6 Überladene Operatoren

6.1 Allgemeines Prinzip

6.1.1 Definition von Operatoren durch gewöhnliche Funktionen

```
// Rationale Zahl (vgl. §5.1.1).
struct Rational {
  // Zähler (numerator) und Nenner (denominator).
  const int num, den;

  // Konstruktor.
  explicit Rational (int n = 0, int d = 1) : num{n}, den{d} {}
};
```

```
// Differenz der rationalen Zahlen x und y (binärer Operator).
Rational operator- (Rational x, Rational y) {
  return Rational{x.num * y.den - y.num * x.den, x.den * y.den};
}

// Negation der rationalen Zahl x (unärer Operator).
Rational operator- (Rational x) {
  return Rational{-x.num, x.den};
}

// Mögliche Verwendung.
Rational r1{1, 2};
Rational r2{2, 3};
Rational r3 = -r1 - r2;
```

6.1.2 Definition von Operatoren durch Elementfunktionen

```
// Rationale Zahl.
struct Rational {
  // Zähler (numerator) und Nenner (denominator).
  const int num, den;

  // Konstruktor.
  explicit Rational (int n = 0, int d = 1) : num{n}, den{d} {}

  // Differenz der rationalen Zahlen *this und y (binärer Operator).
  Rational operator- (Rational y) const {
    return Rational{num * y.den - y.num * den, den * y.den};
  }

  // Negation der rationalen Zahl *this (unärer Operator).
  Rational operator- () const {
    return Rational{-num, den};
  }
};

// Verwendung wie zuvor.
```

6.1.3 Erläuterungen

- Das Schlüsselwort `operator`, gefolgt von einem Operatorsymbol, entspricht syntaktisch einem Funktionsnamen.

- Ein Ausdruck wie z. B. `-r1 - r2` mit Teilausdrücken `r1` und `r2` des Typs `Rational` wird vom Übersetzer entweder in gewöhnliche Funktionsaufrufe oder in Aufrufe von Elementfunktionen (oder eine passende Mischform) transformiert:

```
// Wenn Operatoren durch gewöhnliche Funktionen definiert wurden:
operator-(operator-(r1), r2)

// Wenn Operatoren durch Elementfunktionen definiert wurden:
r1.operator-().operator-(r2)
```

- Normalerweise ist die Definition durch gewöhnliche Funktionen und die Definition durch Elementfunktionen äquivalent.

- Die Operatoren = (Zuweisung), `[]` (Indexoperation), `()` (Funktionsaufruf) und -> (Elementzugriff über Zeiger) können jedoch nur als Elementfunktionen definiert werden.

- Der Operator -> wird hierbei als unärer Postfix-Operator interpretiert, auf dessen Resultat der Operator dann erneut angewandt wird.

- Die Operatoren `::` (Qualifizierung von Namen), `.` (Elementzugriff), `.*` (Elementzugriff über sog. Elementzeiger) und `?:` (Verzweigung) können grundsätzlich nicht überladen werden.

- Man kann weder neue Operatorsymbole einführen noch die Regeln für Vorrang und Assoziativität der vorhandenen Operatoren verändern.

- Mindestens ein Operand eines überladenen Operators muss ein benutzerdefinierter Typ (d. h. eine Klasse/Struktur/Überlagerung oder ein Aufzählungstyp) sein, d. h. die Bedeutung der vordefinierten Operatoren kann nicht verändert werden.

- Bei den Funktionen zur Definition überladener Operatoren kann es sich auch um Schablonen handeln.

- Ebenso wie bei gewöhnlichen Funktionsaufrufen ist es undefiniert, ob vor dem Aufruf eines binären Operators zuerst sein linker oder sein rechter Operand ausgewertet wird.

6.2 Vergleichsoperatoren

- Seit C++20 gelten die im folgenden beschriebenen zusätzlichen Regeln, um den Aufwand zur Definition von Vergleichsoperatoren zu reduzieren.

