

# Programmieren in C++

Move Semantik und andere  
Performance-Verbesserungen



# Inhalt

- Hintergrund der Move-Semantik
- Optimierung bei Datenweitergabe
- Optimierung der Rückgabe
- Wertekategorien
- Temporäre Objekte und Rechtswert-Referenzen
- Besitzübernahme: Move-Semantik
- Rückgabebetyp und Move-Semantik

# Hintergrund der Move-Semantik (1)

- Bei grossen Objekten fällt der Kopieraufwand des Kopierkonstruktors bzw. des Zuweisungsoperators ins Gewicht

```
class RGBImage {
    size_t m_size;
    unique_ptr<uint32_t[]> m_data;
public:

    RGBImage(const RGBImage& im)
        : m_size(im.m_size)
        , m_data(make_unique<uint32_t[]>(m_size))
    {
        // copy data: O(m_size)
        std::copy(im.m_data.get(), im.m_data.get() + m_size, m_data.get());
    }

    RGBImage& operator=(const RGBImage& im) {
        if (&im != this) {
            m_size = im.m_size;
            m_data = move(make_unique<uint32_t[]>(m_size));
            // copy data: O(m_size)
            copy(im.m_data.get(), im.m_data.get() + m_size, m_data.get());
        }
    }
};
```

# Hintergrund der Move-Semantik (2)

- Wie könnte man eine ganze Liste von Bildern verwalten?
  - **Vektor von Rohzeigern:** `vector<RGBImage*> v1;`
    - sehr effizient, weil Vektorelemente problemlos umkopiert werden können
    - unsicher, weil mehrere Zeiger aufs gleiche Bild vorhanden sein können und jeder das Bild löschen kann
  - **Vektor von Zeigerobjekten:** `vector<shared_ptr<RGBImage>> v2;`
    - effizient, weil Vektorelemente problemlos umkopiert werden können
    - sicher, obwohl mehrere Zeiger aufs gleiche Bild vorhanden sein können
  - **Vektor von Objekten:** `vector<RGBImage> v3;`
    - in einem vector müssen Elemente oft vertauscht oder Elemente umkopiert werden, z.B. beim Sortieren
    - nur dann effizient, wenn Elemente (also RGBImages) effizient verschoben werden können, ohne dass die Bilddaten umkopiert werden müssen
    - sicher und leicht verständlich

# Effiziente Datenweitergabe

- Grosse Objekte sollten nicht unnötig kopiert werden
  - Kopieraufwand kann die Performance enorm beeinträchtigen
- Beispiel: Factory mit temporären Objekten
  - innerhalb einer (statischen) Methode werden Objekte erzeugt und über den Rückgabewert zurückgegeben (in einer Factory werden die Objekte selber nicht benötigt)
  - C++: in einer Factory können auch Objekte auf dem Stack erstellt werden; wird ein solches Objekt zurückgegeben, so muss es by value (also durch Kopieren) zurückgegeben werden

```
string Factory() {  
    string t("Text");           // das String-Objekt liegt auf dem Stack,  
                                // die eigentliche Zeichenkette evtl. auf dem Heap  
  
    return t;                   // normale Rückgabe by value  
}                                // t wird in ein temporäres Objekt kopiert  
  
...  
string s = Factory();           // das String-Objekt wird direkt in s erzeugt (copy elision)  
s = Factory();                  // das temporäre String-Objekt kann verschoben werden,  
                                // falls ein Verschiebeoperator vorhanden ist
```

# Optimierung bei Datenweitergabe

- Datenübergabe an Prozeduren/Funktionen/Methoden
  - Referenzen (l-value Referenzen)
    - lösen das Problem der effizienten Datenübergabe an Prozeduren
    - Daten werden nicht umkopiert, sondern innerhalb der Prozedur wird direkt auf die Originaldaten beim Aufruf zugegriffen
- Datenrückgabe bei Funktionen
  - Return Value Optimization (Spezialfall von copy elision)
    - an speziellen Stellen wird auf eine Kopie verzichtet (im Standard definiert)
    - Compiler-Hersteller kann weitere Optimierungen ermöglichen, wenn durch das Wegfallen der temporären Kopie nur die Performance erhöht wird, die Funktionalität sich aber nicht ändert
  - r-value Referenzen
    - sind Referenzen auf temporäre Objekte
    - ermöglichen dem Programmierer auf kontrollierte Art und Weise auf unnötige Datenkopien zu verzichten
- Zuweisung
  - r-value Referenzen wie bei der Datenrückgabe

