Vorlesung

Objektorientiertes Programmieren in C++

Teil 9 - WS 2022/23

Detlef Wilkening
www.wilkening-online.de
© 2022

16 Operator-Funktionen

„Lehnen Sie sich zurück und entspannen Sie sich“ – so oder ähnlich könnte dieses Kapitel heissen, denn prinzipiell bringt es fast gar nichts Neues. Lesen Sie Kapitel 16.1 – und Sie wissen fast alles, was Sie wissen müssen. Nun gut, an einzelnen Stellen gibt es noch ein paar Details zu beachten, aber meistens gibt es selbst diese Komplikationen nicht.

Aber warum weisse ich extra darauf hin?
Und warum ist dann das Kapitel nicht nur 2 Seiten lang?

Das hat was mit meinen Erfahrungen zu tun – eigenen und fremden. Aus irgendeinem mir unbekanntem Grund haben Operator-Funktionen das Stigma des Mystischen, des Undurchschaubaren, des unheimlich Komplizierten – ich habe Leute kennengelernt, die seit Jahren C++ programmierten und dabei schon recht abgedrehte Sachen machten, aber sich nie getraut haben Operatoren zu überladen.

Woran das liegt? Keine Ahnung, denn in Wirklichkeit sind Operator-Funktionen gar nicht kompliziert, ganz im Gegenteil. Aber scheinbar wirken sie so fremd, dass man erstmal die Finger von ihnen lässt.

Und warum ist das Kapitel dann soooooo lang? Genau genommen eigentlich nur, damit es einige Details erklärt und einige Beispiele enthält.

Aber grundsätzlich gilt: keine Panik, dieses Kapitel ist wirklich halb so wild.

16.1 Einführung

In einer fiktiven Klasse „rational“ (bruch) müßten bislang mathematische Operationen als Element-Funktionen abgebildet werden - z.B. um Brüche zu multiplizieren. Die möglichen Element-Funktionen wären z.B. „mul“ für die Multiplikation oder „mul_assign“ für die multiplikative Zuweisung, deren Verhalten der normalen Operator-Semantik von „*=“ bzw. „*“ z.B. bei Ints entsprechen würden. Für den Benutzer der Klassen wäre es sicher angenehmer, wenn er statt der Element-Funktionen diese Operatoren zur Verfügung hätte.

Bisheriges Wissen	**Schöner wäre**

```
rational r1, r2, r3;
r1 = r2.mul(r3);

rational r1, r2;
r1.mul_assign(r2);
```

```
rational r1, r2, r3;
r1 = r2*r3;

rational r1, r2;
r1 *= r2;
```

Hinweis – ich erlebe immer wieder, dass viele Einsteiger die Funktionalitäten wie die „Multiplikation" oder die „Multiplikative-Zuweisung" als freie Funktionen implementieren. Das geht natürlich auch – selbst Klassen-Funktionen wären möglich – aber da die Funktionen Zugriff auf die Attribute der Klasse benötigen und ja logisch zur Klasse gehören, sind Element-Funktionen meist die beste Wahl.

In C++ können die meisten der vorhandenen Operatoren überladen werden. Ein Operator ist im Prinzip eine ganz normale freie Funktion oder eine ganz normale Element-Funktion - bis auf:

- Operatoren müssen mit dem Schlüsselwort **operator** und dem Operator selber als Name deklariert und definiert werden.
- Es muss mindestens ein Parameter ein benutzerdefinierter Typ sein.
- Der Aufruf einer Operator-Funktion ist sowohl in Funktions-Schreibweise, als auch in Operator-Schreibweise möglich – siehe folgendes Beispiel.
- Die Anzahl der Parameter ist durch den Operator festgelegt – Ausnahme Aufruf-Operator. Z.B. der binäre Operator „+" (die Addition) hat zwei Operanden – und das können Sie nicht ändern.
- Im Normallfall sind keine Default-Argumente zugelassen, da sie die syntaktische Eindeutigkeit verletzen würden (Ausnahme: Aufruf-Operator).
- Die genaue Syntax der Deklaration und Definition ist von der Verwendung der Operators als freie Funktion oder als Element-Funktion abhängig.

