Überladen von Operatoren

Einzelne Operatoren sind überladen, d.h. die Semantik (was tut der Operator) ist von den Typen der Operanden abhängig. So ist „/“ einmal Integerdivision, zum anderen Realdivision.

Dieser Teil zeigt, wie Operatorsymbole mittels selbst programmiertem Code mit neuer Bedeutung versehen werden können.

Inhaltsverzeichnis

1 Überblick....... 2
2 Arithmetische Operatoren....... 6
3 Vergleichsoperatoren....... 11
1.Welche Ausgabe erzeugt das o.a. Programm?.......11
2.Wie ist operator== zu ändern, so dass der Vergleich von und true liefert?.......11
4 Ausgabeoperator.......12
5 Indexoperator....... 14
6 Zuweisungsoperator....... 24
7 Inkrementoperator....... 27
8 Typumwandlungsoperator.......34
9 Smart Pointer....... 36
10 Objekte als Funktionen (Funktoren)....... 50

1 Überblick

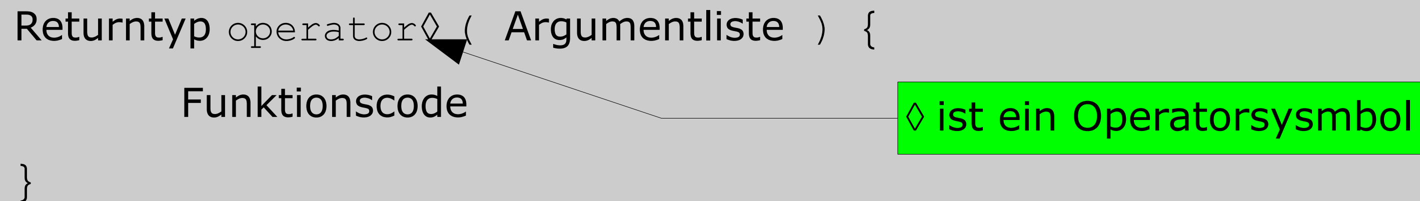
Überladen von Operatoren ist prinzipiell nichts anderes als **Überladen** von **Funktionen**.

Es ist allerdings **nur auf Klassen** anwendbar, also nicht direkt auf elementare Datentypen, wie `int` oder `double`.

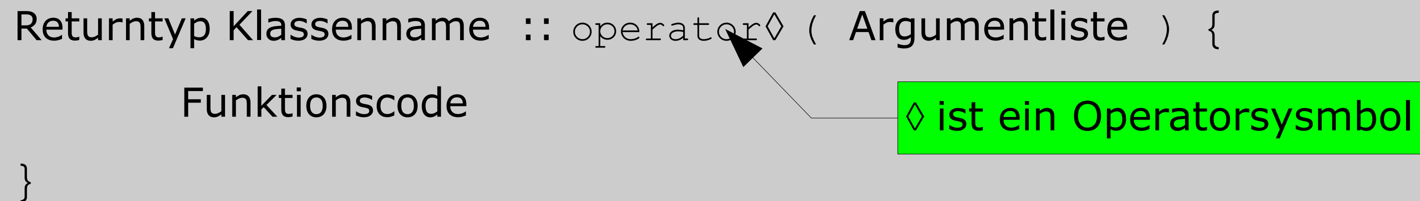
Ziel ist es, besser lesbaren Code zu erzeugen, indem man z.B. Operatoren wie „*" und „<<" für die Multiplikation und Ausgabe rationaler Zahlen definiert und dann etwa folgenden Schreibweise verwenden kann:

```
rational a, b, c;
a = new rational(2,3); // 2/3
b = new rational(3,2);
c = a * b;
cout << c;
```

In C++ wird das Überladen eines Operators für eine **globale Funktion** wie folgt definiert:



Die Definition für das Überladen einer **Elementfunktion** erfolgt nach folgendem Schema:



Ein selbst definierter Operator ist eigentlich nicht anderes als eine andere Schreibweise für eine Funktion. Der **Compiler wandelt** die **Operatorschreibweise** stets in einen **Funktionsaufruf**.

Für einige Operatoren steht nur die Umwandlung in eine Elementfunktion in C++ zur Verfügung.

Folgende Tabelle zeigt die Verwendung und die Umwandlung durch den Compiler:

Syntax	Ersetzung (Compiler) durch	Elementfunktion
`x ◊ y`	`operator◊(x,y)`	nein
`◊ x`	`operator◊(x)`	nein
`x ◊`	`operator◊(x,0)`	nein
`x ◊ y`	`x.operator◊(x)`	ja
`◊ x`	`x.operator◊()`	ja
`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->())->`	ja

Operatoren, wie ++ oder -- werden in Postfix- und Präfixnotation unterstützt!