6.2.1 Gleichheitsoperatoren mit vertauschten Operandentypen

- Bei der Interpretation eines Vergleichs `x == y` werden zusätzlich zu den vordefinierten und explizit definierten (gewöhnlichen oder Element-) Funktionen `operator==` mit Operandentypen `U` und `V` auch die entsprechenden *synthetisierten* Funktionen mit den vertauschten Operandentypen `V` und `U` betrachtet (sofern `U` und `V` verschieden sind).

- Wenn die am besten passende Funktion dann eine dieser synthetisierten Funktionen ist, wird der Ausdruck `x == y` durch den Funktionsaufruf `operator==(y, x)` bzw. `y.operator==(x)` ersetzt, wobei `operator==` die zugehörige vordefinierte oder explizit definierte Funktion bezeichnet.

- Zum Beispiel:

```
// Vergleich der rationalen Zahl x mit der ganzen Zahl y
// oder umgekehrt.
bool operator== (Rational x, int y) {
    return x.num == y * x.den;
}

// Mögliche Verwendung.
Rational r{6, 3};
int i = 2;

// Ausdruck:    // Bedeutung:           // Oder (falls der obige
                                        // Operator als Element-
                                        // funktion von Rational
                                        // definiert ist):
r == i;         // operator==(r, i)     // r.operator==(i)
i == r;         // operator==(r, i)     // r.operator==(i)
```

- Vor C++20 hätte man zusätzlich `operator== (int, Rational)` definieren müssen, damit der Vergleich `i == r` vom Übersetzer akzeptiert wird.

6.2.2 Implizit definierte Ungleichheitsoperatoren

- Bei der Interpretation eines Vergleichs `x != y` werden zusätzlich zu den vordefinierten und explizit definierten Funktionen `operator!=` auch *umgeschriebene* Funktionen betrachtet, die sowohl aus den vordefinierten und explizit definierten als auch aus den gemäß § 6.2.1 synthetisierten Funktionen `operator==` gebildet werden.

- Wenn die am besten passende Funktion dann eine dieser umgeschriebenen Funktionen ist, wird der Ausdruck `x != y` durch `!(x == y)` ersetzt, was gemäß § 6.1.3 wiederum durch `!operator==(x, y)` bzw. `!x.operator==(y)` oder gemäß § 6.2.1 durch `!operator==(y, x)` bzw. `!y.operator==(x)` ersetzt wird.

- Zum Beispiel:

```
// Ausdruck:     // Bedeutung:              // Oder:
r != i;          // !operator==(r, i)      // !r.operator==(i)
i != r;          // !operator==(r, i)      // !r.operator==(i)
```

- Vor C++20 hätte man zusätzlich `operator!=(Rational, int)` und `operator!=(int, Rational)` definieren müssen, damit die Vergleiche `r != i` und `i != r` vom Übersetzer akzeptiert werden. (Das heißt, man hätte insgesamt vier statt nur einer Funktion definieren müssen, um rationale und ganze Zahlen in beiden Richtungen auf Gleichheit und Ungleichheit testen zu können.)

6.2.3 Dreiwegvergleich

Grundprinzip

- Seit C++20 gibt es neben den sechs gewohnten Vergleichsoperatoren <, <=, ==, !=, >=, > einen weiteren Operator <=>, der als *Dreiwegvergleich* bezeichnet wird.
- Sein Resultattyp ist normalerweise einer der Typen `std::strong_ordering`, `std::weak_ordering` oder `std::partial_ordering`.
- Jeder dieser Typen besitzt die Werte `less`, `equivalent` und `greater` (z. B. `strong_ordering::less` oder `partial_ordering::equivalent`), die ausdrücken, dass der linke Operand eines Dreiwegvergleichs kleiner bzw. äquivalent bzw. größer als der rechte Operand ist.
- `strong_ordering` besitzt zusätzlich den Wert `equal`, der hier aber gleichbedeutend mit `equivalent` ist.
- `partial_ordering` besitzt zusätzlich den Wert `unordered`, der ausdrückt, dass die beiden Operanden nicht vergleichbar sind, d. h. dass der linke Operand weder kleiner noch äquivalent noch größer als der rechte Operand ist.
- Jeder dieser Werte kann mit jedem Vergleichsoperator in beiden Richtungen mit dem `int`-Wert 0 verglichen werden, wobei `less` kleiner, `equivalent` und `equal` gleich, `greater` größer und `unordered` weder kleiner noch gleich noch größer als 0 ist. (Wenn ein solcher Vergleich wiederum ein Dreiwegvergleich ist, erhält man einen Wert desselben Typs.)

