Überladen von Operatoren

- Neben Funktionen bzw. Methoden lassen sich in C++ auch **Operatoren überladen**.

- **Ziel** der Operatorüberladung ist es, die vorhandenen **C++ - Operatoren** auch auf **Objekte** anwenden zu können. Damit werden die **Programme lesbarer** und **sicherer** :

```
class ratio {          // ratio.cpp
    int zaehl, nenn;  // Zaehler, Nenner
  public:
    void kuerze(){ /* Kuerzen mittels ggT (Algorithmus von Euklid) */ }

    ratio(int z, int n=1):zaehl(z), nenn(n){ if(!nenn) nenn=1; kuerze(); }

    ratio operator*(const ratio &x){
        ratio r(this->zaehl*x.zaehl,this->nenn*x.nenn); r.kuerze(); return r;
    }

    ratio operator+(const ratio &x){
        ratio r(this->zaehl*x.nenn+x.zaehl*this->nenn, this->nenn*x.nenn); r.kuerze(); return r;
    }

    ratio operator-(const ratio &x){
        ratio r(this->zaehl*x.nenn-x.zaehl*this->nenn, this->nenn*x.nenn); r.kuerze(); return r;
    }

    ratio operator-(){ ratio r(-this->zaehl, this->nenn); return r; }  //ueberladen: operator-()

    int getzaehl() const { return zaehl; }   // Lesen zaehl

    int getnenn() const { return nenn; }     // Lesen nenn
};
```

Überladen von Operatoren

```
ostream &operator<<(ostream &cout, ratio &r){ return cout<<r.getzaehl()<<" / "<<r.getnenn(); }

void main(){ ratio *r1 = new ratio(1,2), *r2 = new ratio(3,5),
                  *r3 = new ratio(2,5), *r4 = new ratio(1,10);

             ratio r5 = *r1 * *r2 + *r3 - -*r4; cout<<"r5 = "<<r5<<endl;

            r5 = ((r1->operator*(*r2)).operator+(*r3)).operator-((*r4).operator-()); // alternativ

             cout<<"r5 = "<<r5<<endl;
}
```

- Folgende C++ - **Operatoren** lassen sich **überladen**:

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

- Benennung: **Indexoperator []** , **Aufrufoperator ()** , **Dereferenzoperator ->**

- Operatoren, die sich **nicht** überladen lassen: Elementauswahl **.**
  Bedingungsoperator **? :**
  Skopeoperator **::**

  Typ- bzw. Objektgröße **sizeof()** bzw. **sizeof**
  Elementauswahl mit Funktionszeiger **.***

Überladen von Operatoren

- **Neue Operatoren** können **nicht** definiert werden, z.B. ist ****** mit **operator**()** nicht erlaubt.

- Die **Operandenanzahl**, **Priorität** und **Assoziativität** eines Operators wird durch dessen Überladung **nicht verändert**.

- Die **Operatoren** ++ (Inkrement) und -- (Dekrement) können sowohl als **Präfix**- als auch als **Postfix - Operatoren** benutzt werden, die **Nutzung** ist in **C++** durch die **Signatur** festgelegt:

```
class day {                              //vgl. Inkrement.cpp
 public:  enum Tag { son, mon, die, mit, don, fre, sam };
 private: Tag t;
 public:  day(day::Tag t = sam):t(t){} //Konstruktor

          day &operator++(){     //Signatur Praefix-Operator
              t = (t == sam ) ? son : Tag(t+1); //Datenmember
              return *this;      //Referenz, keine Kopie
          }
          day operator++(int i){ //Signatur Postfix-Operator
              day dd(*this);     //Kopie von *this
              t = (t == sam ) ? son : Tag(t+1); //Datenmember
              return dd;         //Kopie von dd
          }
};
```

Überladen von Operatoren

- Sei **@** einer der o.g. **überladbaren Operatoren**, dann wird der **Operator @** allgemein durch die Funktion bzw. Methode `operator@(args`$^{0..}$`)` ersetzt.

- Die Funktion `operator@(args`$^{0..}$`)` hat so viele **Parameter** `args`$^{0..}$, wie der aktuell überladene Operator **Operanden** hat. D.h. **einstellige** (unäre) **Operatoren**, wie der **Indexoperator []** oder die **Vorzeichenoperatoren +** oder **-**, besitzen nur **einen Operanden**.

- **Zweistellige** (binäre) **Operatoren**, wie die **Operatoren +**, **-**, ***, / , besitzen **zwei Operanden**.