Folgende Einschränkungen müssen in C++ bzgl. Operatorfunktionen beachtet werden:

- die Operatoren „.“, „.*“ „::“ und „?:“ können **nicht** überschrieben werden;
- die vorgegebenen Vorrangregel können **nicht** verändert werden;
- mindestens ein Argument der Operatorfunktion muss eine Klasse sein.

Beispiel (string concat):

```
$ cat concat.cpp
#include <iostream>
#include <string>
using namespace std;

string operator| (string& a, const string& b) {
        return a.append(b);
}

string operator| (string& a, char* b) {
        return a.append(b);
}

int main() {
        string s1 = "Jennifer";
        string s2 = " Lopez";
        char cs[] = ", wow!";
        string r = s1 | s2;
        r = r | cs;
        cout << r << endl;
        return 0;
}
$
```

neuer Operator |

Compiler erzeugt
r = operator| (s1, s2)

Einige Besonderheiten werden im Folgenden am Beispiel von rationalen Zahlen und Vektoren erörtert.

2 Arithmetische Operatoren

Zunächst soll ein Operator für die **Addition** rationaler Zahlen **realisiert** werden.

Bevor man einen Operator implementiert, sollte man die **Eigenschaften** des Operators überlegen:

- + soll kommutativ sein, d.h. a+b == b+a
- die Addition mit einer natürlichen Zahl soll möglich sein (a+3).

Die Verwendung soll also wie folgt sein:

```
$ cat Version1/main.cpp
#include"ratioop.h"
#include"ratioop.cpp"
#include<iostream>
using namespace std;
int main() {
     rational a,b,c;
     a.definiere(2,3);
     b.definiere(1,3);

     c = a+b+1;

     cout << "c=" << c.Zaehler() << "/" << c.Nenner() << endl;
}
$
```

Zunächst die Prototypen und der Einfachheit halber einige Inline-Funktionen:

```
$ cat ratioop.h
// Klasse für rationale Zahlen mit überladenen Operatoren
#ifndef ratioop_h
#define ratioop_h ratioop_h
#include<iostream>

class rational {
   public:
     rational();
     rational(long);              // Typumwandlungskonstruktor

     long Zaehler() const;       // Abfragen
     long Nenner()  const;

     rational& operator+=(const rational&); // arithmetische Methoden
     // weitere arithmetische Methoden weggelassen ... -> Hoersaalübung

     void definiere(long zaehler, long nenner); // weitere Methoden
     void kehrwert();
     void kuerzen();

   private:
     long zaehler, nenner;
};
```

```
// inline Methoden
inline rational::rational()                    // Konstruktor
: zaehler(1), nenner(1)    {
}

inline rational::rational(long ganzeZahl)
: zaehler(ganzeZahl), nenner(1)    {
}

inline long rational::Zaehler() const {
      return zaehler;
}
inline long rational::Nenner()  const {
      return nenner;
}

// globaler Operator
rational operator+(const rational&, const rational&);
#endif         // ratioop.h
```

Die Implementierung verwendet eine Elementfunktion, um damit einen globalen Operator definieren zu können:

```
$ cat Version1/ratioop.cpp
#include"ratioop.h"
#include<cassert>

// globale Hilfsfunktion (größter gemeinsamer Teiler)
long ggt(long x, long y) {
     while (y) { long rest = x % y;
       x=y; y=rest;
     }
     return x;
}
// ******+ Methodenimplementation ********
void rational::definiere(long z, long n) {
     zaehler = z;
     nenner  = n;
     assert(nenner != 0);
     kuerzen();
}
void rational::kehrwert() {
     long temp = zaehler;
     zaehler = nenner;
     nenner  = temp;
     assert(nenner != 0);
}
void rational::kuerzen() {
     int sign = 1;
     if (zaehler < 0) { sign=-sign; zaehler = -zaehler;}
     if (nenner  < 0) { sign=-sign; nenner  = -nenner;}
     long teiler=ggt(zaehler, nenner);
```

```
    zaehler = sign*zaehler/teiler;
    nenner = nenner/teiler;
}

rational& rational::operator+=(const rational& b) {  // Elementfunktion
    zaehler = zaehler*b.nenner + b.zaehler*nenner;
    nenner  = nenner*b.nenner;
    kuerzen();
    return *this;
}

// ******* globaler Operator ********
rational operator+(const rational& a, const rational& b) {
       rational tmp = a;
       return tmp += b;   // tmp.operator+=(b)
}
$
```

Hier könnte man auch den Kopierkonstruktor verwenden und schreiben:

```
rational operator+(const rational& a,
const rational& b) {
    return rational(a) += b;
}
```

Wie würde die Implementierung für -=, *=, /= aussehen?

