

Kurseinheit 7: Überladen von Operatoren

1. Operatoren und Datentypen
2. Default-Operatoren bei Klassen
3. Überladen von Operatoren
4. Überladen wichtiger Operatoren

1. Operatoren und Datentypen

Überblick C Operatoren

C/C++ Coding
Styleguide

Probleme:

Prioritäten müssen
bekannt sein!

Assoziativität meist
von L nach R.

Regel PA4:

„...
Ausdrücke mit
verschiedenen
Operatoren sind
zu klammern.“

Priorität und Assoziativität		
Priorität: (niedrigerer Wert entspricht höherer Priorität)		Assoziativität
1	++(Postfix) --(Postfix) () [] -> .	L nach R
2	++(Präfix) --(Präfix) ! ~ -(unär) +(unär) &(Adresse) *(Indirektion) sizeof type(Typumwandung/cast)	R nach L
3	*(Multiplikation) / %	L nach R
4	+(Addition) -(Subtraktion)	L nach R
5	<< >>	L nach R
6	< <= > >=	L nach R
7	== !=	L nach R
8	&(bitweise Und)	L nach R
9	^	L nach R
10		L nach R
11	&&	L nach R
12		L nach R
13	?:	R nach L
14	= *= /= %= += -= <<= >>= &= ^= =	R nach L
15	,	L nach R

1. Operatoren und Datentypen

3

Arten von Operatoren

- Unäre Operatoren (ein Operand)
- Binäre Operatoren (zwei Operanden)
- Ternäre Operatoren (drei Operanden) – in C/C++ nur ?: vorhanden ... und dieser sollte auch noch nach vielen Coding Styleguides vermieden werden.

Eine Operation hat einen oder mehrere Operanden. Eine Kaskadierung ist möglich und häufig notwendig!



Operation 1: Operator ist +, Operanden sind s32V1 und s32V2

Operation 2: Operator ist =, Operanden sind s32V1 und Ergebnis von Operation 1

Ein Operatorsymbol (z.B. „-“) kann mehrfach belegt sein – Unterschied: Anzahl Operanden

1. Operatoren und Datentypen

4

Herausforderung

```
int32_t s32V1;  
int32_t s32V2 = 73;  
  
s32V1 = s32V2;           // okay  
  
s32V1 = s32V1 + s32V2; // okay  
s32V1 = s32V1 / s32V2; // okay
```

Bei Standarddatentypen muss nicht überlegt werden:

Kann dieser Operator verwendet werden oder nicht?

Operatoren sind für Standarddatentypen oft „gemacht“.

```
A A1;  
A A2;  
  
A1 = A2;           // okay  
  
A1 = A1 + A2;      // Error  
A1 = A1 / A2;      // Error
```

Eigene Datentypen (Klassen, Strukturen) können nicht per se für die meisten Operatoren verwendet werden!

Je nach Anwendung kann es aber sinnvoll sein, Operatoren für die eigenen Datentypen zu haben.

2. Default-Operatoren bei Klassen

5

Welche Default-Operatoren gibt es?

```
class Empty
{
};
```

auto generated

```
class Empty
{
public:
    Empty() = default;
    Empty(const Empty&) = default;
    Empty(Empty&&) noexcept = default;
    ~Empty() = default;

    // Copy-Assign und Move-Assign Operator
    Empty& operator=(const Empty&) = default;
    Empty& operator=(Empty&&) noexcept = default;
};
```

Eine leere Klasse hat automatisch den Copy-Assign und einen Move-Assign Operator.

```
Empty E1;
Empty E2;
Empty E3;
```

```
E2 = E1;
E3 = std::mov(E1);
```

Elektrotechnik, Medizintechnik und Informatik

C++ - KE07: Überladen von Operatoren

Prof. Dr.-Ing. Daniel Fischer - Version 3.0.1

2. Default-Operatoren bei Klassen

6

Wegfall der Default-Operatoren?

If you explicitly define:

	Nothing	Destructor	Copy Constructor	Copy Assignment	Move Constructor	Move Assignment
Destructor ~Foo()	✓	✓	✓	✓	✓	✓
Copy Constructor Foo(const Foo&)	✓	✓	✓	✓		
Copy Assignment Foo& operator=(const Foo&)	✓	✓	✓	✓		
Move Constructor Foo(Foo&&)	✓		Copies are used in place of moves		✓	
Move Assignment Foo& operator=(Foo&&)	✓					✓

Figure 4-8: A chart illustrating which methods the compiler generates when given various inputs [Los19]

Rule of Zero: Keine Implementierung (Nothing) -> Destructor + Copy/Move

Rule of Three: Destructor oder Copy Constructor oder Copy Assignment: Dann sind nur Destructor, Copy Constructor und Copy Assign vorhanden. Moves sind weg!