- **Mehrstellige Operatoren**, wie der **Aufrufoperator ()**, besitzen **beliebig viele Operanden**.

- Mindestens **einer** der Parameter muß ein **Klassenobjekt** sein. Beispielsweise kann die Addition von **int** - Größen mit dem Operator **+ nicht** überladen werden.

- Zwei Fälle sind zu unterscheiden:
  1. Falls `operator@(args`$^{0..}$`)` eine **Methode einer Klasse** ist, dann ist der **1. Operand** immer das **aktuelle Objekt** ***this** , welches **nicht** in die Parameterliste geschrieben wird.

  z.B. ist für `ratio &operator*(const ratio &x)` (s.o.) ***this** der **1. Operand** und der **1. Parameter x** der **2. Operand** : `r1->operator*(*r2) <==> *r1**r2`

Überladen von Operatoren

2. Falls **`operator@(args`$^{0..}$`)`** eine **Funktion** und **keine Memberfunktion** ist, dann ist der **1. Operand** der **1. Parameter**, der **2. Operand** der **2. Parameter**, u.s.w..

z.B. ist für **ratio operator*(const ratio &x, const ratio &y){ ratio z(x1); /* ... */ return r; }** der **1.Parameter x** der **1.Operand** und der **2.Parameter y** der **2.Operand**:
ratio r1(1 ,2); ratio r2(8, 4); **ratio r = r1 * r2;** steht für **r = operator*(r1, r2);**

z.B. ist für **`ostream &operator<<(ostream &cout, ratio &r)`** (s.o.)
der **1. Parameter cout** der **1. Operand** und der **2. Parameter r** der **2. Operand** :
`cout << r5` steht für **`operator<<( cout, r5)`** und umgekehrt.

Wenn die Funktion **`operator@(args`$^{0..}$`)`** auf **private** - oder **protected** - Member bzw. Methoden einer **Klasse** zugreifen muß, dann kann **`operator@(args`$^{0..}$`)`** entweder als **friend der Klasse** deklariert werden oder über **public** - Methoden der Klasse auf die **private** - Member zugreifen.

- Funktionen bzw. Methoden **`operator@(args`$^{0..}$`)`** sind mit **unterschiedlichen Signaturen** ebenfalls **überladbar**, z.B. ist **`operator-()`** (s.o.) in folgenden Deklarationen **überladen**:

```
ratio operator-(const ratio &x);
ratio operator-();
```

Überladen von Operatoren

- Das Überladen des **Indexoperators** mit **`int &operator[](size_t idx)`** ermöglicht die **Überprüfung** von **Vektor-Indexen** auf **Zulässigkeit**, ohne dass ein unzulässiger Index zum Abbruch führt. Außerdem ermöglicht der überladene Indexoperator als **public-Methode** den Zugriff auf **private-Vektoren** von außerhalb des Objektes über Objektnamen (vgl. **oper1.cpp**)

```
class intvec { size_t len;   // Anzahl Vektorelemente
               int *vec;     // int-Vektor, int ist durch anderen Typ ersetzbar
  public:
    intvec(size_t len=1):len(len), vec(len ? new int[len] : 0){
       for(size_t i=0; i<=len; (*this)[i++]=0); //i<=len , Fehler wird abgefangen
                    // bisher: vec[i++]=0       //Indexfehler mit Abbruch
    }
 /* const */ int &operator[](size_t idx) const { //Indexoperator

          static int dummy;                  //dummy ex. auch ausserhalb
          int *res = &dummy;                 //res als Adresse von dummy
          if(this){                          //this=0 moeglich
                   if(idx < len) res = vec+idx; //Adresse von vec[idx]
                   else cout<<"Index="<<idx<<" >= "<<len<<'\n';
          }
          else cout<<"Vektor existiert nicht\n";
          return *res;                       //vec[idx] oder dummy
    }
};                        // num.operator[](50)=50; // alternativ zu num[50]=50;
void main(){ intvec num(10); num[50]=50;} //Indexfehler, kein Abbruch, dummy=50
```

Überladen von Operatoren

- Falls **verhindert** werden soll, dass ein **Vektorelement** über den Zuweisungsoperator **verändert** wird, dann muß dieser Indexausdruck über **objekt.operator[](index)** bzw. **objekt[index]** formuliert werden und die Operatormethode mit **const** beginnen (s.o.)

- Der **Rückgabetyp** des Operatormethode hängt vom **Typ des Vektors** ab, hier **int**.

- Falls in der Klasse **mehrere Vektoren** existieren, dann kann es **nur eine** Operatormethode für einen der Vektoren geben .

