Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 2

2. Ergänzungen zum Überladen von Operatoren

2.1. Increment- und Decrement-Operator

2.2. Funktionsaufruf-Operator

2.3. Delegations-Operator (->)

2.4. Typkonvertierung

Überladen des Increment- und Decrement-Operators in C++

- **Notationsformen des Increment- bzw Decrement-Operators**

  ◇ Die **unären Operatoren** ++ (**Increment**) und -- (**Decrement**) existieren für die Standard-Datentypen jeweils in **2 Formen** :

  ▻ **Prefix-Notation** : **++x**; Ausdruckswert : veränderter Operandenwert

  ▻ **Postfix-Notation**: **x++**; Ausdruckswert : alter Operandenwert

  ◇ In **früheren** C++-Versionen konnte beim Überladen dieser Operatoren **nicht** zwischen **Prefix**- und **Postfix**-Notation **unterschieden** werden.

  ◇ Im **ANSI/ISO-C++-**Standard ist jedoch eine **Unterscheidungsmöglichkeit** vorgesehen :

  ▻ Die **Prefix-Notation** wird - wie bei den anderen unären Operatoren - durch eine **Operatorfunktion** mit **einem Parameter** (der bei der Definition als Member-Funktion implizit als `this`-Pointer übergeben wird) realisiert.

  ▻ Die **Postfix-Notation** dagegen wird durch eine **Operatorfunktion** mit **zwei Parametern** (von denen bei der Definition als Member-Funktion der erste implizit als `this`-Pointer übergeben wird) realisiert.
  Der **zweite** Parameter ist ein **Dummy-Parameter**. Er muß vom Typ **int** sein.
  Beim Aufruf der Operatorfunktion wird für ihn i.a. der Wert 0 übergeben.

  ◇ Damit sich die überladenen Operatoren bezüglich Prefix- und Postfix-Notation wie die Standard-Operatoren verhalten – was sinnvollerweise der Fall sein sollte – , sollten sich die **beiden** jeweiligen Funktionen lediglich im **Rückgabewert** unterscheiden.
  Dabei ist zu berücksichtigen, daß die Standard-Increment-und Decrement-Operatoren in der **Prefix-**Notation einen ***Lvalue*** (→ Rückgabe einer Referenz) und in der **Postfix-**Notation einen ***Rvalue*** (→ Rückgabe eines Wertes) zurückgeben.

- **Beispiel** :

```
class Ratio
{ public:
    //  ...
    Ratio& operator++(void);      // Überladen Increment-Operator (Prefix)
    Ratio operator++(int);        // Überladen Increment-Operator (Postfix)
  //  ...
};

Ratio& Ratio::operator++(void)        // Prefix
{ zaehler+=nenner;
  return *this;                       // Rückgabe neuer Wert (als Referenz)
}

Ratio Ratio::operator++(int dummy)    // Postfix, dummy ist Dummy-Parameter
{ Ratio temp=*this;
  zaehler+=nenner;
  return temp;                        // Rückgabe alter Wert
}

int main(void)
{
  Ratio a(4,5), b, c;

  b=a++;       // a.operator++(0);    ==>   b ⇐ (4,5),   a ⇐ (9,5)
  c=++a;       // a.operator++();     ==>   c ⇐ (14,5),  a ⇐ (14,5)
  //  ...
}
```

- **Anmerkung :**

  Die **Prefix-Notation** realisiert i.a. eine **schnellere Operation** als die Postfix-Notation (zweimaliges Kopieren des Objekts) → sie sollte – wenn nur der Inc- bzw. Dec-Effekt benötigt wird – in der Regel **bevorzugt verwendet** werden.