Beispiele

```
// Vergleich:                          // Resultatwert:
1 <=> 2                                strong_ordering::less
1 <=> 1                                strong_ordering::equal/equivalent
2 <=> 1                                strong_ordering::greater

1.0 <=> 2.0                            partial_ordering::less
1.0 <=> 1.0                            partial_ordering::equivalent
2.0 <=> 1.0                            partial_ordering::greater
-0.0 <=> +0.0                          partial_ordering::equivalent
double nan = 0.0/0.0;
1.0 <=> nan                            partial_ordering::unordered
nan <=> 1.0                            partial_ordering::unordered
nan <=> nan                            partial_ordering::unordered

1 <=> 2 < 0                            true
1 <=> 2 <= 0                           true
1 <=> 2 == 0                           false
1 <=> 2 != 0                           true
1 <=> 2 >= 0                           false
1 <=> 2 > 0                            false
1 <=> 2 <=> 0                          strong_ordering::less
```

Erläuterungen

- Die `double`-Werte `-0.0` und `+0.0` sind äquivalent, obwohl sie nicht exakt gleich sind. Beispielsweise liefert die Division `1.0/+0.0` einen positiven unendlichen Wert, während `1.0/-0.0` einen negativen unendlichen Wert liefert.

- Die Division `0.0/0.0` liefert einen NaN-Wert (not a number), der weder kleiner noch äquivalent noch größer als jeder `double`-Wert (einschließlich NaN-Werten selbst!) ist.

- Der Dreiwegvergleich `<=>` ist linksassoziativ und bindet stärker als die relationalen Operatoren `<`, `<=`, `>=` und `>`, die wiederum stärker als die Gleichheitsoperatoren `==` und `!=` binden.

- Dementsprechend ist `1 <=> 2 < 0` z. B. gleichbedeutend mit `(1 <=> 2) < 0` und `1 <=> 2 <=> 0` gleichbedeutend mit `(1 <=> 2) <=> 0`.

- Ein typisches Beispiel für `weak_ordering` ist ein lexikographischer Vergleich von Zeichenketten ohne Berücksichtigung von Groß- und Kleinschreibung, wo beispielsweise `"abc"` und `"ABC"` äquivalent, aber nicht exakt gleich sind.

Benutzerdefinierte Dreiwegvergleiche

```
// Vergleich der rationalen Zahlen x und y.
strong_ordering operator<=> (Rational x, Rational y) {
  // Jeden Zähler mit dem jeweils anderen Nenner multiplizieren
  // und die Ergebnisse vergleichen.
  strong_ordering r = x.num * y.den <=> y.num * x.den;

  // Wenn beide Nenner dasselbe Vorzeichen besitzen,
  // bleibt die Relation durch die Multiplikationen erhalten,
  // andernfalls dreht sie sich um.
  return (x.den > 0) == (y.den > 0) ? r : 0 <=> r;
}
```

6.2.4 Dreiwegvergleich mit vertauschten Operandentypen

- Bei der Interpretation eines Dreiwegvergleichs `x <=> y` werden zusätzlich zu den vordefinierten und explizit definierten Funktionen `operator<=>` mit Operandentypen `U` und `V` auch die entsprechenden synthetisierten Funktionen mit den vertauschten Operandentypen `V` und `U` betrachtet (sofern `U` und `V` verschieden sind).

