));
4
5    // v ist ausserhalb
6    // eine rvalue-Ref
7    return v;
8  }
```

```
1  int main() {
2    // Move-ctor
3    Vector v1(CreateVector());
4    // Move-ctor
5    Vector v2 = CreateVector();
6
7    Vector v3;
8    // Move-operator=
9    v3 = CreateVector();
10 }
```

lvalue und rvalue Referenzen

lvalue

- hat immer einen Namen
- `<type>&`
- referenziert “normale” Objekte

rvalue

- hat keinen Namen
- `<type>&&`
- referenziert temporäre Objekte
- zerfällt zu lvalue

```
1  struct Left {
2    Left(const Left& l) {...}
3    Left& operator=(const Left& l) {...}
4  };
5
6  struct Right {
7    Right(Right&& r) {...}
8    Right& operator=(Right&& r) {...}
9  };
```

std::move

Probleme:

- manchmal möchte man auch ein nicht-temporäres Objekt verschieben
- rvalue zerfällt (decay) automatisch zu lvalue
  - z.B. innerhalb von Funktionen

```
1  template<typename T>
2  constexpr T&& move(T&& t) {
3    return static_cast<T&&>(t);
4
5    // return t; würde zu einem normalen T oder T& zerfallen
6  }
```

Beispiel: `std::move`

```
1  std::string s1 = "hello";
2
3  std::string s2 = std::move(s1);
4
5  // s1 == ""
6  // s2 == "hello"
```

`std::move`

- forciert die Konvertierung zu rvalue
- Input muss nachher als zerstört oder leer behandelt werden
  - sollte aber immer noch einen validen Zustand haben

Beispiel: `rvalue Referenzen`

```
1  ExampleData ManipulateAndCopyReturn(ExampleData&& a) {
2    a.value = 99;            // manipuliert a
3    return a;                // kopiert a
4  }
5
6  ExampleData ManipulateAndMoveReturnLValue(ExampleData&& a) {
7    a.value = 99;            // manipuliert a
8    return std::move(a);     // verschiebt a und weist b zu
9    // gcc optimiert dies direkt mit RVO
10 }
11
12 ExampleData&& ManipulateAndMoveReturnRValue(ExampleData&& a) {
13   a.value = 99;            // manipuliert a
14   return std::move(a);     // verschiebt a direkt in b
15 }
16
17 ExampleData a;
18 auto b = Manipulate(std::move(a));
19 // a ist nun leer
```

Perfect-Forwarding

- durch Reference-Collapsing können auch lvalue-Referenzen in Funktionen mit rvalue-Referenz-Parameter verwendet werden.
- hier soll aber keine Forcierung auf rvalue stattfinden
- nur wenn wirklich eine rvalue übergeben wurde

```
1  template<class T>
2  void InsertInVector(std::vector* v, T&& value) {
3    v->emplace_back(std::forward<T>(value);
4  }
5
6  std::vector<int> v;
7  int i = 1;
8  InsertInVector(&v, i);      // value ist ein lvalue
9  InsertInVector(&v, 33);     // value ist ein rvalue
```

Wann anwenden? `std::move` oder `std::forward`?

- Move-Semantik
  - Meistens wollen Sie kein Move
  - Nur wenn Performance optimieren
  - `std::move` nur einsetzen wenn Sie Move-Semantik implementieren
  - oder wenn Sie explizit Move-Semantik forcieren möchten
- Perfect-Forwarding / rvalue-Referenzen
  - Nur in Template-Funktionen, sonst const-Ref
  - `std::forward` kann dann verwendet werden, wenn Move-Semantik unterstützt wird, z.B. in `std::vector::emplace_back()`