3 Vergleichsoperatoren

Ein weiteres Beispiel ist die Realisierung eines **Operator**, der auf **Gleichheit** testen kann:

```
$ cat Version2/*
int main() {
     rational a,b,c;
     a.definiere(2,4);
     b.definiere(1,2);

     if (a == b)
             cout << "a=b" << endl;
     else
             cout << "a!=b" << endl;
}
/ ******* globale Operatoren ********
bool operator==(const rational& a, const rational& b) {
        return   a.Zaehler() == b.Zaehler() && a.Nenner()  == b.Nenner();
}
$
```

Fragen:

1. Welche Ausgabe erzeugt das o.a. Programm?
2. Wie ist **`operator==`** zu ändern, so dass der Vergleich von $\frac{2}{4}$ und $\frac{1}{2}$ true liefert?

4 Ausgabeoperator

Ziel ist es, den Operator << zu überladen, um etwa

```
rational a;
...
cout << a << endl;
```

schreiben zu können.

Da wir die Klasse `ostream` **nicht ändern** können (**wieso eigentlich nicht?**), kann `<<` also **keine Elementfunktion** von `ostream` sein.

Wir implementieren deshalb `<<` als **globale Funktion** mit 2 Argumenten. Um **Verkettungen** zu ermöglichen, soll `<<` eine Referenz auf sein Argument zurück geben:

```
ostream& operator<<(ostream& s, const rationale&)
```

Rückgabe ist Referenz auf s

Somit ist es möglich, zu schreiben:

```
cout << "a=" << a << "b=" << b;
```

Entspricht: `(((cout<<"a=")<<a)<<"b=")<<b;`

Die Implementierung von << ist dann einfach:

```
$ cat Version3/*
std::ostream& operator<< (std::ostream& ausgabe, const rational& r) {
     ausgabe << r.Zaehler() << "/" << r.Nenner();
     return ausgabe;
}
...
$
```

Hörsaalübung

Erweitern Sie die Klasse rational, so dass alle vier Grundrechenarten möglich sind.

5 Indexoperator

Frage:

Was druckt folgendes Programm (Indexfehler/indexFehler.cpp)?:

```
#include <iostream>
using namespace std;

int b[1];
int a;
int main() {
  a = 1;
  b[0] = 2;
  cout << b[1] << endl;
  cout << b[-1000000] << endl;
}
```

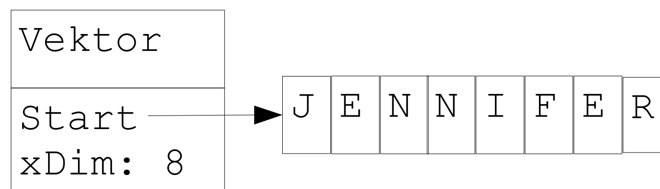
Der Indexoperator ist also nicht sicher.

Der Indexoperator wird an einer Klasse für Vektoren gezeigt, die folgende Unterschiede zu C-Arrays hat:

- Der Zugriff über den Indexoperator ist **sicher**, d.h. es wird eine Fehlermeldung zur Laufzeit erzeugt, wenn eine Indexüberschreitung statt findet.
- Der Vektor ist **dynamisch**, d.h. es gibt keine statische Obergrenze für die Elementanzahl.
- Wir verwenden **Klassentemplates**, um Vektoren beliebigen Typs verfügbar zu haben.

Die Klasse greift auf die Vektorelemente über einen Zeiger zu, ein protected Element speichert die jeweilige Größe des Vektors.

Ein Objekt der Klasse für char wird also wie folgt aussehen:



```
$ cat indexop/vektor.t
// dynamische Vektor-Klasse
#ifndef vektor_t
#define vektor_t vektor_t
#include<cassert>

template<class T>
class Vektor {
  public:
    Vektor(int);                                    // Default-Konstruktor
    Vektor(const Vektor<T>& v);                     // Kopierkonstruktor
    virtual ~Vektor() { delete [] start;}           // Destruktor

    int  Groesse() const {
        return xDim;
    }
    void GroesseAendern(int);                       // dynamisch ändern

    // Indexoperator
    T& operator[](int index) {
         assert(index >= 0  &&  index < xDim);
         return start[index];
    }

    // Indexoperator für konstante Vektoren
    const T&  operator[](int index) const {
         assert(index >= 0  &&  index < xDim);
         return start[index];
    }
```

```
    // Initialisierung des Vektors
    void init(const T&);

    // Zeiger auf Anfang und Position nach dem Ende
    // für nicht-konstante und konstante Vektoren
    T* begin() {
        return start;
    }

    T* end() {
        return start + xDim;
    }

    const T* begin() const {
        return start;
    }

    const T* end() const {
        return start + xDim;
    }

  protected:   // Nicht private wegen Vererbung
    int  xDim;                                  // Anzahl der Datenobjekte
    T    *start;                                // Zeiger auf Datenobjekte
};
```

```
/********** Implementierung ****************/

template<class T>
inline Vektor<T>::Vektor(int x=0)                    // Konstruktor
: xDim(x), start(new T[x]) {
}

template<class T>
inline Vektor<T>::Vektor(const Vektor<T> &v)   { // Kopierkonstruktor
    xDim  = v.xDim;
    start = new T[xDim];                          // Platz beschaffen
    for (int i = 0; i < xDim; ++i)                // Komponentneweise kopieren
        start[i] = v.start[i];
}

template<class T>
inline void Vektor<T>::init(const T& wert) {
     for (int i = 0; i < xDim; ++i)
         start[i]=wert;
}
```

```
template<class T>
inline void Vektor<T>::GroesseAendern(int neueGroesse) {
    // neuen Speicherplatz besorgen
    T *pTemp = new T[neueGroesse];

    // die richtige Anzahl von Elementen kopieren
    int kleinereZahl = (neueGroesse > xDim) ? xDim : neueGroesse;
    for (int i = 0; i < kleinereZahl; ++i)
        pTemp[i] = start[i];

    // alten Speicherplatz freigeben
    delete [] start;

    // Verwaltungsdaten aktualisieren
    start = pTemp;
    xDim  = neueGroesse;
}
#endif // vektor_t
$
```