- **Innerhalb der Methoden** einer Klasse sollte die Operatormethode ebenfalls für den Zugriff auf die Vektorelemente genutzt werden, genutzt wird hierbei ***this**, z.B. **(*this)[i]**

- **operator[]()** zum Überladen des Indexoperators **[]** kann nur eine **nicht-statische Methode** einer Klasse sein, **nicht** jedoch eine **Funktion**, d.h. auch **keine friend**-Funktion einer Klasse.

- Der **Rückgabetyp als Referenz** garantiert, daß eine **Zuweisung eines Ausdrucks** an die **Operatormethode** erfolgen kann, da diese eine **Referenz** (d.h. existierenden Speicherplatz) repräsentiert, z.B. ist **num[50] = 50;** äquivalent zu **num.operator[](50) = 50;**
  Wegen des **unzulässigen** Indexes **50** (s.o.) erfolgt hier indirekt die Zuweisung **dummy = 50;**

Überladen von Operatoren

- Im Falle der Nutzung von **Zeigern**, z.B. **intvec *pp = 0; (*pp)[0] = 5;** //**pp->operator[](0)**; sollte die **operator[]** - Methode vor dem Zugriff auf Datenmember **this != 0** testen (s.o.).

- Über den Indexoperator können auch Datentrukturen angesprochen werden, die **keine Vektoren** sind, ohne daß diese einen Index besitzen müssen. Z.B. können Elemente von **verketteten Listen** oder **Positionen einer Datei** angesprochen werden.

- Der **Typ des Indexes** von **operator[](Index)** ist **nicht beschränkt**, d.h. der Zugriff auf eine Datenstruktur mit der []-Notation kann z.B. auch über Zeichenketten erfolgen:

  int anzahl_studenten = Liste1["HTW"]; // äquivalent: **... = Liste1.operator[] ("HTW")**;

- Im Falle von **mehrdimensionalen Feldern** (Arrays, ...) kann der Indexoperator **nicht** mehr für den Zugriff verwendet werden, da er nur **ein Argument** besitzen darf, z.B. ist `int &operator[](size_t row, size_t col)` **nicht** zulässig.

- Für den Zugriff auf mehrdimensionaler Felder kann der **Aufrufoperator ()** überladen werden, der eine **Methode** sein muß, jedoch eine **beliebige Anzahl** an Argumenten `args`$^{0..}$ zuläßt:

Überladen von Operatoren

```
returntyp operator()(args0..);              // allgemein

int &operator()(size_t row, size_t col);  // konkret
```

- Der **Dereferenz-Operator** wird in der Form `typ *operator->(void)` definiert.

- Der Ausdruck `objekt->member` wird als `(objekt.operator->())->member` interpretiert (wenn `objekt` kein Zeigertyp ist).

- Der Ausdruck `objekt.operator->()` stellt entweder einen **Zeiger auf ein Objekt** dar, dann liefert der Gesamtausdruck die Komponente `member` dieses Objekts, oder es ist eine **Referenz auf ein Objekt**, dann wird die Methode `operator->()` dieses Objekts aufgerufen und der Vorgang **wiederholt** sich **rekursiv**.

- `operator->()` muß eine **Memberfunktion** sein. Falls er benutzt wird, muß sein **Rückgabetyp** ein **Zeiger** sein oder ein **Objekt** einer Klasse**,** auf die man `->` anwenden kann.

- Das Überladen des unären Operators `->` liefert einen effizienten Weg zur Nutzung von **Indirektion** in einem Programm, vgl. **zeigop.cpp**. Damit werden "intelligente" Zeiger bzw. "smart pointers" begründet.

Überladen von Operatoren

- Das **Überladen** der Operatoren << bzw. >> für eine **Klasse** ermöglicht die **benutzereigene Ausgabe** bzw. **Eingabe** von **Objekten** dieser Klasse, wie es bei den einfachen Datentypen der Fall ist (vgl. **ratio_cincout.cpp**):

```
ostream &operator<<(ostream &cout, const ratio &r){
     return cout<<r.getzaehl()<<" / "<<r.getnenn()<<'\n';
}

istream &operator>>(istream &cin, ratio &r){
     ratio rr(ratio::_int("zaehl = "), ratio::_int("nenn = "));
     r=rr; r.kuerze(); return cin;
}

void main(){  ratio *r1 = new ratio(8,4);
  cin>>*r1;
  operator>>(cin, *r1);                          //aequivalent

  cout<<"*r1 = "<<*r1;
  operator<<(operator<<(cout, "*r1 = "), *r1); //aequivalent
}
```

Überladen von Operatoren

- Da der **erste Operand** nicht das ein- oder auszugebende Objekt, sondern immer der **Datenstrom** ist, können die Operatorfunktionen `operator<<()` und `operator>>()` **keine Member-funktionen** sein. Sie müssen als **normale Funktionen** definiert werden.