# Return Value Optimization

```
struct C {  
    C() { cout << "std ctor" << endl;}  
    C(const C& c) { cout << "copy ctor" << endl;}  
    C& operator=(const C& c) { cout << "assignment" << endl; return *this;}  
};  
  
C foo() {  
    return C();  
}  
  
int main() {  
    C c = foo(); // nur std ctor; temporäres Objekt eliminiert  
                // ohne RVO können 1 oder 2 Aufrufe des copy ctor anfallen  
  
    c = foo();   // std ctor & assignment, temporäres Objekt eliminiert  
}
```

# C++ Ausdrücke

- C++-Ausdrücke können anhand zwei unabhängiger Eigenschaften charakterisiert werden
  - Datentyp
  - Wertekategorie
- jeder C++-Ausdruck
  - hat einen Nicht-Referenzdatentyp
  - und gehört einer von drei Wertekategorien an



# Wertekategorien

## ■ 2 Unterscheidungsmerkmale

- Ausdruck hat eine Identität
- Wert kann verschoben werden

## ■ Primäre Wertekategorien

- nur 3 der 4 möglichen Kombinationen werden in C++ verwendet
- typischer x-value: `std::move(x)`

## ■ Sekundäre Wertekategorien

- **gl-value (general left value)** = l-value  $\cup$  x-value
  - hat Identität
- **r-value (right value)** = x-value  $\cup$  pr-value
  - kann verschoben werden
  - Adressoperator kann nicht verwendet werden

	keine Identität	hat Identität	gl-value
kann nicht verschoben werden		<b>l-value</b>	
kann verschoben werden	<b>pr-value</b>	<b>x-value</b>	r-value

# Wertekategorien: Code-Beispiel

```
void test(int& x) {  
    cout << "non movable" << endl;  
}  
void test(int&& x) {  
    cout << "movable" << endl;  
}  
int main() {  
    int x = 5; cout << &x << endl;  
    test(x);  
    test(5);  
    {  
        int x = 5; cout << &x << endl;  
    }  
}
```

# Ausdrücke und Wertekategorien

- l-value (left value): hat Identität, nicht verschiebbar
  - Variable, Funktion, Klassenattribut, ...
  - Parameter, auch wenn vom Typ r-value Referenz
  - Funktionsaufruf mit Rückgabetyl l-value Referenz
  - Stringliteral
- pr-value (pure right value): keine Identität, verschiebbar
  - Literal: 42, true, nullptr, ...
  - arithmetischer Ausdruck:  $a + b$ ,  $a < b$ , ...
  - Funktionsaufruf: `str.substr(1, 2)`, ...
- x-value (expiring value): hat Identität, verschiebbar
  - Funktionsaufruf mit Rückgabetyl r-value Referenz
  - Array- oder Attributzugriff bei einem r-value
- weitere Informationen
  - [Theorie und Beispiele](#)
  - [type traits](#)



# Beispiele von Ausdrücken

```
struct C {  
    int n;  
};
```

...

```
C c;
```

4;	// pr-value	ist nur verschiebbar
C{4};	// pr-value	ist nur verschiebbar
c;	// l-value	hat nur Identität
c.n;	// l-value	hat nur Identität
std::forward<C&>(c);	// l-value	hat nur Identität
std::move(c);	// x-value	hat Identität und ist verschiebbar
C{4}.n;	// x-value	hat Identität und ist verschiebbar

# Temporäre Objekte (r-value)

- 2 Arten der Lebensverlängerung sind möglich

```
int main() {  
    std::string s1 = "Test";  
    std::string&& r1 = s1;           // r-value Referenz darf nicht zu l-value binden  
    std::string& r1 = s1 + s1;      // l-value Referenz darf nicht zu r-value binden  
  
    // konstante l-value Referenz darf zu temporärem Objekt (s1 + s1) binden  
    const std::string& r2 = s1 + s1; // verlängert die Lebensdauer des temp. Obj.  
    // r2 += "Test"; // von r2 gebundener const String darf nicht verändert werden  
  
    // r-value Referenz bindet zu temporärem Objekt (s1 + s1)  
    std::string&& r3 = s1 + s1;      // verlängert die Lebensdauer des temp. Objekts  
    r3 += "Test";                    // erlaubt, weil r3 eine r-value Referenz ist  
    std::cout << r3 << '\n';  
}
```