Syntax

Dekl.: Rückgabetyp operator @ (Parameterliste);
Def.: Rückgabetyp operator @ (Parameterliste) { Funktionsrumpf }
oder Rückgabetyp Klassen-Name::operator @ (Parameterliste) { Funktionsrumpf }

Hinweis – das „@" steht hier für der erlaubten Operatoren wie z.B. „+" oder „*".

Implementieren wir als Beispiel die Multiplikation für eine ganz einfache Bruch-Klasse „rational" – sowohl in der bisherige Form (Element-Funktion „mul") als auch in der neuen Form (Element-Operator-Funktion „*").

Beispiel für die Element-Operator-Funktion „*"

```
#include <iostream>
using namespace std;

class rational
{
public:
   rational(int n=0, int d=1) : numerator_(n), denominator_(d) {}

   rational mul(const rational&) const;          // bisheriges Wissen
```

```
   rational operator*(const rational&) const;   // neues Wissen

   void print() const { cout << numerator_ << '/' << denominator_; }

private:
   int numerator_, denominator_;
};

// Man koennte die beiden Funtionen auch problemlos in einer Zeile implementieren.
// Hier ist es nicht geschehen, um klar zu zeigen, was in ihr passiert.
//
// So wuerden die Funktionen in einer Zeile aussehen:
//    return rational(numerator_ * rhs.numerator_, denominator_ * rhs.denominator_);
//
// Ausserdem sind die Funktionen gute Kandidaten fuer inline-Funktionen.

// Bisheriges Wissen
rational rational::mul(const rational& rhs) const
{
   int n = numerator_ * rhs.numerator_;
   int d = denominator_ * rhs.denominator_;
   rational result(n, d);
   return result;
}

// Neues Wissen
rational rational::operator*(const rational& rhs) const
{
   int n = numerator_ * rhs.numerator_;
   int d = denominator_ * rhs.denominator_;
   rational result(n, d);
   return result;
}

int main()
{
   rational r, r1(2, 3), r2(4, 5);

   r = r1.mul(r2);          // Bisheriges Wissen
   r.print();
   cout << '\n';

   r = r1.operator*(r2);    // Operator-Aufruf in Funktions-Schreibweise
   r.print();
   cout << '\n';

   r = r1 * r2;             // Operator-Aufruf in Operator-Schreibweise
   r.print();
   cout << '\n';
}
```

Ausgabe

```
8/15
8/15
8/15
```

Ich hoffe, Sie sehen wie gleich unsere bisherige Lösung mit Element-Funktion „mul“ und die neue Operator-Funktion „*“ ist. Abgesehen vom Namen und dem möglichen Aufruf in Operator-Schreibweise gibt es keinen Unterschied.

Und da man den Operator „*“ auch als freie Funktion implementieren kann, setzen wir auch dies einmal um – sowohl für eine normale freie Funktion „mul“ als auch für eine freie Operator-Funktion „*“.

Beispiel für die freie Operator-Funktion „*“

```
#include <iostream>
using namespace std;
```

```
class rational
{
public:
   rational(int n=0, int d=1) : numerator_(n), denominator_(d) {}

   // Jetzt benoetigen wir Getter fuer Zaehler und Nenner
   // Oder wir machen die Funktionen zu Friends
   int numerator() const { return numerator_; }
   int denominator() const { return denominator ; }

   void print() const { cout << numerator_ << '/' << denominator_; }

private:
   int numerator_, denominator_;
};

// Deklarationen der freien Funktionen
rational mul(const rational&, const rational&);
rational operator*(const rational&, const rational&);

// - Definitionen der freien Funktionen
// - Die Implementierung waere natuerlich auch hier in einer Zeile moeglich:
//   return rational(lhs.numerator()*rhs.numerator(),lhs.denominator()*rhs.denominator());
// - Als 'friend' Funktion koennte man sich die Get-Funktionen sparen,
//   was sicher sinnvoller waere.
// - Und natuerlich wieder ein super Kandidat fuer eine inline-Funktion.

rational mul(const rational& lhs, const rational& rhs)
{
   int n = lhs.numerator() * rhs.numerator();
   int d = lhs.denominator() * rhs.denominator();
   rational result(n, d);
   return result;
}

rational operator*(const rational& lhs, const rational& rhs)
{
   int n = lhs.numerator() * rhs.numerator();
   int d = lhs.denominator() * rhs.denominator();
   rational result(n, d);
   return result;
}

int main()
{
   rational r, r1(2, 3), r2(4, 5);

   r = mul(r1, r2);          // Bisheriges Wissen
   r.print();
   cout << '\n';

   r = operator*(r1, r2);    // Operator-Aufruf in Funktions-Schreibweise
   r.print();
   cout << '\n';

   r = r1 * r2;              // Operator-Aufruf in Operator-Schreibweise
   r.print();
   cout << '\n';
}
```