- Um auf die **private**-Member des **übergebenen Objektes** (**2.Parameter**) zugreifen zu können, müssen die Funktionen entweder **friend** der Klasse sein oder **public**-Methoden nutzen.

- Die Nutzung in **Shift-Ketten** macht die **Rückgabe** von **ostream**- bzw. **istream**-Objekten als **Referenzen** notwendig. Der **erste Parameter** wird als **Referenz** übergeben und als **Referenz** zurückgegeben.

Überladen von Operatoren

```
      static void _(unsigned long *m){          // *m = Anzahl Zeichen in cin
         char c;                                // local
         if(cin.get(c)&&c!='\n')(*m)++,_(m);    // down
         cin.putback(c);                        // up
      }                                         // Beispiel: string2_better.cpp
};

ostream &operator<<(ostream &cout, zk &zkd){
   if(&zkd){
            char *s = zkd.get_s();
            size_t laenge = 0;
            if(s) laenge = strlen(s); else laenge=0;
            cout<<"\nLaenge(zkd) = "<<laenge<<'\n';
            cout<<"Inhalt(zkd) = "<<(s?s:"0")<<endl;
            delete [] s; s=0;
   }
   return cout;
}

istream &operator>>(istream &cin, zk &zkd){
   if(&zkd){
            unsigned long n=0UL;
            cout<<"\nEingabe zk.s = ";
            zk::_(&n);
            char *s = new char[n+1UL];

            cin.getline(s, n+1UL,'\n');
            zkd.set_s(s);
            delete [] s; s=0;
  }
  return cin;
}
```

Überladen von Operatoren

```
void main(){                          // Beispiel: string2_better.cpp
    zk *s2 = new zk("HTW");

    cin>>*s2;                         // Ueberladen operator>>()
    operator>>(cin, *s2);             // aequivalent

    cout<<*s2<<endl;                  // Ueberladen operator<<()
    operator<<(cout, *s2)<<endl; // aequivalent

    delete s2; s2=0;
}
```

Überladen von Operatoren

Objekt als Funktion (Funktoren)

Der **Aufrufoperator / Funktionsoperator ( )** kann durch die Operatorfunktion **operator( )( )** überladen werden. Ein Objekt kann dann wie eine Funktion aufgerufen werden.

Ein algorithmisches Objekt dieser Art wird **Funktionsobjekt** bzw. **Funktor** genannt. Funktoren sind Objekte, die sich wie Funktionen verhalten, aber alle Eigenschaften von Objekten haben.

```
#include <cmath>   //  sin(), Konstante M_PI für pi
#ifndef M_PI
 #define M_PI 3.14159265358979323846
#endif

class Sinus {
    public:
      enum Modus { bogenmass, grad, neugrad };
      Sinus(Modus m = bogenmass): berechnungsart(m) {}

      double operator()(double arg) const {
         double erg = 0.0;
         switch(berechnungsart) {
                  case bogenmass : erg = std::sin(arg);           break;
                  case grad      : erg = std::sin(arg/180.0*M_PI); break;
                  case neugrad   : erg = std::sin(arg/200.0*M_PI); break;
                  default   : ;
         }
         return erg;
      }
    private: Modus berechnungsart;
};
```

Überladen von Operatoren

```
#include<iostream>
#include"sinus.h"
using namespace std;

void sinusAnzeigen(double arg, const Sinus& funktor) {
    cout << "        Funktor   = "<<funktor(arg) << endl;
}

int main() {
    Sinus sinrad;
    Sinus sinGrad(Sinus::grad);
    Sinus sinNeuGrad(Sinus::neugrad);

    // Aufruf der Objekte wie eine Funktion
    cout << "sin(" << M_PI/4.0 <<" rad) = "         //  pi/4
         << sinrad(M_PI/4.0) << endl;
    cout << "sin(45 Grad)      = " << sinGrad(45.0) << endl;
    cout << "sin(45 Grad)      = " << sinGrad.operator()(45.0)
             << endl;
    cout << "sin(50 Neugrad)   = " << sinNeuGrad(50.0) << endl;

    // Übergabe eines Funktors an eine Funktion
    sinusAnzeigen(50.0, sinNeuGrad);

    cin.get();
}
/*
sin(0.785398 rad) = 0.707107
sin(45 Grad)      = 0.707107
sin(45 Grad)      = 0.707107
sin(50 Neugrad)   = 0.707107
        Funktor   = 0.707107
*/
```