Funktionsaufruf-Operator in C++ (1)

- **Funktionsaufruf**
  - ◇ Ein **Funktionsaufruf** hat die Form
    `expression(expression-list)`
  - ◇ Er kann formal als **zweistellige** (binäre) Operation aufgefasst werden :
    - **`()`** ist der – binäre – **Funktionsaufruf-Operator**
    - **`expression`** ist der **linke Operand**.
      Er muß eine Funktion referieren oder einen Funktionspointer als Wert ergeben.
    - **`expression-list`** ist der **rechte Operand**.
      Er besteht aus der durch Kommata separierten Liste der aktuellen Parameter, die auch leer sein kann.

- **Überladen des Funktionsaufruf-Operators `()`**
  - ◇ Auch der Funktionsaufruf-Operator lässt sich für Objekte selbstdefinierter Klassen überladen.
    → In diesem Fall ist der linke Operand keine Funktion (bzw Funktionspointer) sondern ein Objekt
  - ◇ Die Operatorfunktion **`operator()`** muß eine **nichtstatische Memberfunktion** sein, die beliebig viele Parameter (auch gar keine) haben kann.
    Default-Werte für die Parameter sind **zulässig**.
  - ◇ **Beispiel** :

```
class Gerade           // y= m*x + t
{
  public :
    Gerade(double, double=0);
    double operator()(double=0);
  private :
    double m_dM;       // Steigung m
    double m_dT;       // Achsenabschnitt t
};

// ---------------------------------------------------------------------------

Gerade::Gerade(double m, double t)
{ m_dM=m;
  m_dT=t;
}

double Gerade::operator()(double x)
{ return x*m_dM + m_dT;
}

// ---------------------------------------------------------------------------

int main(void)
{ Gerade line(0.5, 2);
  double arg;
  cout << endl << "Achsenabschnitt : " << line() << endl ;
  while (cout << "? ", cin >> arg)
    cout << "Wert : " << line(arg) << endl;
  cout << endl;
  return 0;
}
```

  - ◇ Die Operatorfunktion **`operator()`** kann für eine Klasse auch **mehrfach überladen** werden (unterschiedliche Parameterlisten !)

Funktionsaufruf-Operator in C++ (2)

- **Funktionsobjekte**
  - ◇ Objekte von Klassen, für die der Funktionsaufrufoperator überladen ist, bezeichnet man als **Funktionsobjekte**.
  - ◇ Es sind **Objekte**, die **wie Funktionen verwendet** werden können :
    Der Ausdruck

    `object(expression-list)`

    ist kein Aufruf der Funktion `object()`,
    sondern ein Aufruf der Memberfunktion `operator()()` für das Objekt `object.`:

    `object.operator()(expression_list)`

- **Anwendung von Funktionsobjekten**
  - ◇ Funktionsobjekte stellen häufig eine sinnvolle **objektorientierte Alternative zu freien Funktionen** dar.
    Insbesondere dann, wenn mehrere gleichartig strukturierte aber mit unterschiedlichen Kenngrößen (z.B. Koeffizienten von Polynomen) arbeitende Funktionen benötigt werden.
    Statt diese Kenngrößen bei jedem Funktionsaufruf zusätzlich zu den eigentlichen Funktionsparametern zu übergeben, werden sie durch einen Konstruktor in Datenkomponenten von Funktionsobjekten abgelegt.
  - ◇ Das Überladen der Operatorfunktion `operator()()` ist weiterhin auch immer dann sinnvoll, wenn für eine Klasse nur **eine einzige** oder eine wichtige **überwiegend angewendete Funktionalität** existiert.
    Beispiele : - Implementierung von einfachen **Iteratoren** (→ Programmbeispiel s. Kap. 8.5 "Iteratoren")
    - **Applikatorklassen** als eine Möglichkeit zur Realisierung von Manipulatoren mit Parametern
  - ◇ Beispiele anderer gebräuchlicher Anwendungen des überladenen Funktionsaufruf-Operators sind :
    - Einsatz als **Substring-Operator**
    - **Indizierung mehrdimensionaler Arrays** bzw. von Datenstrukturen, die als mehrdimensionale Arrays behandelt werden können
  - ◇ **Beispiel** zur Indizierung eines mehrdimensionalen Arrays

```
class IntMatrix
{ public :
    IntMatrix(unsigned, unsigned);
    int& operator()(unsigned, unsigned);
  private :
    unsigned m_uRows;
    unsigned m_uCols;
    int* m_piMatr;
};


IntMatrix::IntMatrix(unsigned rows, unsigned cols)
{ m_uRows=rows;
  m_uCols=cols;
  m_piMatr=new int[m_uRows*m_uCols];
  // Initialisierung aller Komponenten mit 0
}

int& IntMatrix::operator()(unsigned row, unsigned col)
{ row=row%m_uRows;
  col=col%m_uCols;
  return m_piMatr[row*m_uCols+col];
}


int main(void)
{
  IntMatrix matr(5, 10);
  matr(2,3)=25;
  // ...
}
```