# Parameter vom Typ r-value Referenz

- Funktionsparameter sind innerhalb der Funktion l-values

```
string foo(string&& s) {  
    s += "456";           // innerhalb von foo ist s ein lvalue  
    return move(s);       // stellt sicher, dass Move-Semantik  
}                          // für die Rückgabe verwendet wird
```

...

zur Erzeugung von t wird der Verschiebekonstruktor verwendet

```
string t = foo(string("123")); // foo wird mit einem temporären  
                               // String-Objekt aufgerufen
```

```
string x;
```

```
foo(x);           // ein l-value darf nicht an foo übergeben werden
```



# Besitzübernahme (Move-Semantik)

## ■ Idee der Besitzübernahme

- temporäre Objekte werden kurz nach der Erstellung wieder zerstört
- werden dem temporären Objekt vor seiner Zerstörung die Daten entzogen, so stört das nicht weiter

```
class PointVector {  
    unique_ptr<Point[]> m_array;  
    size_t m_size;  
public:  
    PointVector(size_t s = 0) ...  
    void add(const Point& p) { ... }  
};  
PointVector create() {  
    PointVector v;  
    v.add(Point(1,2,3));  
    return v;  
}
```

```
int main() {  
    // Verschiebekonstruktor  
    PointVector pv1 = create();  
    PointVector pv2(create());  
  
    // Verschiebeoperator  
    PointVector pv3;  
    pv3 = create();  
}
```

Daten des PointVector aus create()  
an das neue Objekt übertragen!

# Umsetzung der Move-Semantik

```
class PointVector {
    unique_ptr<Point[]> m_array;
    size_t m_size;
public:
    // benötigt eigenen Standardkonstruktor und Destruktor
    // Verschiebekonstruktor
    PointVector(PointVector&& v) : m_array(std::move(v.m_array)), m_size(v.m_size) {
        v.m_size = 0;
    }

    // Verschiebeoperator
    PointVector& operator=(PointVector && v) {
        if (this != &v) {
            m_size = v.m_size; v.m_size = 0;
            m_array = std::move(v.m_array); // unique_ptr hat keinen Zuweisungsoperator
                                           // aber einen Verschiebeoperator
        }
        return *this;
    }
};
```

# std::exchange und std::swap

■ `T exchange( T& obj, U&& new_value );`

- ersetzt den Wert von obj mit dem neuen Wert new\_value und gibt den alten Wert von obj zurück
- eignet sich gut für die Implementierung des Verschiebekonstruktors

```
PointVector(PointVector&& v)
    : m_array(std::exchange(v.m_array, nullptr))
    , m_size(std::exchange(v.m_size, 0))
    {}
```

■ `void swap( T& a, T& b );`

- vertauscht die Werte der beiden Variablen a und b
- eignet sich gut für die Implementierung des Verschiebeoperators

```
PointVector& operator=(PointVector && v) {
    if (this != &v) {
        std::swap(m_size, v.m_size);
        std::swap(m_array, v.m_array);
    }
    return *this;
}
```



# std::move

## ■ std::move(T x)

- ist im Wesentlichen ein Typkonvertierungsoperator, um aus x eine rvalue Referenz zu machen: `static_cast<T&&>(x)`
- verschiebt selber gar nichts
- stellt sicher, dass der Compiler einen allfälligen Verschiebekonstruktor bzw. Verschiebeoperator anstatt dem Kopierkonstruktor bzw. Zuweisungsoperator aufruft

## ■ Einsatzzweck

- Aufruf des Verschiebekonstruktors/-operators erzwingen

## ■ Beispiel

```
std::string s1 = "hello";  
std::string s2 = std::move(s1); // Verschiebekonstruktor  
// s1 == ""                    // hinterlässt in s1 gültiges Objekt  
// s2 == "hello"               // aber die Daten sind nun in s2
```

# Default-Methoden

## ■ Verschiebeoperationen

- nur wenn keine Kopieroperationen und kein Destruktor definiert worden sind
- wird ein eigener Verschiebekonstruktor angeboten, so sollte auch der Verschiebeoperator implementiert werden
- Verschiebekonstruktor `C::C(C&&)`
- Verschiebeoperator `C& operator=(C&&)`

# Rückgabetyt und Move-Semantik

```
string foo(string&& s) {  
    s += "456";  
    return move(s);  
}
```

// verwendet Move-Konstruktor  
// bei der Erstellung des temp. Objekts

  

```
string&& foo(string&& s) {  
    s += "456";  
    return move(s);  
}
```

// Rückgabetyt r-value-Referenz ist  
// hier möglich  
// die Erstellung des temp. Objektes  
// kann ganz wegoptimiert werden