Ausgabe

```
8/15
8/15
8/15
```

Beide Lösungen sind prinzipiell identisch – aber beide haben ihre Vorteile:

Element-Operator-Funktionen:

- Sie sind semantisch eindeutig der Klasse zugeordnet.
- Sie haben Zugriff auf alle Elemente der Klasse.

- Sie können überschrieben werden – siehe später

Freie Operator-Funktionen:
- Sie können symmetrisch arbeiten (siehe Kapitel 16.3).
- Sie können für fremde Klassen definiert werden (siehe Kapitel 16.4).
- Sie können für Enums definert werden.
- Sie müssen aber oft als **friend** deklariert werden, oder mit einer Indirektion implementiert sein, um ihre Aufgabe erfüllen zu können. (Für Polymorphie müssen freie Operator-Funktionen mit einer Indirektion implementiert sein

16.2 Verständnis Beispiel

Nur noch mal zum Verständnis – da ich es fast jedes Mal erlebe, dass Studenten eine Element-Operator-Funktion folgendermassen deklarieren und definieren wollten:

```
class A
{
public:
   A operator+(const A&, const A&);    // Compiler-Fehler - dies sind drei Parameter
};
```

Die häufig gehörte „falsche“ Begründung für diesen Code ist, dass der Additions-Operator „+“ doch zwei Parameter benötigt. Das ist natürlich richtig, aber hierbei wird der implizite Objekt-Parameter „this“ vergessen. Auch eine Operator-Element-Funktion ist eine Element-Funktion, und kann damit nur für ein Objekt der Klasse aufgerufen werden, und dieses wird der Element-Funktion implizit mitgegeben – Sie erinnern sich an den ominösen Parameter „this“?

Wird eine Operator-Element-Funktion in Operator-Schreibweise aufgerufen, so ist das erste Argument immer das Objekt, für das die Operator-Element-Funktion aufgerufen wird. Und damit ist dieses Objekt direkt ansprechbar.

16.3 Symmetrische Operatornutzung

Element-Operator-Funktionen können nur mit einem Objekt als erstem Argument aufgerufen werden. Freie Operator-Funktionen können dagegen auch für das erste Argument die implizite Typumwandlung nutzen - damit kann der Operator flexibler genutzt werden.

Achtung – wir haben das komplexe und dunkle Thema Typumwandlungen bislang nicht detailliert besprochen, und werden dass auch weiterhin aus Zeitmangel nicht machen. Nehmen sie hier einfach hin, dass diese Umwandlungen erlaubt sind, und vom Compiler automatisch gemacht werden.

```
class A                          // Klasse A mit freier Operator-Funktion +
{
public:
   A();
   A(int);
};
```

```
A operator+(const A&, const A&);

class B                              // Klasse B mit Element-Operator-Funktion +
{
public:
   B();
   B(int);
   B operator+(const B&) const;
};

int main()
{
   A a1, a2, a3;
   a1 = a2 + a3;            // okay -> a1 = operator(a2, a3)
   a1 = 1 + a3;             // okay -> a1 = operator(A(1),a3)
   a1 = a2 + 2;             // okay -> a1 = operator(a2, A(2))
   a1 = 3 + 4;              // okay -> a1 = A(3+4)

   B b1, b2, b3;
   b1 = b2 + b3;            // okay -> b1 = b2.operator(b3)
   b1 = 1 + b3;             // Compiler-Fehler -> b1 = B(1).operator(b3) nicht machbar
   b1 = b2 + 2;             // okay -> b1 = b2.operator(B(2))
   b1 = 3 + 4;              // okay -> b1 = B(3+4)
}
```