Damit kann man die Klasse etwa wie folgt verwenden:

```
$ cat indexop/main.cpp
// Beispiel zur Vektor-Klasse
#include <iostream>
using namespace std;
```

```
#define PRINT(X) cout << (#X) << " = " << (X) << endl
#include "vektor.t"

// globaler Operator für int-Vektoren
ostream& operator<<(ostream& os, const Vektor<int>& v) {
    // Verbesserungen im Ausgabelayout sind möglich!
    for(int i = 0; i < v.Groesse(); ++i) {
        os << v[i] << '\t';
        if ((i+1)%8==0 || i==v.Groesse()-1)
            os << endl;
    }
    return os;
}

int main() {
     Vektor<int> v(3);      // Vektor mit 3 int Elementen
     v.init(7);
     PRINT(v);
     PRINT(v.Groesse());
     v[0]=1000;             // Indexoperator verwenden
     PRINT(v);
```

```
    const int anz=16;
    cout << "Vektor bedarfsweise vergrößern:\n";
    for (int i=0; i<anz; ++i) {
        if (i==v.Groesse()) {   // Platz verbraucht
            v.GroesseAendern(i+10);
            cout << "neu: "; PRINT(v.Groesse());
        }
        v[i]= i*i;
    }
}
$
```

Wie wäre die Klasse Vektor zu ändern, damit sie sich selbst organisert und somit die o.a. Abfrage (`if (i==v.Groesse()`) nicht erforderlich wäre?

Bei der Deklaration des Indexoperators ist der Rückgabetyp eine **Referenz** auf T.

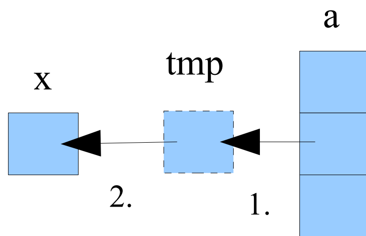
Wieso muss das so sein, d.h. was passiert, wenn von `operator[]` keine Referenz zurückgegeben wird?

Dies ist erforderlich, da ansonsten eine temporäre Kopie verwendet werden würde und das Ergebnis fehlerhaft wäre (der Compiler würde aber schon einen Fehler melden, wenn eine Anweisung der Form „`a[i] = 123`“ verwendet werden würde).

Annahme:

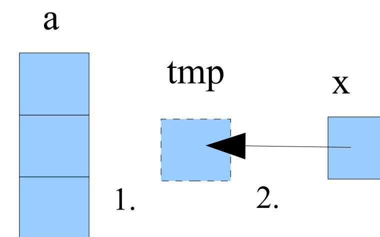
```
T operator[](int index) {     // Rückgabe also keine Referenz
        assert(index >= 0  &&  index < xDim);
        return start[index];
}
```

x = a[1]



a[1], also der Zuweisungsop. liefert eine Kopie tmp. Der Wert wird an x zugewiesen.
Also alles ok !

a[1] = x

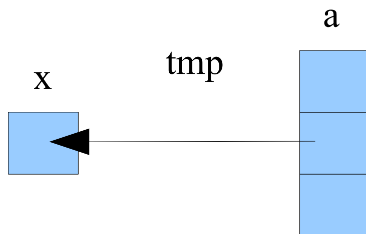


a[1], also der Zuweisungsop. liefert eine Kopie tmp. Dieser Kopie wird x zugewisen, dann wird die Kopie gelöscht, der Wert von a[1] bleibt also **unverändert**.
Also **nicht** ok !

