

Operatoren überladen

Grundsätzliches

Die erste, allgemeinste und wichtigste Regel für das Überladen von Operatoren lautet: *Tu's nicht*.

Das mag paradox klingen, aber der Grund dafür ist, daß sich die hinter der Anwendung eines Operators verborgene Semantik nur dann erschließt, wenn man mit der Anwendung des Operators an dieser Stelle vertraut ist. Das ist i.A. nur in seltenen Fällen gegeben. Wann immer die Bedeutung eines Operators nicht offensichtlich ist (Was bedeutet es wohl, wenn der Modulo-Operator auf zwei Objekte vom Typ `my_array` angewendet wird?¹), sollte auf das Überladen verzichtet werden.

Diese Operatoren lassen sich in C++ überladen:

- Arithmetik: `% + - * /` (sowie `% = += -= *= /=`) und `++ --`
- Bitmanipulationen: `& | ^ ~ << >>` (sowie `&= |= ^= ~= <<= >>=`)
- boolesche Algebra: `== != < > <= >= | | && !`
- andere Operatoren: `=` (Zuweisung) `[]` (Feldzugriff/Subskription), `*` (Dereferenzierung) `&` (Adressoperator), `->` (Elementzugriff), `()` (Funktionsaufruf)

Diese Operatoren lassen sich nicht überladen:

- `.` und `::` (beide für Elementzugriff)
- `?:` (bedingter Ausdruck)

Folgende Operatoren müssen als *Elementfunktionen* überladen werden, weil die Syntax von C++ dies verlangt: `=` (Zuweisung), `*` (Dereferenzierung), `&` (Adressoperator), `[]` (Subskription) und `()` (Funktionsaufruf).

Andere Operatoren müssen als *freie Funktionen* überladen werden, weil z.B. der linke Operand nachträglich nicht verändert werden kann. Dazu gehören insbesondere die Ein- und Ausgabeoperatoren `<<` und `>>`, auf deren linker Seite Objekte von Klassen der Standardbibliothek stehen müssen.

Für alle Operatoren, bei denen man sich entscheiden muß, ob sie als freie oder als Elementfunktionen implementiert werden, gelten folgende Daumenregeln:

1. Unäre Operatoren sollten i.A. als *Elementfunktionen* der Klasse implementiert werden, auf deren Instanzen sie angewendet werden.
2. Behandelt ein binärer Operator beide Operanden gleich (i.A. verändert er sie nicht), dann sollte dieser Operator als *freie Funktion* implementiert werden.
3. Behandelt ein binärer Operator seine Operanden nicht gleich (i.A. wird der linke verändert), dann sollte der Operator als *Elementfunktion* implementiert werden.

Natürlich kann es (wie bei allen Daumenregeln) in gut begründeten Fällen Ausnahmen geben.² Es ist allerdings relativ unwahrscheinlich, daß Sie in diesem Kurs Code schreiben, der eine solche Ausnahme rechtfertigt.

Operatoren, die als freie Funktionen implementiert sind, sind oft ein **friend** ihrer Operanden:

```
class X {
    friend bool operator==(const X& lhs, const X& rhs);
};
```

Da **friend** den Zugriff anderer Funktionen und Klassen auf private Elemente einer Klasse erlaubt, verletzt es ein wichtiges Prinzip der objektorientierten Programmierung: die Kapselung. Solcher Code läßt sich praktisch immer auch so umschreiben, daß er ohne **friend** auskommt, allerdings führt das oft zu Code, der entweder weniger effizient oder schwerer zu verstehen ist, oder zu Schnittstellen, die mehr

¹ Erfahrene C-Programmierer haben z.B. oft ein Problem damit, daß Strings in C++ per Operator `+` verknüpft werden. Weil Strings in C ja vom Typ `char []` bzw. `char*` sind, setzen sie „Strings“ mit Zeigern gleich – und die Addition von Zeigern ergibt praktisch nie einen Sinn.

Auch ist die Verknüpfung von Strings nicht kommutativ – die Addition sonst aber schon.