Empfehlung – benutzen Sie, wenn möglich, eine Element-Operator-Funktion. Bei Klassen mit impliziter Typumwandlung und Operatoren, die symmetrisch aufgerufen können werden sollen, muss es dagegen eine freie Operator-Funktion sein.

16.4 Ausgabe

16.4.1 Teil 1

Bislang mussten wir folgendes schreiben:

```
date d;
d.print();
```

Schöner wäre aber die normale Ausgabe mit:

```
date d;
std::cout << d;
```

Lösung – ein entsprechender Operator muss definiert werden.

16.4.2 Einschub über Streams

Um einen Ausgabe-Operator zu definieren, sollten wir uns mit dem jetzigen Wissen die normale Ausgabe noch einmal anschauen, und überlegen, was wohl dahinter steckt.

Frage – was verbirgt sich wohl hinter dieser Anweisung?

```
std::cout << 8;
```

Analyse – bei „std::cout“ wird es sich um ein Objekt einer Klasse handeln - „std::ostream“. Entweder ist für die Typen „std::ostream“ und „int“ eine freie Operator-Funktion „<<“ definiert, oder „std::ostream“ enthält eine entsprechende Element-Operator-Funktion.

© Detlef Wilkening 2022

Hypothese – das Symbol „std::cout“ ist wahrscheinlich ein globales Objekt der Klasse „std::ostream“. In dieser Klasse ist wahrscheinlich u.a. eine Element-Operator-Funktionen für den Ausgabe-Operator „<<“ mit dem elementaren Daten-Typ „int“ definiert.

Test der Hypothese – wenn das stimmt, müsste sich der Ausgabe-Operator auch in Funktions-Schreibweise aufrufen lassen:

```
std::cout << 8;
std::cout.operator<<(8);
```

Und als komplettes Programm:

```
#include <iostream>
using namespace std;

int main()
{
   cout << 8;
   cout.operator<<(8);
}
```

```
Ausgabe
88
```

Klappt! Gut, und weiter.

Frage – und wie kommt die Verkettung der Ausgabe-Operatoren zustande?

```
std::cout << 8 << 'x';
```

Analyse – der Ausgabe-Operator „<<“ wird von links nach rechts ausgewertet, d.h. zuerst der Teil „std::cout << 8“, der dem Funktions-Aufruf „std::cout.operator<<(8)“ entspricht. Damit auch der zweite Funktions-Aufruf „std::cout.operator<<('x')“ funktionieren kann, muss die erste Operator-Funktion den veränderten Stream zurückgegeben haben. Außerdem muss es auch noch eine Element-Operator-Funktion für den Ausgabe-Operator „<<“ mit dem elementaren Daten-Typ „char“ geben.

Hypothese – die Definition der Klasse „std::ostream“ sieht ungefähr so aus:

```
// Wahrscheinlich ungefaehre Definition von std::ostream

// Irgendwie wird der Namespace std erzeugt - noch unbekannt

class ostream
{
public:
   ostream& operator<<(char);
   ostream& operator<<(signed char);
   ostream& operator<<(unsigned char);
   ostream& operator<<(short);
   ostream& operator<<(unsigned short);
   ostream& operator<<(int);
   ostream& operator<<(unsigned int);
   // usw...

   // und vieles mehr
};
```

Bemerkung – in Wirklichkeit sieht das ganze leider viel komplizierter aus, da „std::ostream“ nur ein typedef auf die Template-Klasse „std::basic_ostream<charT, traits>“ ist, die u.a. auch

noch eine Basis-Klasse hat.