In unserer Implementierung also:

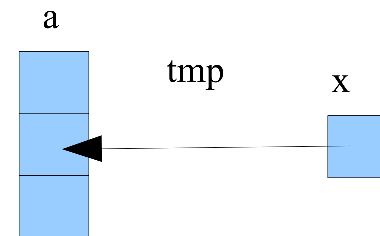
```
T& operator[](int index) {     // Rückgabe also keine Referenz
         assert(index >= 0  &&  index < xDim);
         return start[index];
}
```

x = a[1]



a[1], also der Zuweisungsop. liefert eine Referenz. Der Wert wird an x zugewiesen.
Also alles ok !

a[1] = x



a[1], also der Zuweisungsop. liefert eine Referenz, der x zugewiesen wird.
Also alles ok !

6 Zuweisungsoperator

Vektoren will man oft als Ganzes behandeln, d.h. auch zuweisen. Ist ein spezieller Zuweisungsoperator nicht definiert, so wird der Compiler **automatisch** einen Zuweisungsoperator erzeugen, der ein Objekt elementweise kopiert. Dabei wird für jedes Element selbst der Zuweisungsoperator für die Klasse, zu der das Element gehört verwendet.

- Der Zuweisungsoperator für Grunddatentypen (int, ...) kopiert dabei bitweise.
- Enthält eine Klasse **Konstanten**, wird **nicht kopiert**, da Konstanten nur initialisiert werden können (im Konstruktor).
- **Normale Operatorfunktionen** werden wie normale Funktion **vererbt**.
- Der **Zuweisungsoperator** wird allerdings **nicht vererbt**, damit wird verhindert, dass die zusätzlichen Datenelemente abgeleiteter Klassen bei der Zuweisung nicht „vergessen" werden.

=> In der abgeleiteten Klasse ist der Zuweisungsoperator zu überschreiben !!!!

In der Beispielklasse `vektor` ist ein eigener Zuweisungsoperator realisiert:

```
$ cat zuweisungsop/vektor.t
...
template<class T>
inline Vektor<T> &Vektor<T>::operator=(const Vektor<T> &v) {
    if (this != &v) {                        // Zuweisung identischer Objekte vermeiden
        T *temp = new T[v.xDim];      // neuen Platz beschaffen
        for (int i = 0; i < v.xDim; ++i)
            temp[i] = v.start[i];
        delete[] start;                      // Speicherplatz freigeben
        xDim  = v.xDim;
        start = temp;
    }
    return *this;
}
...
$
```

```
$ cat  zuweisungsop/main.cpp
int main()
{    Vektor<int> v1(3), v2(2);
     v1.init(7);
     PRINT(v1);
     PRINT(v2);

     v2 = v1;
     PRINT(v2);

     return 0;
}
$
```

7 Inkrementoperator

Der Inkrementoperator wird am Beispiel der Klasse datum verdeutlicht. Dabei soll das Inkrementieren Monatsübergänge und Schaltjahre berücksichtigen.

```
$ cat datum/main.cpp
int main() {
  Datum heute, d1, d2;
  cout << "d1:   " << heute << endl;
  d1 = heute;
  d2 = ++d1;                           // entspricht d2.operator++()
  cout << "++d1: " << d2 << endl;

  d1 = heute;
  d2 = d1++;                           // entspricht d2.operator++(0)
  cout << "d1++: " << d2 << endl;
}
$
```

Die Implementierung verwendet Standardfunktionen zur Ermittlung der Systemzeit.

```
$ cat datum/datum.h
#ifndef datum_h
#define datum_h datum_h
#include<string>
```

```
#include<iostream> //  für operator<<()
class Datum {
  public:
    Datum();                            // Default-Konstruktor
    Datum(int t, int m, int j=1997); // allgemeiner Konstruktor
    void setzeDatum(int t, int m, int j); // Datum setzen
    void aktuell();                     // Systemdatum setzen
    bool Schaltjahr() const;
    Datum& operator++();                // Tag hochzählen, präfix
    Datum operator++(int);              // Tag hochzählen, postfix
    int Tag()   const;
    int Monat() const;
    int Jahr()  const;
  private:
     int tag, monat, jahr;
};

bool Schaltjahr(int jahr);

// inline-Methoden
inline Datum::Datum(int t, int m, int j) { setzeDatum(t, m, j);}

inline int Datum::Tag()   const { return tag;  }
inline int Datum::Monat() const { return monat;}
inline int Datum::Jahr()  const { return jahr; }
```

```
inline bool Datum::Schaltjahr() const {
  return ::Schaltjahr(jahr);
}

// global
bool korrektesDatum(int t, int m, int j);

inline bool Schaltjahr(int jahr)  {
     return  jahr % 4==0 && jahr % 100 || jahr % 400==0;
}

int DatumDifferenz(const Datum& , const Datum&);
#endif   // datum_h
$
```

Die Implementierung der Inkrementoperatoren unterscheidet Präfix- und Postfixnotation.

