C++ - Operatoren

Eigene Klassen mit neuen Funktionen

Übersicht

- Klassen bisher
- Eigene Operatoren definieren

Bisher

- Durch Kapselung, Vererbung und Polymorphy können nun eigene Klassen definiert werden, die Eigenschaften anderer Klassen nutzen und erweitern.
- Es können neue Methoden hinzugefügt werden, mit denen die original Methoden überladen werden.
- Jedoch – was ist mit den durch die Sprache C++ vorgegebenen Operatoren wie:
  - +
  - +=,…

```
class Point {
private:
   double X , Y;
public:
    Point(double xx, double yy)
   {X=xx; Y=yy;}
        …
}

main(){
Point A , B ;

A += B;      // ???
A++;         // ???
```

Überladung von Operatoren

- Für die Verarbeitung von Objekten einer Klasse lassen sich eigene Operatoren (Operatorfunktionen) definieren, die den Operatoren auf intrinsische Datentypen überladen sind.
- Eigenschaften überladener Operatoren:
  - Operatoren, die Basisvariablen verknüpfen, lassen sich nicht überladen, d.h. wenigstens ein Operand muss Objekt einer Klasse sein.
  - Nicht alle Operatoren können überladen werden.
  - Es können keine neuen Symbole für Operationen kreiert werden, sondern nur bestehende überladen werden.
  - Die Hierarchie der Operatoren ist unveränderbar.
  - Die Stelligkeit (unär, binär) der Operatoren ist unveränderbar.
  - Die unären Inkrement (++)- und Dekrementoperatoren (--) erfahren eine Differenzierung als Prä- oder Postfix-Operator durch Verwendung eines Dummy-Arguments (int) für die Postfix-Variante.
  - Die Operatoren [ ], ( ), = und -> müssen Mitglieder einer Klasse sein und dürfen nicht als static deklariert werden.

Überladung von Operatoren

Prio.	Operatoren	Ass.	Stell.	Überl.	Wirkung
1	::	rechts	1	N	globaler Bezug
1	::	links	2	N	Klassenbezug
2	.	links	2	N	Memberzugriff mit Objekt
2	->	links	2	J	Memberzugriff mit Objektzeiger
2	[ ]	links	1	J	Feldindex
2	( )	links	1	J	Funktionsaufruf
2	typ (variable)	links	1	J	Typkonvertierung
2	++, --	rechts	1	J	Inkrement, Dekrement
3	sizeof	links	1	N	Speicherbedarf
3	!	rechts	1	J	Negation logisch
3	~	rechts	1	J	Negation bitweise
3	+, -	rechts	1	J	Vorzeichen (unär)
3	*	rechts	1	J	Derefenzierung
3	&	rechts	1	J	Adresse
3	new	rechts	1	J	Speicherallokation
3	delete	rechts	1	J	Speicherfreigabe
3	(typ)	rechts	1	J	Typkonvertierung
4	->*	links	2	J	Memberzeiger mit Objektzeiger
4	.*	links	2	N	Memberzeiger mit Objekt

Überladung von Operatoren

Prio.	Operatoren	Ass.	Stell.	Überl.	Wirkung
5	*, /	links	2	J	Multiplikation, Division
5	%	links	2	J	modulo
6	+, -	links	2	J	Summe, Differenz
7	<<, >>	links	2	J	bitweises Schieben
8	<, <=, >, >=	links	2	J	Relation
9	==, !=	links	2	J	Gleichheit, Ungleichheit
10	&	links	2	J	bitweises AND
11	^	links	2	J	bitweises XOR
12	|	links	2	J	bitweises OR
13	&&	links	2	J	logisches AND
14	||	links	2	J	logisches OR
15	?:	links	3	N	arithmetisches IF
16	+=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=	rechts	2	J	Zuweisung
17	,	links	2	J	Komma-Operator

Optionen zur Definition eines Operators

- Der Operator ist Member einer Klasse
  Syntax:
  rückgabetyp klassenname:: **operator** operatorsymbol (formalparameterliste)
  {
  Anweisungen;
  }
- Der Operator ist nicht Member einer Klasse
  Syntax:
  rückgabetyp **operator** operatorsymbol (formalparameterliste)
  {
  Anweisungen;
  }
- Der Operator ist nicht Member einer Klasse aber friend einer Klasse
  Syntax:
  rückgabetyp **operator** operatorsymbol (formalparameterliste)
  {
  Anweisungen;
  }