Elektrotechnik, Medizintechnik und Informatik

C++ - KE07: Überladen von Operatoren

Prof. Dr.-Ing. Daniel Fischer - Version 3.0.1

3

3. Überladen von Operatoren

7

Überladbare und nicht überladbare C++ Operatoren

Die folgenden C++ Operatoren können überladen werden:

```
() [] -> ->* & * ~ ! + - ++ -- new delete * / % >> << < > >= <= == != ^  
| && || = += -= *= /= %= <<= >>= &= ^= |=
```

Kompakte Darstellung zweier Operationen

Die folgenden C++ Operatoren können **nicht** überladen werden:

```
:: . sizeof .* ?: typeid
```

Bei **C++ Operatoren** handelt es sich um **Funktionen**, welche unterschiedlich notiert sind und unterschiedlich aufgerufen werden.

```
s32V1 = s32V1 + s32V2; ← äquivalent → s32V1 = s32plus(s32V1, s32V2);
```

Die C++ Operatoren ->* sowie .* werden im Rahmen der LV nicht behandelt.

3. Überladen von Operatoren

8

Implementierung

Will man (oder muss man) einen Operator überladen, so muss man diesen Operator als **Funktion** implementieren. Es stehen dabei die beiden folgenden Möglichkeiten zur Verfügung:

- Operatorfunktion als Methode einer Klasse (häufiger in der Praxis)

Hinweis: Unsichtbarer this-Zeiger

```
Rückgabewert Klassenname::operator # (Argumentenliste);
```

↓
Operatorsymbol

- Operatorfunktion als friend-Methode

```
Rückgabewert operator # (Klasse[&], Argumentenliste);
```

↓
Operatorsymbol

Auf den ersten Blick sind beide Varianten äquivalent – bei einer genauen Betrachtung wird später erkennbar sein, dass in **Ausnahmefällen nur eine Operatorfunktion** möglich ist.

3. Überladen von Operatoren

9


Aufruf der Operatoren

Operatorfunktionen können **implizit** oder **explizit** aufgerufen werden!

Bsp.: `A3 = A1 + A2;` `A1, A2` und `A3` seien Objekte einer Klasse `A`

	implizit	explizit
Operatorfunktion als Methode einer Klasse	<code>A3 = A1 + A2;</code>	<code>A3 = A1.operator+(A2);</code>
Operatorfunktion als friend-Methode	<code>A3 = A1 + A2;</code>	<code>A3 = operator+(A1, A2);</code>

Beim impliziten Aufruf der Operatorfunktionen ist kein syntaktischer Unterschied vorhanden. In der Praxis werden Operatorfunktionen meist implizit aufgerufen.


 Elektrotechnik, Medizintechnik und Informatik

C++ - KE07: Überladen von Operatoren

Prof. Dr.-Ing. Daniel Fischer - Version 3.0.1

3. Überladen von Operatoren

10

Beispiel: Operator als Methode einer Klasse

```


class Complex
{
public:
    Complex(f64_t f64Real, f64_t f64Img) :
        f64Real_(f64Real), f64Img_(f64Img) {}
    void vPrint()
    {
        std::cout << f64Real_ << " +j " << f64Img_ << std::endl;
    }
    Complex operator+(const Complex& rComplex)
    {
        Complex ComplexTemp(0., 0.);
        ComplexTemp.f64Real_ = f64Real_ + rComplex.f64Real_;
        ComplexTemp.f64Img_ = f64Img_ + rComplex.f64Img_;
        return ComplexTemp;
    }
private:
    f64_t f64Real_;
    f64_t f64Img_;
};

```

`typedef double f64_t;`
`// not in stdint`

Anzahl und
 Datentyp der
 Über-
 gabeparameter?

this-Objekt
 soll nicht
 überschrie-
 ben werden.
 Daher Temp-
 Objekt als
 return.