Die Realisierung der **Postfixoperation** geschieht nach dem Schema:

- **Merken** des **aktuellen Zustand**s des Objektes
- Objekt **inkrementieren** und damit verändern
- **Rückgabe** des **gemerkten** Zustandes

```
$ cat datum/datum.cpp
#include"datum.h"
```

```
#include<ctime>
#include<cstdlib>
#include<cassert>
#include<limits>   // fuer maxDatum()

Datum::Datum()  { aktuell();}

void Datum::setzeDatum(int t, int m, int j) {
    tag = t;
    monat = m;
    jahr = j;
    if (jahr < 100)
       jahr += 1900;
    assert(korrektesDatum(tag, monat, jahr));
}

void Datum::aktuell() {   // Systemdatum eintragen
    // time_t, time(), tm, localtime() sind in <ctime> deklariert
    time_t now = time(NULL);
    tm *z = localtime(&now);           // Zeiger auf struct tm
    jahr  = z->tm_year + 1900;
    monat = z->tm_mon+1;               // localtime liefert 0..11
    tag   = z->tm_mday;
}
```

```
Datum& Datum::operator++() {      // Datum um 1 Tag erhoehen, Praefix ++d
   ++tag;
   // Monatsende erreicht?
   if(!korrektesDatum(tag, monat, jahr)) {
     tag = 1;
     if (++monat > 12) {
        monat = 1;
        ++jahr;
     }
   }
   return *this;
}
```

```
Datum Datum::operator++(int) {         // Datum um 1 Tag erhoehen, Postfix d++
   Datum temp = *this;
   ++*this;                            // Praefix ++ aufrufen
   return temp;
}
```

```
// globale Funktionen + Operatoren
bool korrektesDatum(int t, int m, int j) {
     // Tage pro Monat(static vermeidet Neuinitialisierung):
     static int tmp[]={31,28,31,30,31,30,31,31,30,31,30,31};
     if(Schaltjahr(j))
         tmp[1]=29;

     return (   m >= 1    && m <= 12 && j  >= 1583 && j  <= 2399  // oder mehr
                && t  >= 1    && t   <= tmp[m-1]);
}

std::ostream& operator<<(std::ostream& os, const Datum& d) {
   os << d.Tag() << '.' << d.Monat() << '.' << d.Jahr();
   return os;
}
```

Hörsaalübung

Erweitern Sie das Beispiel mit der Klasse `datum` um globale Operatoren (=, !=, <) zur Abfrage, ob zwei datum-Objekte gleich, ungleich oder kleiner sind und etwa folgendes Programm möglich ist:

```
int main() {
  Datum heute, d1, d2;
  d1 = heute;
  d2 = d1++;

  if (d1 < d2) {
        cout << d1 << " < " << d2 << endl;
  } else if (d1 == d2) {
        cout << d1 << " = " << d2 << endl;
  } else {
        cout << d1 << " > " << d2 << endl;
  }
}
```

8 Typumwandlungsoperator

Der C++ Compiler führt für Grunddatentypen automatisch eine Typumwandlung durch, wenn unterschiedlich getypte Operanden in einem Ausdruck auftreten (vgl. ProgrammierenI). Auch kann man als Programmierer den cast-Operator dazu verwenden.

Beispiel:

```
int i = 18;
float f = 1.23;
f = f + i;                  // implizite Typumwandlung durch Compiler
f = f + (float) 2;          // explizite Typumwandlung durch Programmierer
```

Für selbst entwickelte Klassen kann man einen Typumwandlungsoperator wie folgt definieren:

`operator` Datentyp `();`

Beispiel (Umwandlung von `date` in `string`):

```
$ cat typumwandlungsoperator/main.cpp
int main() {
  Datum d(1,1,2000);
  string str = d;                 // Typumwandlung implizit: datum -> string
  cout << str << endl;
  string str1 = (string) d;       // Typumwandlung explizit: datum -> string
  cout << str1 << endl;
}
$
```

Die Änderungen der Klasse `date` sind:

```
$ cat typumwandlungsoperator/date.cpp
...
datum::operator string() {        // Typumwandlung date->string
    std::string temp("tt.mm.jjjj");
    temp[0] = char(tag/10)+'0';
    temp[1] = char(tag%10)+'0';
    temp[3] = char(monat/10)+'0';
    temp[4] = char(monat%10)+'0';
    int pos = 9;                     // letzte Jahresziffer
    int j = jahr;
    while(j > 0) {
       temp[pos] = j % 10 + '0';   // letzte Ziffer
       j = j/10;                     // letzte Ziffer abtrennen
       --pos;
    }
    return temp;
}
...
$
```

9 Smart Pointer

Operatoren zum Zugriff auf Objekte, auf die ein Zeiger verweist können ebenfalls überladen werden. Damit sind „**sichere Zeiger**“ realisierbar, d.h. die Nachteile der Verwendung von Zeiger sollen vermieden werden.