Aber im Prinzip konnten wir analysieren und verstehen, was sich hinter der so lange *“blind“* benutzten Ausgabe verbirgt. Damit haben wir auch das nötige Verständnis, um einen eigenen Ausgabe-Operator für unsere Klassen zu schreiben.

16.4.3 Teil 2

Genug des Vorgeplänkels – definieren wir den Ausgabe-Operator für die Klasse „date“.

Da

- der erste Parameter des Ausgabe-Operators „<<“ der Stream ist, und
- die Klasse „std::ostream“ – als Klasse der C++ Standard-Bibliothek – von uns nicht erweitert werden kann,
- => muss der Ausgabe-Operator als freie Operator-Funktion implementiert werden.
- Und damit der Ausgabe-Operator Zugriff auf die Attribute der Klasse „date“ hat, wird er als Friend-Funktion deklariert.

```
#include <ctime>
#include <iomanip>
#include <iostream>
using namespace std;

class date
{
public:
   date();
   date(int d, int m, int y);

   friend ostream& operator<<(ostream&, const date&);    // Deklaration Ausgabe-Operator

private:
   int day_;
   int month_;
   int year_;
};

date::date()
{
   time_t timer = time(0);
   tm* tblock = localtime(&timer);
   day_ = tblock->tm_mday;
   month_ = tblock->tm_mon+1;
   year_ = tblock->tm_year+1900;
}

date::date(int d, int m, int y)
 : day_(d), month_(m), year_(y)
{
}

ostream& operator<<(ostream& out, const date& d)         // Ausgabe-Operator
{
   char c = out.fill('0');
   out << setw(2) << d.day_ << '.' << setw(2) << d.month_ << '.' << setw(4) << d.year_;
   out.fill(c);
   return out;
}

int main()
{
   date d1;
   cout << "Es ist der " << d1 << '\n';

   date d2(31, 12, 2000);
```

```
   cout << "Das letzte Jahrtausend endete am " << d2 << '\n';
}
```

```
Mögliche Ausgabe (da das aktuelle Datum vorkommt)
Es ist der 13.01.2013
Das letzte Jahrtausend endete am 31.12.2000
```

Bemerkung – in der Praxis passiert es häufig, dass der Ausgabe-Operator private Attribute benötigt, für die es keine Getter-Funktionen gibt. In diesem Fall wird er häufig als „friend“ der Klasse implementiert werden (wie hier im Beispiel), oder er delegiert die eigentliche Ausgabe an eine Element-Funktion der Klasse.

Hinweis – selbst wenn Sie noch nicht verstehen warum, dieser Ausgabe-Operator funktioniert auch mit Ausgabe-File-Streams oder String-Streams. Der Grund dahinter ist die „ist-ein“ Beziehungs-Semantik von öffentlicher Vererbung – siehe später.

16.5 Kopier-Zuweisungs-Operator =

Der Kopier-Zuweisungs-Operator „=“ ist der Zuweisungs-Operator „=“, der ein Objekt der Klasse selber erwartet. Er hat eine ähnliche Sonderstellung wie der Kopier-Konstruktor.

Im Normallfall sind in C++ Objekte einander zuweisbar. Vom Compiler wird daher, wenn Sie selber keinen Kopier-Zuweisung-Operator deklarieren, automatisch ein impliziter „public“ Kopier-Zuweisungs-Operator erzeugt, der ein const-Referenz-Objekt der Klasse erwartet, für jedes Attribut der Klasse wiederum den Zuweisungs-Operator aufruft, und eine Referenz auf das aktuelle Objekt zurückgibt.