Umsetzung der Operatorsyntax in einen Operatorfunktionsaufruf:

Klassenmember	Syntax	Operatorfunktion
nein	x ⊗ y	operator ⊗ (x, y)
nein	⊗ x	operator ⊗ (x)
nein	x ⊗	operator ⊗ (x,0)
ja	x ⊗ y	x.operator ⊗ (y)
ja	⊗ x	x.operator ⊗ ( )
ja	x ⊗	x.operator ⊗ (0)
ja	x = y	x.operator = (y)
ja	x(y)	x.operator ( ) (y)
ja	x[y]	x.operator [ ] (y)
ja	x->	(x.operator −>( )) −>

Beispiel: Klasse Point

```
class Point
    {
    friend Point operator - (const Point &obj);                 //unär -
    friend Point operator * (double z, const Point &obj);       //binär *
    private:
          double x, y;
    public:
          Point(void) {x=0.; y=0.;}                          //Standard-Konstruktor
          Point(const Point &obj) {x=obj.x; y=obj.y;} //Kopier-Konstruktor

          Point(double xx, double yy) {x=xx; y=yy;} //generischer Konstruktor

          Point operator ++ (void);                  //unär ++ (prefix)
          Point operator + (const Point &obj); //binär + (return object)
    };
Point Point::operator ++ (void)                 //Definition Klassenmember 1
    {
          x += 1.;
          y += 1.;
          return *this;     //Übergabe des aufrufenden
                            //Objekts durch Dereferenzierung
    }
```

Beispiel: Klasse Point

```
Point Point::operator + (const Point &obj)  //Definition Klassenmember 2
{
    return Point(x + obj.x, y + obj.y);     //Rückgabe mit Konstruktoraufruf
}

Point operator - (const Point &obj)          //Definition friend-Funktion 1
{
    return Point(-obj.x, -obj.y);            //Rückgabe mit Konstruktoraufruf
}

Point operator * (double z, const Point &obj)   //Definition friend-
                                                //Funktion 2
{
    return Point(z*obj.x, z*obj.y);   //Rückgabe mit Konstruktoraufruf
}

......
Point p1(1., 2.), p2(3., 4.);
p1++;                                 //p1.x = 2., p1.y = 3.
Point p3 = -p1 + p2;                  //unär -, binär +
Point p4 = 2.* p1;                    //p4.x = 4., p4.y = 6.
```

Eigenschaften des Zuweisungsoperators =

- Der Zuweisungsoperator kann einem Objekt ein anderes Objekt der gleichen oder einer anderen Klasse zuweisen.
- Ist der Zuweisungsoperator nicht explizit definiert und ein Objekt der gleichen Klasse wird zugewiesen, dann kopiert der Compiler die Daten des einen Objekts in das zweite Objekt. Die Konsequenzen sind vergleichbar denen bei fehlendem Kopier-Konstruktor.
- Die Operatorfunktion muss als Member der Klasse definiert werden.
- Für die Zuweisung muss das Zielobjekt bereits definiert sein. Wird das Operatorsymbol „=“ bei der Definition verwendet, erfolgt der Aufruf des Kopier-Konstruktors, nicht der Aufruf der Operatorfunktion.
- Für die Verwendung kombinierter Zuweisungen muss das Objekt, das den Aufruf tätigt, mit *this zurückgegeben werden.
  Beispiel: Point p5 = p1 += p2

Definition eines Konversionsoperators (cast)

Für die Konversion eines Objekts in ein anderes kann ein Konversionsoperator definiert werden.

Beispiel:

```
class Demo
{
private:
    double xxd;
public:
    Demo(double x) {xxd = x;}          //„Konversions“-Konstruktor

    operator int(void) {return (int) xxd;} //Konversionsoperator
};

.......
Demo objDemo = 3.1;          //Instanzierung + Initialisierung
int n;
n = objDemo;                 //impliziter Aufruf von int( )
n = (int) objDemo;           //expliziter Aufruf (cast-Variante)
n = int (objDemo);           //expliziter Aufruf (function-Variante)
```

Eigenschaften eines Konversionsoperators (cast)