 Elektrotechnik, Medizintechnik und Informatik

C++ - KE07: Überladen von Operatoren

Prof. Dr.-Ing. Daniel Fischer - Version 3.0.1

3. Überladen von Operatoren

11

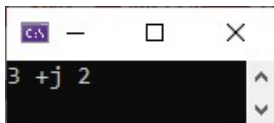
Beispiel: Operator als Methode einer Klasse

```
Complex C1(1., 1.);  
Complex C2(2., 1.);  
  
Complex C3(0., 0.);  
  
C3 = C1 + C2;  
C3.vPrint();
```

Operation 1: Addition C1 + C2

Aufruf des +Operators

Es wird die Operatorfunktion des ersten Operanden aufgerufen (this-Zeiger wird implizit mitgegeben) und der zweite Operand als formaler Parameter mitgegeben. Operation liefert ein temporäres Objekt ComplexTemp zurück!



Operation 2: Zuweisung c3 = ComplexTemp;

Es wird hier der existierende Default-Zuweisungsoperator aufgerufen.

Wie viele Complex-Objekte werden im obigen Programm instanziiert? _____

Was passiert, wenn jetzt in Complex noch der MoveAssign-Konstruktor noch implementiert wird?

3. Überladen von Operatoren

12

Beispiel: Operator als friend-Methode einer Klasse

```
class Complex  
{  
public:  
    // more Code  
private:  
    f64_t f64Real_;  
    f64_t f64Img_;  
  
    friend Complex operator-(const Complex& rC1, const Complex& rC2);  
};  
  
Complex operator-(const Complex& rC1, const Complex& rC2)  
{  
    Complex ComplexTemp(0., 0.);  
    ComplexTemp.f64Real_ = rC1.f64Real_ - rC2.f64Real_;  
    ComplexTemp.f64Img_ = rC1.f64Img_ - rC2.f64Img_;  
    return ComplexTemp;  
}
```

```
typedef double f64_t;  
// not in stdint
```

Friend-Deklaration gehört nicht zum private-Bereich

3. Überladen von Operatoren

13

Beispiel: Operator als friend-Methode einer Klasse

```
Complex C1(1., 1.);  
Complex C2(2., 1.);  
  
Complex C4(0., 0.);  
  
C4 = C1 - C2;  
C4.vPrint();
```



Operation 1: Subtraktion C1 - C2

Jetzt wird die als friend deklarierte Operatorfunktion aufgerufen.

Die Objekte C1 und C2 werden dabei als formale Parameter (konstante Referenzen) übergeben.

Die Operatorfunktion liefert ein temporäres Complex-Objekt ComplexTemp zurück.

Operation 2: Zuweisung C4 = ComplexTemp

Es wird hier der existierende Default-Zuweisungsoperator aufgerufen.

Wiederholung: Beim impliziten Aufruf der Operatorfunktionen ist kein syntaktischer Unterschied zwischen einer Klassen- und Friendmethode vorhanden. Es ist somit im obigen Code nicht erkennbar, wie der Operator implementiert wurde.

3. Überladen von Operatoren

14

Überladen von überladenen Operatoren

```
class Complex  
{  
public: // more Code  
    Complex operator+(const Complex& rComplex)  
    {  
        Complex ComplexTemp(0., 0.);  
        ComplexTemp.f64Real_ = f64Real_ + rComplex.f64Real_;  
        ComplexTemp.f64Img_ = f64Img_ + rComplex.f64Img_;  
        return ComplexTemp;  
    }  
    Complex operator+(f64_t f64Real)  
    {  
        Complex ComplexTemp(0., 0.);  
        ComplexTemp.f64Real_ = f64Real_ + f64Real;  
        ComplexTemp.f64Img_ = f64Img_;  
        return ComplexTemp;  
    }  
private:  
    // more Code  
};
```

Wiederholung:

In C++ können Funktionen überladen werden. Gleicher Funktionsname, aber unterschiedliche Übergabeparameter.

Da Operatoren intern auch Funktionen sind, können diese auch nochmals überladen werden.

3. Überladen von Operatoren

15

Überladen von überladenen Operatoren

```
class Complex
{
public: // more Code
    Complex operator+(const Complex& rComplex)
    { // more Code }
    Complex operator+(f64_t f64Real)
    { // more Code }
private:
    // more Code
};
```

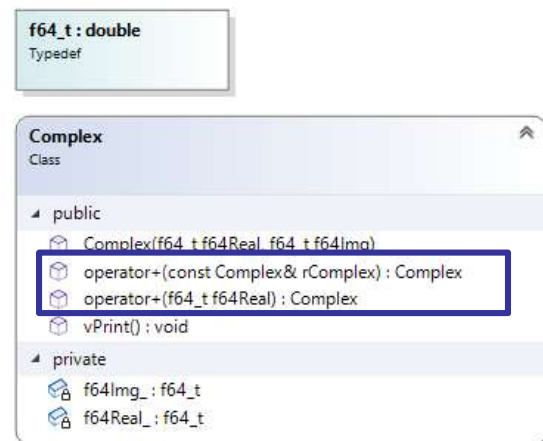
Die überladenen Operatoren können selbst wieder überladen werden. Da sich die Anzahl der formalen Parameter (durch Operator vorgegeben) nicht ändern darf, kann man nur noch im **Datentyp variieren**.