Operator -> in C++ (1)

• Interpretation des Operators ->

◇ **Standardmäßig** wird der Operator **->** als **binärer Operator** verwendet.
Er dient zum Zugriff zu Komponenten eines Objekts.
Dabei muß der **linke Operand** ein **Pointer auf ein Objekt** und der **rechte Operand** der **Name einer Komponente** dieses Objekts sein.
Der "**Wert**" eines derartigen Ausdrucks ist die dadurch **referierte Objekt-Komponente**.
→ **dereferenzierender Komponenten-Operator**.

◇ Der Operator lässt sich für Klassen so **überladen**, dass er direkt auf Objekte (statt auf Objekt-Pointer) angewendet werden kann. :
In dieser überladenen Form stellt er einen **unären Operator** dar, der bei einem Objekt als Operanden – direkt oder indirekt – einen Objekt-Pointer zurückliefern muß.

→ Ein Ausdruck **`object->name`**
wird interpretiert als **`(object.operator->())->name`**

◇ Der zurückgelieferte Pointer muß nicht das Objekt referieren, auf das der Operator angewendet wurde.
Vielmehr wird er sich bei realen Anwendungen im allgemeinen auf ein anderes Objekt beziehen.
Die durch `name` bezeichnete Objektkomponente muß dann eine Komponente dieses Objekts und nicht eine des Operanden-Objekts sein.
→ Der Komponentenzugriff wird an ein anderes Objekt **delegiert**.
→ **Delegations-Operator**.

• Operatoratorfunktion `operator->`

◇ Eine den Operator **->** überladende Funktion muß eine **nichtstatische Memberfunktion ohne Parameter** sein.

◇ Sie sollte einen **Pointer auf ein Objekt** als **Funktionswert** zurückgeben.

◇ Falls die Funktion statt eines Objekt-Pointers ein Objekt (oder eine Referenz auf ein Objekt) zurückgibt, wird die **Auswertung** eines **-> - Ausdrucks rekursiv** fortgesetzt : Für das zurückgelieferte Objekt wird wiederum eine Operatorfunktion `->` aufgerufen. Falls für dessen Klasse keine definiert ist, endet die Auswertung fehlerhaft.

◇ Ein expliziter Aufruf der Operatorfunktion `operator->` muß ohne Parameter erfolgen (das entspricht **einem Operanden**). Eine Verwendung von `->` in einem Ausdruck dagegen erfordert **zwei Operanden**.

```
class Abc
{ public :
    // ...
    X* operator->();          // X sei eine andere Klasse
  private :
    // ...
};

void func(Abc p)
{
  X* px1 = p.operator->();    // ok
  X* px2 = p->;               // Fehler !
  // ...
}
```

• Beispiele für Anwendungen

▻ Realisierung von **"*smart*" Pointern**.
Hierbei handelt es sich um Objekte, die wie Pointer verwendet werden können, bei jedem Komponentenzugriff aber eine zusätzliche Funktionalität realisieren. (Kapselung der eigentlichen Pointer in dem Objekt)

▻ **Ausdehnung der Fähigkeiten** einer vorhandenen Klasse auf eine andere Klasse, in die sie eingekapselt wird, ohne Vererbung sondern mittels **Delegation**.