Dereferenzieren von **nicht** initialisierten Zeigern:

```
$ cat dereferenzieren.cpp
#include <iostream>
using namespace std;
class I {
        public:
                I() {
                        data = 0;
                };
                int getData() {
                        return data;
                };
                void setData(int x) {
                        data = x;
                };
        private:
                int data;
};

int main() {
        I d;
        I *p;
        cout << "d: "  <<  d.getData() << endl;
        cout << "*p: " << p->getData() << endl; // Fehler!
        return 0;
}
$
```

delete auf Objekt anwenden, dass **nicht** mit **new** erzeugt wurde:

```
$ cat deleteOhneNew.cpp
class I {
        public:
                I() {
                        data = 0;
                };
                int getData() {
                        return data;
                };
                void setData(int x) {
                        data = x;
                };
        private:
                int data;
};

int main() {
        I d;
        I *p;
        p = &d;
        cout << "d: "  <<  d.getData() << endl;
        cout << "*p: " << p->getData() << endl;
        delete p; // Fehler!
        return 0;
}
$
```

delete mehr als **einmal** auf das selbe Objekt anwenden:

```
$ cat deleteMehrmals.cpp
class I {
        public:
                I() {
                        data = 0;
                };
                int getData() {
                        return data;
                };
                void setData(int x) {
                        data = x;
                };
        private:
                int data;
};

int main() {
        I *p = new I;
        cout << "*p: " << p->getData() << endl;
        delete p;
        delete p; // Fehler!
        return 0;
}
$
```

hängende Zeiger (engl. dangling pointer):

```
$ cat haengendePointer.cpp
class I {
        public:
                I() {
                        data = 0;
                };
                int getData() {
                        return data;
                };
                void setData(int x) {
                        data = x;
                };
        private:
                int data;
};

int main() {
        I *p = new I;
        I *q;
        q = p;
        cout << "*p: " << p->getData() << endl;
        delete p;
        cout << "*q: " << q->getData() << endl;   // Fehler!
        return 0;
}
$
```

verwitwete Objekte:

```
$ cat verwitweteObjekte.cpp
class I {
        public:
                I() {data = 0;};
                int getData() {
                        return data;
                };
                void setData(int x) {
                        data = x;
                };
        private:
                int data;
};
int main() {
        I *p = new I;
        I *q = new I;
        p->setData(1);
        cout << "*p: " << p->getData() << endl;
        cout << "*q: " << q->getData() << endl;
        q = p;  // Fehler: verwitwetes Objekt (*q) entstanden
        cout << "*p: " << p->getData() << endl;
        cout << "*q: " << q->getData() << endl;
        return 0;
}
$
```

Die o.a. Nachteile können durch Smart Pointer beseitigt werden. Ein Smart Pointer soll folgende Eigenschaften haben:

- Die **Verwendung** von Smart Pointer soll **analog** zu normalen Pointern sein (-〉 Überladen von ->, *).
- Ein Smart Pointer soll für **verschiedene Klassen** möglich sein (-〉 Templates).
- Smart Pointer sind **stets initialisiert**, entweder mit einem Objekt oder sie zeigen auf NULL.
- Dereferenzieren von **nicht intialisierten** Objekten führt nicht zu Programmabsturz.
- Es existieren **keine verwitweten** Objekte, weil der Smart Pointer das Objekt löscht, wenn er nicht mehr auf es zeigt.
- Smart Pointer sind **nur** für **Heap**-Objekte (mit new erzeugt) definiert.
- Smart Pointer für **Grunddatentypen** werden **nicht** unterstützt.