Operatoren als Elementfunktion einer Klasse sowie als Friend-Funktionen dürfen nicht redundant (gleicher Datentyp der formalen Parameter) für einen Operator sein.

3. Überladen von Operatoren

16

Anzeige von Überladung von überladenen Operatoren



Der Class Designer kann überladene Operatoren anzeigen, wenn unter Change Members Format die Einstellung Display Full Signature ausgewählt ist.

Auch das verwendete Typedef für double (64 Bit nach IEEE754) wird angezeigt.

Die selbstsprechenden formalen Parameter wie f64Real werden auch ins Klassendiagramm übernommen.

3. Überladen von Operatoren

17

Zusammenfassung

- Operatorfunktion ist stets klassenbezogen (statisch [nur friend] oder nicht statisch)
- Prioritäten der Operatoren können nicht verändert werden
- Assoziativität der Operatoren können nicht verändert werden
- Operandenanzahl bleibt unverändert (Ausnahme new und delete)
- Es können keine Defaultwerte (Parameterübergabe) vergeben werden
- Es können **keine neuen** Operatorsymbole eingeführt werden

Ungenutzte Operatorsymbole verwenden, z.B. Bitshift-Operatoren >> und << für cin und cout

Ansonsten Implementierung der Operationen in einer Funktion ohne Operatorsymbol wie bisher praktiziert:

```
f64_t X::f64GetMax(f64_t f64A, f64_t f64B,  
                  f64_t f64_C)
```

4. Überladen wichtiger Operatoren

18

<< für den Outputstream

Bisher wurde zur Ausgabe der Basistypen sowie zur Ausgabe eines String-Objektes der <<-Operator verwendet.

```
std::string StringText{"Hello "};  
int32_t s32SheldonsPerfectNumber = 73;  
  
std::cout << StringText << s32SheldonsPerfectNumber << std::endl;
```

Das globale Objekt cout (schon im Namespace std instanziiert) vom Typ ostream hat den Operator << Methode einer Klasse überladen. Beispiel:

```
ostream& operator<< (int val);
```

Jetzt wäre es wünschenswert, wenn eigene Objekte über den gleichen Mechanismus (inklusive Verkettung) ausgegeben werden könnten.

Zur Zahl 73: Die beste (perfekte) Zahl: <https://www.youtube.com/watch?v=33pH6ELDEeI>

4. Überladen wichtiger Operatoren

19

<< für den Outputstream

```
class Number
{
public:
    // more Code

private:
    int32_t s32Num_;

friend std::ostream& operator<<(std::ostream& ost, Number Number1);
};

std::ostream& operator<<(std::ostream& ostr, Number Number1)
{
    ostr << "Number is: " << Number1.s32Num_;
    return ostr;
}
```

Klassen, bzw. deren Objekte, die Attribute über `std::cout` ausgeben, müssen einen Operator überladen (es bietet sich der `<<` von `cout` an – Einheitlichkeit), der als erster Parameter ein `ostream&` erhält. Daher kann dies **nur als friend-Methode** realisiert werden.

3. Überladen wichtiger Operatoren

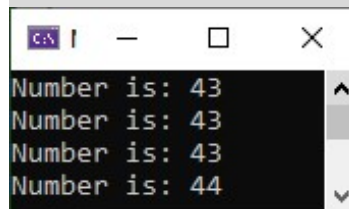
20

Präfix und Postfix

```
class Number
{
public:
    // more Code
    Number& operator++() ← Für Präfix
    {
        s32Num_++;
        return (*this);
    }
    Number operator++(int32_t s32Dummy) ← Für Postfix
    {
        Number NumberTemp = *this;
        s32Num_++;
        return NumberTemp;
    }
private:
    int32_t s32Num_;
};
```

Die beiden Operatoren `++` und `--` gibt es in einer Präfix und Postfix-Variante. Eine Unterscheidung im Code wäre somit nicht möglich. C++ stellt allerdings über eine Dummyvariable einen Mechanismus dar, damit Post- und Präfix unterschieden werden können.

```
Number N1{42};
std::cout << ++N1 << std::endl;
std::cout << N1 << std::endl;
std::cout << N1++ << std::endl;
std::cout << N1 << std::endl;
```



3. Überladen wichtiger Operatoren

21

Verkettung bei Präfix und Postfix

Präfix

```
Number& operator++()  
{  
    s32Num_++;  
    return (*this);  
}
```

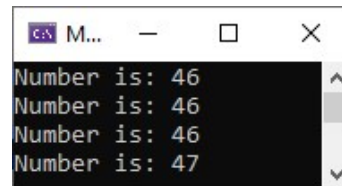
Liefert eine **Referenz auf das Original** zurück.
Verkettung ist **möglich**.