Operator -> in C++ (2)

- **Beispiel zur Delegation mittels Operatorfunktion `operator->`**

```
class Worker
{ public :
    Worker(int=0);
    // ...
    void doWork();
  private :
    // ...
    int m_id;
};


#include <iostream>
using namespace std;


inline Worker::Worker(int id)
{
  m_id=id;
}


inline void Worker::doWork()
{
  cout << "\nIch, der Worker mit id=" << m_id << ", arbeite gerade !\n";
}

// ---------------------------------------------------------------------------

class Chief
{ public :
    // ...
    void hireWorker(Worker&);
    Worker* operator->();
  private :
    Worker* myEmpl;
};


inline void Chief::hireWorker(Worker& fred)
{
  myEmpl=&fred;
}


inline Worker* Chief::operator ->()
{
  return myEmpl;
}

// ---------------------------------------------------------------------------

int main(void)
{ Chief moi;
  Worker tu(1);
  moi.hireWorker(tu);
  moi->doWork();          // Delegation von doWork() an Worker-Objekt
  return 0;
}
```

Ausgabe des Programms :

```
Ich, der Worker mit id=1, arbeite gerade !
```

Konvertierungsfunktionen in C++

- **Funktionen zur Typkonvertierung :**
  - ◇ **Konstruktoren**, die mit **einem Parameter** aufgerufen werden können, können als **Funktionen zur Konvertierung** des Parameter-Typs **in den Klassen-Typ** betrachtet werden.
  - ◇ Zur **entgegengesetzten Konvertierung** - **aus dem Klassentyp** in einen anderen Typ - dienen **Konvertierungs-funktionen** (*conversion functions*).
  - ◇ Im Prinzip handelt es sich bei **einer Konvertierungs-Funktion** um eine **Operatorfunktion** zum **Überladen beider Formen der** einfachen **Typkonvertierungs-Operatoren** (Cast-Operator `(typ)` und Werterzeugungs-Operator `typ()`) sowie des Operators **`static_cast`**.

- **Eigenschaften einer Typkonvertierungsfunktion :**
  - ◇ Sie muß eine **nichtstatische Member-Funktion** sein.
  - ◇ Da die Typkonvertierungs-Operatoren **unäre Operatoren** sind, hat sie **keine expliziten Parameter**.
  - ◇ Der Typ ihres Rückgabewertes ist implizit durch den Ziel-Typ, der nach dem Schlüsselwort `operator` anzugeben ist, festgelegt. Eine **explizite Festlegung des Funktionstyps** erfolgt daher **nicht**.

    → **Allgemeine Syntax** für die **Funktionsdeklaration :**

    ```
    operator typ();        // typ ist der Ziel-Typ
    ```
  - ◇ Der Ziel-Typ **`typ`** kann ein **beliebiger Typ**, auch ein anderer Klassentyp, sein.
  - ◇ Eine Konvertierungsfunktion wird **nicht nur** bei **expliziter Anwendung** eines **Typkonvertierungs-Operators** sondern auch in Fällen **impliziter Typkonvertierung** aufgerufen.

- **Beispiel :**

```
class Ratio
{ public:
    // ...
    operator double(void);     // Konvertierungsfunktion Ratio --> double
    // ...
};

// ---------------------------------------------------------------------------

Ratio::operator double(void)
{
  return (double)zaehler/nenner;
}

// ...

/ ---------------------------------------------------------------------------

int main(void)
{
  Ratio a(3,-2), b(4,9);
  double z, v, w;
  // ...
  z=(double)a;           // Cast-Operator    --> Aufruf von a.operator double()
  w=double(b);           // Werterzeug.-Op   --> Aufruf von b.operator double()
  u=static_cast<double>(b); // static_cast   --> Aufruf von b.operator double()
  v=a;                   // impliz.Typ-Konv. --> Aufruf von a.operator double()
  // ...
}
```

Anmerkungen zur impliziten Typkonvertierung in C++ (1)

- **Implizite Typkonvertierungen :**

  Stimmen bei einer **Zuweisung** oder **Initialisierung** die **Typen** des **Ziel-Objekts** und des **Quell-Wertes nicht überein**, versucht der Compiler eine **automatische** - **implizite** - **Typkonvertierung** des Quell-Wertes in den Typ des Ziel-Objekts vorzunehmen.