² Eine mögliche Ausnahme wäre z.B. die Implementierung der Inkrementierungs- und Dekrementierungsoperatoren für einen **enum**-Typen.

von der Implementierung preisgeben, als nötig. Deshalb gilt für die Verwendung von **friend**: So selten wie möglich, jedoch so oft wie nötig.

Boolsche Operatoren

Die relationalen Operatoren sollten (entsprechend der obigen Daumenregeln als freie Funktionen) i.A. wie folgt überladen werden:

```
inline bool operator==(const X& lhs, const X& rhs) { /* ... */ }
inline bool operator!=(const X& lhs, const X& rhs) {return !operator==(lhs, rhs);}
inline bool operator<(const X& lhs, const X& rhs) { /* ... */ }
inline bool operator>(const X& lhs, const X& rhs) {return operator<(rhs, lhs);}
inline bool operator<=(const X& lhs, const X& rhs) {return !operator>(lhs, rhs);}
inline bool operator>=(const X& lhs, const X& rhs) {return !operator<(lhs, rhs);}

```

Wichtig ist hier, daß nur zwei Operatoren wirklich implementiert werden. Die Implementierungen der anderen Operatoren basieren auf diesen. Wenn Algorithmen (wie `std::sort()`) und Typen (wie `std::map<>`) der Standardbibliothek Vergleichsoperatoren benötigen, benutzen diese immer und ausschließlich den `operator<()`. Benutzer Ihrer Klassen erwarten jedoch, daß, wenn Sie `a<b` schreiben könne, auch `a>b`, `a==b` usw. funktioniert.

Der (unäre) Negationsoperator sollte als Elementfunktion überladen werden. Allerdings ist es sehr unwahrscheinlich, daß Sie für diesen einen sinnvollen Anwendungsfall finden.

Für die anderen binären boolschen Operatoren (`||`, `&&`) gilt zwar eine den relationalen Operatoren vergleichbare Syntax, aber es ist extrem unwahrscheinlich, daß sie für diese einen sinnvollen Anwendungsfall finden werden.

Arithmetische Operatoren und Operatoren für Bitmanipulationen

Bei den unären arithmetischen Operatoren müssen Prefix- und Postfix-Varianten unterschieden werden.³ Letztere sind in der Performance ungünstiger, da ein zusätzliches temporäres Objekt benötigt wird. Sie können auf Basis der ersteren implementiert werden. Als Beispiel hier die Implementierung der Inkrementierungsoperatoren (die Dekrementierungsoperatoren werden analog implementiert):

```
class X {
    X& operator++()
    {
        /* inkrementieren */
        return *this;
    }
    X operator++(int)
    {
        X tmp(*this);
        operator++();
        return tmp;
    }
};

```

Die mit einer Zuweisung gekoppelten binären arithmetischen Operatoren verändern den linken Operanden und sollten deshalb als Elementfunktion implementiert werden. Die „normalen“ binären arithmetischen Operatoren verändern beide Operanden *nicht* und sollten deshalb als freie Funktionen implementiert sein. ANDREW KOENIG verdanken wir die Erkenntnis, daß letztere i.A. m.H. der ersteren implementiert werden können. Als Beispiel hier die Addition, alle anderen binären arithmetischen Operatoren werden nach demselben Schema überladen:

³ Um sich zu merken, daß das zusätzliche `int`-Argument bei der Postfix-Variante des Operators verwendet wird, sollte man sich daran erinnern, daß alle anderen unären Operatoren Prefix-Operatoren sind. Die Postfix-Varianten von `++` und `--` sind die einzigen Ausnahmen und haben daher auch eine besondere Syntax (eben das zusätzliche `int`-Argument).

```

class X {
    X& operator+=(const X& rhs)                { /* addieren */ return *this; }
};

inline X operator+(const X& lhs, const X& rhs)
{
    X tmp(lhs);
    tmp+=rhs;
    return tmp;
}

```

Die binären Operatoren zur Bitmanipulation (&, |, ^, ~, <<, >>) werden analog zu den binären arithmetischen Operatoren überladen. Allerdings gilt auch hier, daß es relativ unwahrscheinlich ist, daß Sie für diese einen sinnvollen Anwendungsfall finden.

