Fachhochschule Nordwestschweiz
Hochschule für Technik

Programmieren in C++

Christian Lang (Lac)

8. November 2019

Operatoren

Inhalt

- Überladen von Methoden
- Overload Resolution
- Überladen von Operatoren
- Übliche Operatoren
- Signatur-Beispiele
- Implementations-Beispiele
- Operatoren mit Move-Semantik
- Freie Funktionen und `friend`
- Operatoren und Vererbung

Überladen von Methoden

- Methoden können mit unterschiedlicher Signatur überladen werden
- Signatur besteht aus:
  - Namespace
  - Klasse
  - Methoden-Name
  - Parameterliste (nur Typen)
- Return-Typ ist nicht Teil der Signatur
- Signatur muss immer eindeutig sein

```
1  struct Point {
2    Point& Move(double x, double y = 0, double z = 0);
3    Point& Move(double delta[3]);
4    Point& Move(const Point& p);
5  };
```

Overload Resolution

- Generell
  - Auflösen der Funktions-Aufrufe wird zur Compiletime gemacht
  - ein impliziter Cast für jeden Parameter ist erlaubt
  - nur Signatur ist relevant
- in Klassenhierarchien
  - überladene Methoden-Namen verdecken die Basis-Methoden
  - Basis-Methoden mittels `using Father::Foo;` sichtbar machen

```
1  struct Father {
2    void Foo(char);
3  };
4  struct Son : Father {
5    int Foo(int);
6  };
7
8  Son s;
9  s.Foo(5);       // Aufruf von Son::Foo
10 s.Foo('A');     // Aufruf von Son::Foo (weil Father::Foo verdeckt)
```

Überladen von Operatoren

- nicht nur Methoden sondern auch Operatoren können überladen werden
- bekanntes Beispiel: `operator<<`
- erlaubt schönere Syntax als mit Methoden

```
1  struct Complex {
2    ...
3
4    Complex operator+(const Complex& rhs) const {
5      return Complex(real + rhs.real, imaginary + rhs.imaginary);
6    }
7  };
8
9  Complex c1(2, 4);
10 Complex c2(2, -4);
11 Complex c = c1 + c2;
```

Grundregeln für Operator-Overloading

- neue Operatoren können nicht definiert werden
- Überladen von `&&` und `||` deaktiviert short-circuit-Evaluierung
- vorgegebene Vorrangregeln dürfen nicht verletzt werden
- auch hier ist Return-Typ nicht relevant
- mindestens ein Argument muss ein Objekt sein oder der Operator muss eine Instanzmethode sein
  - verhindert, dass Operatoren der primitiven Typen verändert werden
- nicht erlaubt:
  - `.`
  - `.*`
  - `::`
  - `? :`

Übliche Operatoren

assignment

```
a = b
a += b
a -= b
a *= b
a /= b
a %= b
a &= b
a |= b
a ^= b
a <<= b
a >>= b
```

increment decrement

```
++a
--a
a++
a--
```

arithmetic

```
+a
-a
a + b
a - b
a * b
a / b
a % b
~a
a & b
a | b
a ^ b
a << b
a >> b
```

logical

```
!a
a && b
a || b
```

comparison

```
a == b
a != b
a < b
a > b
a <= b
a >= b
a <=> b
```

member access

```
a[b]
*a
&a
a->b
a->*b
```

other

```
a(...)
a, b
new
new[]
delete
delete[]
```

Genauere Infos

cppreference.com - operator overloading

Signatur-Beispiele: Operator-Overloading für T

```
1  T& operator=(const T& other);   // copy-assign
2  T& operator=(T&& other);        // move-assign
3
4  // stream operators
5  std::ostream& operator<<(std::ostream& os, const T& obj);
6  std::istream& operator>>(std::istream& is, T& obj);
7
8  void operator()(int n);         // functor
9
10 T& operator++()                 // prefix increment
11 T operator++(int)               // postfix increment (called with 0)
12
13 // comparison
14 inline bool operator< (const T& lhs, const T& rhs);
15 inline bool operator==(const T& lhs, const T& rhs);
```