  Anmerkung : Diese Typkonvertierung findet auch bei **Parameterübergaben** in Funktionsaufrufen und in **Ausdrücken**, die ja in Aufrufe von Operatorfunktionen umgesetzt werden, statt
  (Die Übergabe von Wertparametern stellt eine Erzeugung und Initialisierung temporärer Objekte dar)

- **Regeln für die implizite Typkonvertierung**

  ◇ Für die implizite – automatische - Typkonvertierung werden die folgenden - vereinfacht dargestellten - **Regeln** in der **angegebenen Reihenfolge** angewendet (**absteigende Priorität**) :

  1. Anwendung **trivialer Typumwandlungen**
     (z.B. Array-Name → Pointer, Funktionsname → Pointer, `T` → `T&`, `T&` → `T`, `T` → `const T`, `T*` → `const T*`)
  2. Anwendung der **informationserhaltenden Standard-Typumwandlungen**
     - Integral Promotion :
       `char`, `short`, `enum`, Bitfeld → `int`,
       `unsigned char`, `unsigned short`, `unsigned` Bitfeld → `(unsigned) int`
     - Umwandlung `float` → `double`
  3. Anwendung der **übrigen Standard-Typumwandlungen**
     - Standard-Typumwandlungen von C
       (z.B. `int` → `double`, `int` → `unsigned` , usw)
     - Umwandlung von Referenzen und Pointer auf abgeleitete Klassen in Referenzen und Pointer auf Basisklassen
  4. Anwendung **benutzerdefinierter Typumwandlungen**
     - **Konstruktoren**
     - **Konvertierungsfunktionen**

  ◇ Eine Typkonvertierung kann **über mehrere Umwandlungsschritte** erfolgen.
  Existieren **mehrere Konvertierungswege**, wählt der Compiler den **kürzesten** aus.
  Existieren **zwei oder mehr gleichberechtigte Wege**, erzeugt er eine **Fehlermeldung** .

- **Implizite Anwendung benutzerdefinierter Typumwandlungen**

  Es gelten die folgenden Regeln :

  ◇ Sie werden **nur dann** versucht, wenn **keine** der **Standard-Typumwandlungen** zum **Erfolg** führt.

  ◇ An einer Typkonvertierung darf **maximal eine benutzerdefinierte Typumwandlung** beteiligt sein,
  Standard-Typumwandlungen können zusätzlich ohne Einschränkung angewendet werden.

  ◇ Es darf **nur einen Konvertierungsweg** geben, an dem **eine benutzerdefinierte Typumwandlung beteiligt** ist (unterschiedliche Weglängen spielen hierbei keine Rolle). → Konvertierung muß eindeutig sein.

  ◇ **Konstruktoren** und **Konvertierungsfunktionen** sind **gleichberechtigt**.

Anmerkungen zur impliziten Typkonvertierung in C++ (2)

- Die **automatische benutzerdefinierte Typkonvertierung** kann z.B. dazu genutzt werden, um **typgemischte Ausdrücke** unter Beteiligung von Objekten selbstdefinierter Klassen zu ermöglichen.

  Statt die jeweilige Operatorfunktion mehrfach - für die zulässigen Typkombinationen - zu überladen, benötigt man **nur** jeweils **eine Operatorfunktion** und **adäquate Konstruktoren**.
  Soll eine **"symmetrische" Typ-Mischung** möglich sein, so muß die Operatorfunktion auch hier eine **"freie"** – gegebenenfalls befreundete – **Funktion** sein.