Reicht der automatische Kopier-Zuweisungs-Operator nicht aus, so müssen wir selber einen definieren. Typischerweise ist er „public“, erwartet das zugewiesene Objekt per Const-Referenz und gibt das aktuelle Objekt als Referenz zurück – verhält sich also vergleichbar zum automatischen Kopier-Zuweisungs-Operator.

```
class A
{
public:
   A& operator=(const A&);
};

A& A::operator=(const A&)
{
   cout << "Kopier-Zuweisungs-Operator\n";
   return *this;
}

int main()
{
   A a1, a2;
   a1 = a2;             // Ausgabe: Kopier-Zuweisungs-Operator
}
```

Der implizite Kopier-Zuweisungs-Operator ist unabhängig von allen anderen Zuweisungs-Operatoren, die Sie deklarieren bzw. definieren.

```
class A
```

```
{
public:
   A& operator=(int) { return *this; }
};

int main()
{
   A a1, a2;
   a1 = 1;         // okay - unser eigener Operator=(int)
   a1 = a2;        // okay - impliziter Kopier-Zuweisungs-Operator
}
```

Achtung – der Zuweisungs-Operator muss immer als Element-Operator-Funktion definiert sein. Er darf nicht als freie Operator-Funktion implementiert werden.

Hinweis – bei unseren bisherigen Beispielen reichte der implizite Kopier-Zuweisungs-Operator aus – genauso wie der implizite Kopier-Konstruktor und der implizite Destruktor. In späteren Kapiteln finden sich Beispiele und Hinweise für Klassen, bei denen der implizite Kopier-Zuweisungs-Operator nicht ausreicht. Kann kein sinnvoller Kopier-Zuweisungs-Operator implementiert werden, kann es sinnvoll sein, diesen zu verbieten – siehe Kapitel 16.5.1.

Hinweis – wenn Sie einen eigenen Kopier-Konstruktor schreiben müssen, müssen Sie eigentlich auch immer einen eigenen Kopier-Zuweisungs-Operator und einen eigenen Destruktor schreiben. Dies wird auch gerne „rule-of-three“ („Regel der Drei“) genannt.

Achtung – machen Sie sich bitte den Unterschied zwischen einem Kopier-Konstruktor und einem Kopier-Zuweisungs-Operator klar. Obwohl beide sehr ähnlich wirken, sind ihre Funktionen doch recht verschieden:

- Ein Kopier-Konstruktor **erzeugt** ein **neues** Objekt aus einem bestehenden.
- Ein Kopier-Zuweisungs-Operator **weist** ein bestehendes Objekt einem **bestehenden** zu.

Hinweis – um dem Anwender die Benutzung Ihrer Klassen zu vereinfachen, bzw. die Benutzung intuitiv zu gestalten, sollten Sie – wenn möglich – die normale, aus C bzw. C++ bekannte Operatorsemantik erhalten. Daher sollten z.B. die Zuweisungs-Operatoren wie „=“ oder „+=“ das aktuelle Objekt als Referenz zurückgeben.

16.5.1 Kopier-Zuweisungs-Operator verbieten

Wie verbietet man den Kopier-Zuweisungs-Operator? Oder genauer: wie verbietet man das Zuweisen von Typen? Dies läuft vollkommen analog zum Verbieten des Kopier-Konstruktors ab. Das Beispiel zeigt das Verbieten von Kopier-Konstruktor und Kopier-Zuweisungs-Operator. Dazu deklariert man einfach die entsprechenden Elemente mit „= delete“.

```
class A
{
public:
   A();
   A(const A&) = delete;               // Kein Kopier-Konstruktor
   A& operator=(const A&) = delete;    // Kein Kopier-Zuweisungs-Operator
};
```

```
int main()
{
   A a1, a2;
   A a3(a1);         // Compiler-Fehler, da A nicht kopierbar
   a1 = a2;          // Compiler-Fehler, da A nicht zuweisbar
}
```

16.6 Move-Zuweisungs-Operator =

So wie der Kopier-Konstruktor der Bruder des Kopier-Zuweisungs-Operators ist, so gibt es in C++11 natürlich auch einen Bruder zum Move-Konstruktor – den Move-Zuweisungs-Operator. Wie der Move-Konstruktor erwartet er eine Non-Const-Referenz auf ein Objekt der Klasse und sollte die Move-Semantik für die Zuweisung implementieren.

```
class A
{
public:
   A(A&&);                // Deklaration Move-Konstruktor
   A& operator=(A&&);     // Deklaration Move-Zuweisungs Operator
};

A& A::operator=(A&&)      // Definition Move-Zuweisungs Operator
{
   // Wie auch immer eine sinnvolle Implementierung aussieht...
}
```