- Wenn die am besten passende Funktion dann eine dieser synthetisierten Funktionen ist, wird der Ausdruck `x <=> y` durch `0 <=> (y <=> x)` ersetzt, wobei `<=>` innerhalb der Klammer den zugehörigen vordefinierten oder explizit definierten Operator bezeichnet.

6.2.5 Implizit definierte relationale Operatoren

- Bei der Interpretation eines Vergleichs `x @ y`, wobei `@` einer der vier relationalen Operatoren `<`, `<=`, `>=` oder `>` ist, werden neben den jeweiligen vordefinierten und explizit definierten Operatoren auch umgeschriebene Funktionen betrachtet, die sowohl aus den vordefinierten und explizit definierten als auch aus den gemäß § 6.2.4 synthetisierten Funktionen `operator<=>` gebildet werden.

- Wenn die am besten passende Funktion dann eine dieser umgeschriebenen Funktionen ist, wird der Ausdruck `x @ y` durch `x <=> y @ 0` ersetzt, wobei `<=>` den zugehörigen vordefinierten, explizit definierten oder synthetisierten Operator bezeichnet (der dann ggf. wieder gemäß § 6.2.4 interpretiert wird).

- Zum Beispiel:

```
// Ausdruck:                  // Bedeutung:
r < r                         // (r <=> r) < 0
r >= r                        // (r <=> r) >= 0
```

- Vor C++20 hätte man vier statt nur einer Funktion definieren müssen, damit die vier relationalen Vergleiche `r @ r` vom Übersetzer akzeptiert werden.

- Fortsetzung des Beispiels:

```
// Vergleich der rationalen Zahl x mit der ganzen Zahl y
// oder umgekehrt.
strong_ordering operator<=> (Rational x, int y) {
    return x <=> Rational{y, 1};
}

// Ausdruck:                  // Bedeutung:
r < i                         // r <=> i < 0
i >= r                        // 0 <=> (r <=> i) >= 0 bzw.
                              // r <=> i <= 0
```

- Vor C++20 hätte man zusätzlich acht statt nur einer Funktion definieren müssen, damit die relationalen Vergleiche `r @ i` und `i @ r` vom Übersetzer akzeptiert werden.

6.2.6 Allgemeine Regeln

- Wenn eine vordefinierte oder explizit definierte und eine synthetisierte oder umgeschriebene Funktion gleich gut passen (aber nur dann), wird die vordefinierte bzw. explizit definierte Funktion bevorzugt.

- Wenn zwei umgeschriebene Funktionen gleich gut passen, von denen eine direkt aus einer vordefinierten oder explizit definierten Funktion und die andere aus einer synthetisierten Funktion mit vertauschten Operandentypen gebildet wurde, wird erstere bevorzugt.

6.2.7 Anmerkungen

- Dreiwegvergleiche werden nur bei der Interpretation der vier relationalen Vergleiche <, <=, >= und > berücksichtigt, aber nicht bei Tests auf Gleichheit und Ungleichheit.

- Obwohl die neuen Regeln von C++20 die Definition von Vergleichsoperatoren zum Teil erheblich vereinfachen, führen sie gelegentlich auch dazu, dass Code, der bis C++17 korrekt funktioniert hat, mit C++20 nicht mehr funktioniert, weil Vergleiche aufgrund der neuen Regeln plötzlich anders als zuvor interpretiert werden.

Beispiel

```
struct A { ..... };
struct B : A { ..... };

bool operator== (A x, A y) { ..... }   // AA
bool operator== (A x, B y) { ..... }   // AB

// Ausdruck: // Aufruf von Operator AA oder AB?
A a; B b;    // Bis C++17:          // Ab C++20:

a == b;      // AB                  // AB

b == a;      // AA                  // AB mit vertauschten Operanden
             // (AB passt nicht)    // (passt besser als AA)

b == b;      // AB                  // AB, aber eigentlich mehrdeutig:
                                    // AB oder
                                    // AB mit vertauschten Operanden
```