  Beispiel :

```
class Ratio
{ public:
    // ...
    Ratio(double);                                  // alternativer Konstruktor
    // ...
    friend Ratio operator+(const Ratio&, const Ratio&);
  // ...                                                    // nicht überladen
};

// ...

int main(void)
{
  Ratio a(4,9), b, c;
  // ...
  b=3.7+a;                          // Aufruf von operator+(Ratio(3.7), a)
  c=a+4.3;                          // Aufruf von operator+(a, Ratio(4.3))
  // ...
}
```

- **Probleme**

  Funktionen zur **automatischen Typkonvertierung** können auch **Probleme** erzeugen.
  Wird im obigen Beispiel zusätzlich eine Konvertierungsfunktion

  `Ratio::operator double()`

  die ein `Ratio`-Objekt in einen `double`-Wert umwandelt, definiert, sind die **gemischten Additions-Ausdrücke nicht** mehr **eindeutig** :

Beispielsweise kann	**`3.7 + a`**	dann mittels Typkonvertierung
als	**`Ratio(3.7) + a`**	
und als	**`3.7 + double(a)`**	
interpretiert werden.		

  → **Beide Konvertierungen** verwenden eine **benutzerdefinierte Umwandlung**, was **nicht zulässig** ist.

  => Konstruktoren und Konvertierungsfunktionen sind nur sehr wohl durchdacht zu definieren.

  Insbesondere muß darauf geachtet werden, daß **keine zyklischen Typkonvertierungen** möglich werden.
  Die ist z.B. der Fall,
  - ▻ wenn - wie im obigen Beispiel - **in einer Klasse** ein **Konstruktor** und eine **Konvertierungsfunktion** für genau **entgegengesetzte Typumwandlungen** definiert werden,
  - ▻ oder in **zwei Klassen** jeweils ein **Konstruktor** vorhanden ist, der eine **Typumwandlung aus der jeweiligen anderen Klasse** vornimmt.

Ergänzungen zur Typkonvertierung in C++

• Weiteres Beispiel zur Mehrdeutigkeit bei automatischer Typkonvertierung

Die **Konvertierung** eines Objekts der **Klasse X** in ein Objekt der **Klasse T** kann erreicht werden

- ▻ entweder durch die **Konvertierungsfunktion** : `X::operator T()` `// Member von X`
- ▻ oder durch die **Konstruktor-Funktion** : `T::T(X)` `// Member von T`

Wenn **beide Funktionen definiert** sind, sind **automatische Typkonvertierungen** ebenfalls **nicht mehr eindeutig** → Compiler-Fehler

Beispiel :

```
class X
{ // ...
  operator T();            // Konvertierungsfunktion
  // ...
};

class T
{ // ...
  T(X);                    // Konstruktor
  T(const T&);             // Copy-Konstruktor
  // ...
};

void main(void)
{
  X xobj;
  T tobj=xobj;            // Konstruktor oder Konvertierungsfunktion?
   // ...
}
```

• Vermeidung der Problematik von Mehrdeutigkeiten

Wegen der **Gefahr von Mehrdeutigkeiten** ist es häufig **besser**, durch **Vermeiden von Konvertierungsfunktionen** automatische Typkonvertierungen zu reduzieren.
Trotzdem **benötigte Typkonvertierungen** lassen sich i.a. **problemlos** durch **explizite Aufrufe** entsprechend definierter **normaler Member-Funktionen** realisieren.
Diesen Funktionen sollte man dann entsprechend **aussagekräftige Namen** geben.

Beispiel :

```
class Ratio
{ public:
    // ...
    double asDouble(void);     // Funktion zur expliziten Typkonvertierung
  // ...
 }

// ...

void main(void)
{ Ratio a(5,13);
  double z = a.asDouble();     // expliziter Aufruf der Typkonvertierung
  // ...
}
```