Auch der Move-Zuweisungs-Operator wird vom Compiler automatisch erzeugt, wenn:

- kein benutzer-deklarierter Kopier-Konstruktor,
- kein benutzer-deklarierter Kopier-Zuweisungs-Operator,
- kein benutzer-deklarierter Move-Konstruktor,
- kein benutzer-deklarierter Move-Zuweisungs-Operator, und
- kein benutzer-deklarierter Destruktor

vorliegt.

Detaillierter will ich hier nicht auf den Move-Zuweisungs-Operator und die Move-Semantik eingehen.

16.7 Funktions-Aufruf Operator

Ein ganz spezieller Operator ist in C++ der Funktions-Aufruf-Operator „()“ – die runden Klammern. Mit ihnen kann man einem Objekt Funktions-Charakter geben, d.h. Objekte wie Funktionen nutzen.

```
#include <iostream>
using namespace std;

class A
{
public:
   void operator()() const;
};

void A::operator()() const
{
   cout << "Funktions-Aufruf-Operator ()\n";
}
```

```
int main()
{
   A a;
   a();          // Keine Funktion - Aufruf des Operators "()" fuer das Objekt "a"
}
```

Ausgabe
```
Funktions-Aufruf-Operator ()
```

Der Funktions-Aufruf-Operator ist der einzige Operator in C++, bei dem der Operator die Anzahl an Parametern nicht festlegt und auch Default-Argumente erlaubt sind – d.h. der Funktions-Aufruf-Operator darf mit beliebigen Parameter-Anzahlen definiert werden und es dürfen dabei Default-Argumente benutzt werden.

```
#include <iostream>
using namespace std;

class A
{
public:
   void operator()() const;
   void operator()(int) const;
   void operator()(bool) const;
   void operator()(double, int=5) const;
};

void A::operator()() const
{
   cout << "Funktions-Aufruf-Operator ()\n";
}

void A::operator()(int n) const
{
   cout << "Funktions-Aufruf-Operator (int " << n << ")\n";
}

void A::operator()(bool b) const
{
   cout << "Funktions-Aufruf-Operator (bool " << b << ")\n";
}

void A::operator()(double d, int n) const
{
   cout << "Funktions-Aufruf-Operator (double " << d << ", int " << n << ")\n";
}

int main()
{
   A a;
   a();              // Aufruf des Operators "()"
   a(4);             // Aufruf des Operators "(int)" mit "4"
   a(true);          // Aufruf des Operators "(bool)" mit "true"
   a(2.72);          // Aufruf des Operators "(double, int)" mit "2.72, 5"
   a(3.14, 6);       // Aufruf des Operators "(double, int)" mit "3.14, 6"
}
```

Ausgabe
```
Funktions-Aufruf-Operator ()
Funktions-Aufruf-Operator (int 4)
Funktions-Aufruf-Operator (bool 1)
Funktions-Aufruf-Operator (double 2.72, int 5)
Funktions-Aufruf-Operator (double 3.14, int 6)
```

Hinweis – der Funktions-Aufruf Operator muss immer als Element-Funktion implementiert werden. Als freie Funktion kann er nicht implementiert werden.

Wahrscheinlich fragen Sie sich: Was soll das? Wozu implementiert man einen Funktions-Aufruf Operator? Das ist doch sinnlos! Nein – ist es nicht. Funktions-Objekte sind viel mächtiger als Funktionen, da sie z.B. Attribute enthalten können und damit einen Status haben, der einen einzelnen Funktions-Aufruf *überlebt* – ohne dass man gleich eine *böse* globale Variable benötigt. Außerdem kann der Compiler Funktions-Objekte radikaler als Funktionen optimieren, so daß Funktions-Objekte oft performanter sind. Funktions-Objekte, daher das Implementieren des Funktions-Aufruf Operators, sind ein wichtiges Idiom in C++. Zum Beispiel im Zusammenhang mit STL Algorithmen werden sie sehr häufig verwendet.