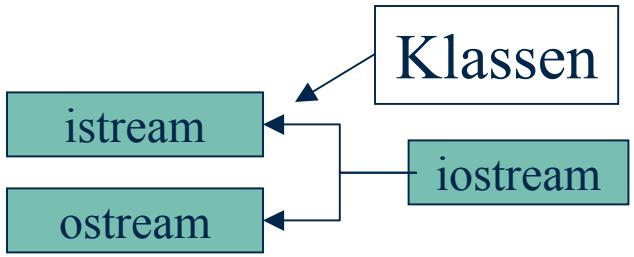
- Die Konversionsoperatorfunktion muss Member der Quellklasse sein.
- Ein return-Typ darf nicht angegeben werden, da dieser automatisch aus dem Operatornamen ermittelt wird.
- Eine return-Anweisung muss in der Definition angegeben werden.
- Der Datentyp des Formalparameters muss void sein.
- Die Quellklasse muss eine vom Anwender definierte Klasse sein. Konversionsoperatoren für Basisvariablentypen können nicht überladen werden.
- Die Zielklasse kann eine vom Anwender definierte Klasse oder ein Basisvariablentyp sein.
- Für den gleichen Zweck können nicht gleichzeitig ein Konversions-Operator und ein Konversions-Konstruktor eingesetzt werden.
- Ist die Zielklasse ein Basisvariablentyp, muss ein Konversions-Operator verwendet werden.

Empfehlungen für die Auswahl unter den Optionen zur Operatordefinition:

- Steht der Quellcode der Klasse nicht zur Verfügung, muss die Operatordefinition als Nicht-Member-Funktion realisiert werden.
- Muss auf private-Member-Daten unterschiedlicher Klassen zugegriffen werden, empfiehlt sich die Verwendung von friend-Funktionen.
- Ist ein binärer Operator zu definieren und der erste Operand ist ein Basisvariablentyp, kann eine Member-Funktion zur Operatordefinition nicht verwendet werden.
  Beispiel: Point p4 = 2.* p1
- Die Operatoren [ ], ( ), = und -> müssen als Mitglieder einer Klasse definiert werden.
- Für die Verwendung kombinierter Zuweisungen muss das Objekt, das den Aufruf tätigt, mit *this zurückgegeben werden.
  Beispiel: Point p5 = ++p1 + p2

Beispiel: Point Ausgabe <<

- Gesucht ist der Operator <<
  in Verbindung mit dem Ausgabestrom ostream



- Zunächst muss geklärt werden ob sich der Operator überladen lässt und wenn ja – was für ein Typ er ist
  - JA, binär

Prio.	Operatoren	Ass.	Stell.	Überl.
7	<<, >>	links	2	J

- In welcher Form kann ein binärer Operator geschrieben werden
  - 2 Möglichkeiten
    Klassenmember Ja und Nein
  - Aber (siehe oben) es geht nicht um die Erweiterung von Point sondern um ostream
    - Nur eine Nicht Member Funktion bleibt

Klassenmember	Syntax	Operatorfunktion
nein	$x \otimes y$	operator $\otimes$ (x, y)
ja	$x \otimes y$	x.operator $\otimes$ (y)

Beispiel: Point Ausgabe <<

- D.h. die Operatorfunktion muss nach dem Muster
  operator<< (x,y)
  gebildet werden
- Damit eine Verkettung von Streamoperation funktioniert
  cout << x << y << z
  muss jeder Operator selbst wieder eine Referenz auf das Objekt liefern d.h. Der Rückgabewert ist :
  ostream& operator <<
- Der erste Parameter ist eine Referenz des Streams selbst (i.e. genau der Wert, den ein vorhergehender Operator liefert
  ostream& operator << (ostream& Ausgabe,
- Der zweite Parameter benötigt ein Point Objekt – in diesem Fall (als kompl. Objekt) eine Referenz auf ein Point Objekt
  ostream& operator << (ostream& Ausgabe, Point& P)

Beispiel: Point Ausgabe <<

- Als letztes ist zu entscheiden was die Operatorfunktion machen soll:

```
ostream& operator<< (ostream &Ausgabe , const Point &P )
{
    Ausgabe << "P(" << P.x << "," << P.y << ")";
    return Ausgabe;
}
```

  in diesem Fall greift sie auf die "privaten" Attribute x und y zu

- D.h.Die Operatorfunktion muss ein Freund der Klasse Point sein:

```
friend ostream& operator<< (ostream &Ausgabe , const Point &P );
```

  muss in der Klasse Point definiert sein.