Überladen von Operatoren

Funktoren können erzeugt, als Parameter übergeben oder in ihrem Zustand verändert werden. Die Zustandsänderung erlaubt einen flexiblen Einsatz, der mit Funktionen nur über zusätzliche Parameter möglich wäre. Funktionsobjekte sind flexibler als Funktionen, weil virtuelle Funktionen und Vererbung möglich sind.

Die Algorithmen der **Standardbibliothek** benutzen häufig **Funktoren**. Ein Beispiel hierfür ist **setprecision** zur Festlegung der Dezimalstellen bei der Ausgabe.

Überladen von Operatoren

Typumwandlungsoperator

C++ erlaubt die Definition von **Typumwandlungsoperatoren** für selbst definierte Klassen, um ein Objekt der Klasse in einen anderen Datentyp abzubilden. Typumwandlungsoperatoren sind **Methoden** der Klasse und haben **keine Parameterliste** und **keinen Rückgabetyp**:

operator *datentyp* **( ) const ;**

Die Methode muß einen Wert vom Typ "*datentyp*" zurückgeben. Methode wird sowohl bei der expliziten Konvertierung mittels cast als auch bei der impliziten Konvertierung gerufen:

```
class  ratio {
              int zaehl, nenn;  // Zaehler, Nenner
          public:
              operator float( ) const { return  float(zaehl) / nenn; }
              // ...
};

int main(){ ratio  rat(2, 3);
            float x = rat;            // implizite Konvertierung
            float y = float(rat)*2.5; // explizite Konvertierung
            // ...
}
```

Überladen von Operatoren

Überladen von new und delete

```
#include <iostream>                    // Beispiel newover2
class Objekt {                         // Objekt.h
public:
   virtual ~Objekt() {
      std::cout << "Objekt-Destruktor aufgerufen ("
                << this << ")" << std::endl;
   }
   static void *operator new(size_t size) {
      std::cout << "new aufgerufen. size=" << size << std::endl;
      return  ::operator new(size);
   }
   static void operator delete(void* ptr, size_t size) {
      std::cout << "delete aufgerufen. size="  << size << std::endl;
      ::operator delete(ptr);
   }
   static void *operator new[](size_t size) {
      std::cout << "new[] aufgerufen. size=" << size << std::endl;
      return  ::operator new[](size);
   }
   static void operator delete[](void* ptr, size_t size) {
      std::cout << "delete[] aufgerufen. size="  << size << std::endl;
      ::operator delete[](ptr);
   }
};
```

Überladen von Operatoren

```
#include <iostream>
#include <string>
#include "Objekt.h"
using namespace std;

class Person : public Objekt {
   public:
     Person(const string& n = "HTW") : name(n) {}

     ~Person() { cout << "Person-Destruktor aufgerufen (" << name << ")" << endl; }

     const string& getName() const { return name; }

   private:   string name;
};

int main() {
   Person person("Lena");                          // Stack-Objekt
   cout << "Name : " << person.getName() << endl;
   Person* ptr1 = new Person("Jens");              // Heap-Objekt
   cout << "Name : " << ptr1->getName() << endl;
   delete ptr1;                                    // Löschen des Heap-Objekts

   size_t anz = 2;
   Person* arr = new Person[anz];                  // dynamisches Array anlegen
   for(size_t i = 0; i < anz; ++i) {
      cout << i << ": " << arr[i].getName() << endl;
   }
   delete[] arr;                                   // dynamisches Array löschen
}
```

Überladen von Operatoren

Bemerkungen:

- Destruktor ist **virtual**

- alle Methoden sind **static**, kann weggelassen werden, **new** und **delete** sind generell **static**

- bei den **delete**-Funktionen kann 2.Parameter **size** weggelassen werden

- der Standardoperator wird über den Scope-Operator :: realisiert

- Compiler wandelt die Anweisung **new Person("Jens");** in
  Objekt::operator new (sizeof (Person)) um

- Compiler wandelt die Anweisung **new Person [anz];** in
  Objekt::operator new[ ] (anz * sizeof (Person) + x) um

- **x** ist eine implementationsabhängige Größe, um Verwaltungsinformationen abzulegen