Andere binäre Operatoren

Der Zuweisungsoperator sieht immer so aus⁴:

```

class X {
    X& operator=(const X& rhs)                { /* ... */ return *this; }
};

```

Bei der Implementierung von Zuweisungsoperatoren, die Ressourcen (z.B. Speicher) benötigen, sollte darauf geachtet werden, daß beim Holen der entsprechenden Ressourcen (**operator new**) Ausnahmen auftreten können. Es ist wichtig, den Zuweisungsoperator so zu implementieren, daß, falls eine Ausnahme auftritt und die Funktion abgebrochen wird, das Objekt nicht verändert wurde.⁵

Auch die Subskription (Feld-Zugriff) ist eine binäre Operation. (Die beiden Operanden sind der Index und das Objekt, welches indiziert wird). Ihre kanonische Form sieht so aus:

```

class X {
    value_type& operator[](index_type idx);
    const value_type& operator[](index_type idx) const;
};

```

Achtung: Ist `value_type` einer der eingebauten Datentypen, dann sollte statt einer konstanten Referenz natürlich eine Kopie zurückgegeben werden.

Zur Ein- und Ausgabe per Streams werden die Bit-Schiebe-Operatoren überladen. Für eigene Typen müssen sie – entgegen meiner Daumenregeln – als globale Funktionen überladen werden, weil wir die linken Operanden, die Stream-Klassen der Standardbibliothek, nicht nachträglich ändern können. Die kanonischen Formen für Eingabe und Ausgabe lauten:

```

std::ostream& operator<<(std::ostream& os, const T& obj)
{
    /* obj nach os schreiben */
    return os;
}

std::istream& operator>>(std::istream& is, T& obj)
{
    /* obj aus is lesen */
    if( /* Fehler beim Einlesen */ ) { is.setstate(std::ios::failbit); }
    return is;
}

```

Beachten Sie, daß Streams stets per nicht-konstanter Referenz übergeben werden.

⁴ Bitte denken Sie auch an die *Rule of Three*: Wenn Sie einen Zuweisungsoperator, einen Kopierkonstruktor oder einen Destruktor benötigen, dann benötigen Sie höchstwahrscheinlich alle drei.

⁵ Sonst würde z.B. der alte Speicher eines Strings schon freigegeben, neuer konnte nicht alloziert werden und beim Abbruch der Funktion durch eine Ausnahme bleibt ein „halbzugewiesenes“ Objekt übrig.

Der Funktionsaufrufoperator

Der Funktionsaufrufoperator ist der einzige Operator in C++, der beliebig viele Argumente haben kann. Er hat immer mindestens ein Argument, nämlich ein Objekt der Klasse, für die er überladen ist. Darüber hinaus kann er eine beliebige Anzahl von Funktionsargumenten haben.

Das folgende Beispiel zeigt ein Funktionsobjekt (oft auch „Funktork“ genannt), welches ein übergebenes Zeichen mit einem gespeicherten Zeichen vergleichen kann:

```
struct compare {
    compare(char c)                                : c_(c) {}
    bool operator()(char o) const                 { return o == c_; }
    char c_;
};
```

Das läßt sich z.B. für diese Funktion verwenden:

```
inline std::string::iterator find_if( std::string::iterator begin
                                     , std::string::iterator end
                                     , const compare& condition )
{
    while( begin!=end && !condition(*begin) ) ++begin;
    return begin;
}
```

Der folgende Code benutzt das Funktionsobjekt und diese Funktion, um einem String nach beliebigen Zeichen zu durchsuchen. Die beiden oben hervorgehobenen **const** (das erste wird wegen des zweiten benötigt) bewirken, daß ein solches Funktionsobjekt wie unten hervorgehoben auch als temporäres Objekt innerhalb des Funktionsaufrufes erzeugt werden kann:

```
void f3(std::string& str)
{
    compare cmp('X');
    std::string::iterator it1 = find_if( str.begin(), str.end(), cmp );
    std::string::iterator it2 = find_if( str.begin(), str.end(), compare('Y!') );
    // ...
}
```

Von der Standardbibliothek werden Funktionsobjekte (wie übrigens Iteratoren) allerdings immer per Kopie weitergegeben. Das Kopieren eines Funktionsobjektes sollte deshalb eine billige Operation sein.