Relationale Operatoren

- typischerweise nur `operator<` und `operator==` implementieren
- restliche können anhand dieser beiden implementiert werden

```
1  bool operator!=(const X& l, const X& r) { return !operator==(l, r); }
2  bool operator> (const X& l, const X& r) { return  operator< (r, l); }
3  bool operator<=(const X& l, const X& r) { return !operator> (l, r); }
4  bool operator>=(const X& l, const X& r) { return !operator< (l, r); }
```

- oder `namespace std::rel_ops` wird verwendet, welcher diese trivialen Implementation für generische Typen zur Verfügung stellt

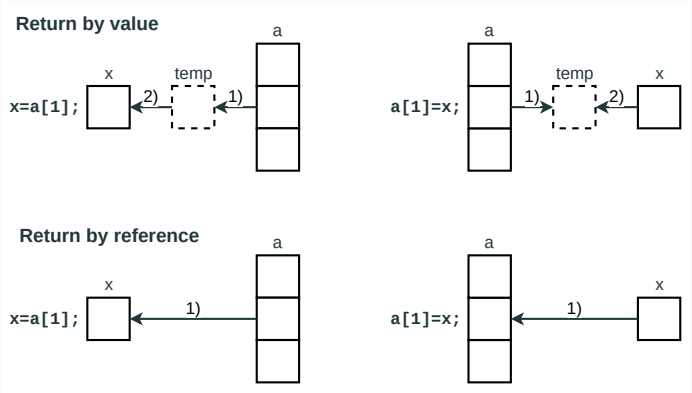
```
1  <utility>
2  using namespace std::rel_ops;
```

- seit C++20 soll nur noch `operator<=>` implementiert werden

Index Operator

- üblicherweise als veränderbar und als `const` implementieren
- üblicherweise ohne Range-Check
- Best-Practice: Methode `at(size_t)` mit Range-Check

```
1  T operator[](size_t idx) const;
2  T& operator[](size_t idx);
```



Typkonvertierungs-Operationen

- Typen können implizit oder explizit konvertiert werden
- Compiler darf maximal eine implizite Konvertierung vornehmen
- kann mit `explicit` verhindert werden
- Konstruktor erlaubt Konvertierung von etwas anderem
- `operator T` erlaubt Konvertierung zu etwas anderem

```
1  struct Point {
2    explicit Point(int a);                 // von int
3    Point(int a, int b = 0);               // aufrufbar ohne b
4
5    explicit void operator int() const;    // zu int
6    void operator int*() const;            // zu int*
7  };
```

Verwendung impliziter Konvertierungen

- Nur dort wo Konvertierung semantisch Sinn macht

Operatoren mit Move-Semantik

Move-Semantik kann auch bei anderen Operatoren oder Methoden Sinn machen.

```
1  class Point;
2
3  Point operator+(const Point& a, const Point& b);     // a + b
4  Point operator+(const Point& a, Point&& cd);         // a + c*d
5  Point operator+(Point&& ab, const Point& c);         // a*b + c
6  Point operator+(Point&& ab, Point&& cd);             // a*b + c*d
7
8  Point operator+(const Point& a, Point&& cd) {
9    cd += a;
10   return std::move(cd);
11 }
```

Freie Funktionen und `friend`

- Operatoren und Methoden können auch als freie Funktionen implementiert werden
- nur Zugriff auf `public` Member
- kann mittels `friend` erweitert werden
- hilft "tight coupling" zu reduzieren
- erlaubt Compiler mehr implizite Konversionen

```
1  class Point {
2    friend bool operator<(const Point& lhs, const Point& rhs) {
3      return lhs.x < rhs.x || ... ;
4    }
5    friend bool operator==(const Point& lhs, const Point& rhs);
6  };
7
8  bool operator==(const Point& lhs, const Point& rhs) {
9    return lhs.x == rhs.x && ... ;
10 }
```

Operatoren und Vererbung

- auch automatisch synthetisierte Operatoren sind nie `virtual`
- per default immer statische Bindung verwendet
- gilt für Pointer- und Referenz-Typen

```
1 // Beispiel Assigment-Operator bei Klasse Student
2 Student stud1;
3 Student stud2;
4 Person& pers = stud1;
5 pers = stud2;           // nur Personen-Attribute werden kopiert
6 stud2 = pers;           // nicht möglich, weil nur eine Personen-Projektion
```

macht selten Sinn

- kopieren Sie Instanzen nur über spezifische Pointer/Referenzen
- bei Bedarf zuerst `dynamic_cast`