Postfix

```
Number operator++(int32_t s32Dummy)  
{  
    Number NumberTemp = *this;  
    s32Num_++;  
    return NumberTemp;  
}
```

Liefert ein **temporäres Objekt** zurück.
Korrekte Verkettung **nicht möglich**.

```
Number N1{42};  
  
std::cout <<+++++++N1 << std::endl;  
std::cout << N1 << std::endl;  
std::cout << N1+++++++ << std::endl;  
std::cout << N1 << std::endl;
```



Warum gibt Postfix keine Referenz auf NumberTemp zurück?

4. Überladen wichtiger Operatoren

22

Rückgabe von Referenzen

Immer dann, wenn ein Operator den Wert eines Operanden verändert, ist eine Referenz zurückzugeben.

Beispiele:

Hinweis: Unsichtbarer this-Zeiger

```
Number& operator++()  
{  
    s32Num_++;  
    return (*this);  
}
```

Operator ändert den Wert
eines Operanden (hier this).

Hinweis: Unsichtbarer this-Zeiger

```
Complex operator+(const Complex& rComplex)  
{  
    Complex ComplexTemp(0., 0.);  
    ComplexTemp.f64Real_ = f64Real_ +  
                           rComplex.f64Real_;  
    ComplexTemp.f64Img_ = f64Img_ +  
                          rComplex.f64Img_;  
    return ComplexTemp;  
}
```

Operator ändert den Wert der beiden Operanden nicht.

Methoden wurden hier in h-Datei implementiert -> automatisches Inlining.

4. Überladen wichtiger Operatoren

23

Zuweisungsoperator

```
class A
{
public:
    A(char* pcStr)
    {
        pcStr_ = static_cast<char*> (malloc(strlen(pcStr) + 1));
        if (pcStr_ != nullptr)
        {
            strcpy_s(pcStr_, strlen(pcStr), pcStr);
        }
    }
    virtual ~A()
    {
        free(pcStr_); // if pcStr_ is nullptr, nothing is done
    }
private:
    char* pcStr_;
    friend std::ostream& operator<<(std::ostream& ostr, A& rA1);
};
```

4. Überladen wichtiger Operatoren

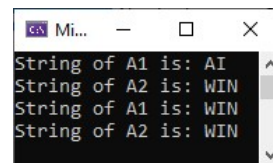
24

Zuweisungsoperator

Der Default-Zuweisungsoperator kopiert (wie auch der Kopierkonstruktor) die Attribute immer nur elementweise. Nur **flache Kopien** bei Referenzen oder Pointern.

```
char szAI[] = "AI";
char szWIN[] = "WIN";
A A1(szAI);
A A2(szWIN);

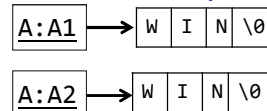
std::cout << "String of A1 is: " << A1 << std::endl;
std::cout << "String of A2 is: " << A2 << std::endl;
A1 = A2;
std::cout << "String of A1 is: " << A1 << std::endl;
std::cout << "String of A2 is: " << A2 << std::endl;
```



```
String of A1 is: AI
String of A2 is: WIN
String of A1 is: WIN
String of A2 is: WIN
```

Zuweisungsoperator ist zu überladen.

Wunsch: Tiefe Kopie



z.B. mit realloc

4. Überladen wichtiger Operatoren

25

Index-Operator

```
char& operator[] (int32_t s32Index) noexcept (false)
{
    if (s32Index < -1 && s32Index <= strlen(pcStr_))
    {
        return (pcStr_[s32Index]);
    }
    else
    {
        throw std::invalid_argument("s32Index out of range");
    }
}
```

Exception muss geworfen werden!

Der Index-Operator gibt eine Referenz auf char zurück! **Schreiben und Lesen möglich!**

```
std::cout << A2[2] << std::endl;
A2[2] = '-';
std::cout << "String of A2 is: "
          << A2 << std::endl;
```

Try- und catch-Block zur Vereinfachung weggelassen.