Hinweis – Funktions-Objekte werden oft auch „Funktoren“ genannt, und als Oberbegriff für Funktionen und Funktions-Objekte findet man auch oft den Begriff „Callables“.

16.8 Spezialitäten

Es lassen sich fast alle C++ Operatoren überladen – folgende sind die Ausnahme:

- Komponentenzugriff „.“
- Bereichszuordnung „::“
- „sizeof“
- Komponente über Komponentenzeiger „.*“
- Bedingter Ausdruck „? :“
- Cast-Operatoren: „static_cast“, „const_cast“, „reinterpret_cast“ und „dynamic_cast“

Das heißt, dass sich auch nicht so offensichtliche Operatoren überladen lassen, wie z.B.

- Pfeil-Operator „->“
- Komma-Operator „,“
- Index-Operator „[ ]“

Hier noch ein paar Bemerkungen zu einigen speziellen Operatoren:

- Für die Inkrement und Dekrement Operatoren „++“ und „—“ existieren jeweils 2 Varianten für die Überladung, um die Prä- und die Postfix Notation zu unterscheiden.
- Auch der Index Operator „[ ]“ läßt sich überladen – damit kann eine Klasse quasi Array-Charakter bekommen – siehe z.B. „std::vector“ oder „std::string“.
- Für die dynamische Speicherverwaltung stellt C++ die Operatoren „new“ und „delete“ in verschiedenen Ausführungen zur Verfügung. Diese Operatoren können sowohl global als auch klassenspezifisch überladen werden.
- Neben Konvertierungs-Konstruktoren gibt es in C++ auch spezielle Konvertierungs-Operatoren, die der Compiler für die implizite Typumwandlung nutzen kann (vergleichbar zu den Konvertierungs-Konstruktoren).

© Detlef Wilkening 2022

16.9 Fazit

16.9.1 Grenzen

Die Operator-Überladung hat Grenzen:

- Die Priorität, Syntax, Parameteranzahl und Auswertungsreihenfolge liegen fest.
- Die Operatoren für nicht-benutzerdefinierte Datentypen können nicht verändert werden.
- Es können keine neue Operatoren definiert werden, d.h. z.B. Operator „**“ zum Potenzieren ist **nicht** möglich.
- Operatoren dürfen keine Klassen-Funktionen („static“) sein.
- Der Compiler macht keine eigenständigen Übertragungen. Aus „+“ und „=“ wird **nicht** automatisch „+=“, bzw. die Postfix Semantik bei Inkrement und Dekrement Operatoren wird nicht automatisch erzeugt.
- Einige Operatoren können nicht überladen werden – s.o.
- Fast alle Operatoren lassen sich als Element-Funktionen definieren (Ausnahme „new“ und „delete“) – aber nicht alle als freie Funktion:
  - Alle Zuweisungen wie z.B. =, *=, +=, …
  - Konvertierungen
  - ( )
  - [ ]
  - ->
  - new/delete
- Bei Operator-Funktionen sind ausser beim Funktions-Aufruf-Operator keine Default-Argumente zugelassen.

Da die Semantik der Operatoren *vorbelegt* ist, die Priorität und Auswertungsreihenfolge feststehen, kann eine **willkürliche Verwendung** die Lesbarkeit eines Programmes stark einschränken. Setzen Sie Operatoren daher mit Verstand ein.

16.9.2 Anmerkungen

Im Prinzip sind Operatorfunktionen nichts wirklich Neues – sie ergänzen die Sprache um keine echte neue Funktionalität. Trotzdem sind sie in C++ sehr wichtig. Denn sie erlauben die Benutzung der bekannten Operatoren für eigene Typen, was in vielen Fällen die Les- und Benutzbarkeit stark erhöht – z.B. bei mathematischen Klassen, aber auch der Element-Zugriff bei Vektoren. Außerdem können mit Operatoren Klassen-Implementierungen umgesetzt werden, die sich wie eingebaute Daten-Typen verhalten.

© Detlef Wilkening 2022