Andere Operatoren

Für eigene Iteratoren oder Smart-Pointer werden u.a der unäre Dereferenzierungsoperator sowie der Pfeil-Operator benötigt:

```
class my_ptr {
    value_type& operator*();
    const value_type& operator*() const;
    value_type* operator->();
    const value_type* operator->() const;
};
```

Für den Pfeil-Operator muß `value_type` eine Klasse sein (damit auf deren Elemente zugegriffen werden kann).

Der unären Adressoperator sollte niemals überschrieben werden.⁶

Dasselbe gilt auch für den Kommaoperator. Meines Wissens ist bisher kein Anwendungsfall für einen überladenen Kommaoperator bekannt, der *nicht* zu abstrusem, unleserlichem und unverständlichem Code führt.

Da implizite Konvertierungen oft große Überraschungen hervorrufen, sollten i.A. auch keinerlei Umwandlungsoperatoren implementiert werden.

⁶ Laut PETE BECKER (von Dinkumware) ist es praktisch unmöglich, die Standardbibliothek zu implementieren, so daß deren Container-Klassen auch für Datenelemente korrekt funktioniert, für die dieser Operator überladen ist. (Das gilt natürlich nicht nur für die Standardbibliothek, sondern auch für vielen anderen Code.)

Die Operatoren `new` und `delete`

Globale Operatoren `new` und `delete`

Die globalen Operatoren `new` und `delete` gibt es in der Standardbibliothek in mehreren Varianten. Die wichtigsten sind:

```
void* operator new(size_t) throw(bad_alloc);
void operator delete(void*) throw();
void* operator new[](size_t) throw(bad_alloc);
void operator delete[](void*) throw();
```

Werden sie vom Benutzer überladen, ersetzen sie die von der Standardbibliothek bereitgestellten Varianten. Wird der Operator `new` überladen, so sollte *immer* auch Operator `delete` überladen werden. Im Allgemeinen sollten dann auch immer die Varianten zur Allokierung von Feldern überladen werden.

Placement

Das sogenannte placement new erlaubt es z.B., Objekte an bestimmten Adressen zu instanzieren:

```
class X { /* ... */ };
char buffer[ sizeof(X) ];
void f()
{
    X* p = new(buffer) X(/*...*/);
    // use 'p' ...
    p->~X(); // call destructor
}
```

Auch dafür stellt die Standardbibliothek überladene Operatoren zur Verfügung:

```
void* operator new(size_t, void* p) throw(bad_alloc);
void operator delete(void* p, void*) throw();
void* operator new[](size_t, void* p) throw(bad_alloc);
void operator delete[](void* p, void*) throw();
```

Die Operatoren `new` und `delete` lassen sich auch mit anderen zusätzlichen Argumenten überladen. Diese Varianten werden meist auch dann als *placement new* bezeichnet, wenn sie einem völlig anderen Zweck dienen.

Klassenspezifische Operatoren `new` und `delete`

Außerdem lassen sich `new` und `delete` klassenspezifisch überladen.

```
class my_class {
public:
    // ...
    void* operator new();
    void operator delete(void*, size_t);
    void* operator new[](size_t);
    void operator delete[](void*, size_t);
    // ...
};
```

Sie verhalten sich dann, als wären sie statische Elementfunktionen.

Da die Operatoren ja klassenspezifisch sind, wird das Argument `size_t` des Operators `new` zwar scheinbar nicht gebraucht, sollte aber dennoch beachtet werden. Es ist ja möglich, daß jemand von `my_class` ableitet⁷. In diesem Falle würde das Argument die Größe eines Objektes der abgeleiteten Klasse angeben. So etwas ist dann aber u.U. recht schwer zu implementieren.

⁷ Dann muß die Klasse natürlich einen virtuellen Destruktor haben.