Die Klasse `smartPointer` soll die o.a. Forderungen erfüllen. Sie verwendet eine Variable `ZeigerAufObjekt`, die auf einen beliebigen Typ verweisen kann und Methoden, die die Zusatzfunktionalität umsetzen.

```
$ more smartptr.t
#ifndef smartptr_t
#define smartptr_t smartptr_t
#include<cassert>
using namespace std;
```

```
template <class T>
class smartPointer {
   public:
      smartPointer(T *p);                    // Konstruktor
      virtual ~smartPointer();               // Destruktor
      T* operator->() const;                 // Derefernzierungsoperator
      T& operator*() const;                  // Derefernzierungsoperator
      smartPointer& operator=(T *p);         // Zuweisungsoperator
      operator bool() const;                 // Vergleichsoperator
      void loescheObjekt();
   private:
      T* ZeigerAufObjekt;                        // eigentlicher Zeiger
      void operator=(const smartPointer& ); // p1 = p2;  verbieten
      smartPointer(const smartPointer&);     // Initialisierung mit
                                             // smartPointer verbieten
};

template <class T>
inline smartPointer<T>::smartPointer(T *p=0)    // nur initialisierte Zeiger,
                                                // mit 0 oder wenn Objekte
                                                // übergeben mit dem Objekt
: ZeigerAufObjekt(p){
}

template <class T>
inline smartPointer<T>::~smartPointer() {
   loescheObjekt();        // Destruktor löscht Objekt und stellt sicher,
                           // dass Zeiger danach mit 0 initialisiert ist
}
```

```
// Die Operatoren -> und * prüfen, ob sie auf ein Objekt verweisen, wenn nicht
// wird Fehlermeldung ausgegeben
template <class T>
inline T* smartPointer<T>::operator->() const {
    assert(ZeigerAufObjekt);
    return ZeigerAufObjekt;
}
template <class T>
inline T& smartPointer<T>::operator*() const {
    assert(ZeigerAufObjekt);
    return *ZeigerAufObjekt;
}

// Der Zuweisungsoperator weist einem smartPointer die Adresse eines Objektes
// vom Typ T zu.
template <class T>
inline smartPointer<T>& smartPointer<T>::operator=(T* p) {
    assert(ZeigerAufObjekt == 0); // stellt sicher, dass nur ein freier
                                  // Zeiger verwendet wird, es kann also keine
                                  // verwitweten Objekte geben.
    ZeigerAufObjekt = p;
    return *this;
}
```

```
// Mit Hilfe des Typumwandlungsoperator wird ein Vergleichtsoperator
// definiert. Damit sind Abfragen möglich, die Testen ob ein smartPointer
// auf ein Objekt zeigt oder nicht.
template <class T>
inline smartPointer<T>::operator bool() const {
   return bool(ZeigerAufObjekt);
}

template <class T>
inline void smartPointer<T>::loescheObjekt() {
    delete ZeigerAufObjekt;
    ZeigerAufObjekt = 0;
}
#endif    // smartptr_t
```

Wir wollen **verhindern**, dass **mehrere Zeiger auf das selbe Objekt** zeigen. Dies könnte erreicht werden, indem man die Verweise mit zählt (engl. **reference counting**). Hier ist der einfache Weg gewählt, den Zuweisungsoperator, den Initialsierungs -bzw den Kopierkonstruktor **private zu definieren**.

Damit könen wir die Verwendung der `smartPointer` innerhalb main demonstrieren. Es werden zwei Klassen `A` und `B` verwendet, wobei `B` von `A` erbt:

```
$ cat main.cpp
#include"smartptr.t"
#include<iostream>
using namespace std;

class A {
    public:
      virtual void hi() {
           cout << "hier ist A::hi()" << endl;
      }

      virtual ~A() {
           cout << "A::Destruktor" << endl;
      }
};

class B : public A {
    public:
      virtual void hi() {
           cout << "hier ist B::hi()" << endl;
      }

      virtual ~B() {
           cout << "B::Destruktor" <<endl;
      }
};
```

```
int main() {
    cout << "Zeiger auf dynamische Objekte:" << endl;
    cout << "Konstruktoraufruf" << endl;
    smartPointer<A> spA(new A);
    cout << "... oder Zuweisung" << endl;
    smartPointer<B> spB;
    spB = new B;

    cout << "Operator ->" << endl;
    spA->hi();
    spB->hi();

    cout << "Operator *" << endl;
    (*spA).hi();
    (*spB).hi();

    cout << "Polymorphismus:" << endl;
    smartPointer<A> spAB; //Basisklassenzeiger
    spAB = new B;         // zeigt auf B-Objek
    spAB->hi();           // B::hi()
}
$
```

```
$ main
Zeiger auf dynamische Objekte:
Konstruktoraufruf
... oder Zuweisung
Operator ->
hier ist A::hi()
hier ist B::hi()
Operator *
hier ist A::hi()
hier ist B::hi()
Polymorphismus:
hier ist B::hi()
B::Destruktor
A::Destruktor
B::Destruktor
A::Destruktor
A::Destruktor
$
```

Die Parameterübergabe eines smartPointer und die Sicherheitsmaßnahmen werden in folgendem Programm deutlich:

```
$ cat main2.cpp
...
// Übergabe per Wert ist hier nicht möglich, wohl aber per Referenz:
template <class T>
void perReferenz(smartPointer<T>& p) {
    cout << "Aufruf: perReferenz(smartPointer<T>&):";
    p->hi();
}

int main() {
    smartPointer<A> spAB;            // Basisklassenzeiger
    spAB = new B;                    // zeigt auf  B-Objekt
    spAB->hi();                      // B::hi()

    // Parameterübergabe eines smartPointer
    perReferenz(spAB);

    spAB.loescheObjekt();
```

```
/* Die  Wirkung der Sicherungsma?nahmen im Vergleich zu einfachen
   C-Zeigern zeigen die folgenden Zeilen:
*/

// nicht-initialisierter Zeiger bewirkt Laufzeitfehler:
    smartPointer<B> spUndef;
    if (!spUndef)          // =  if(!(spUndef.operator bool()))
        cout << "undefinierter Zeiger!" << endl;
    spUndef->hi();                   // Laufzeitfehler!
    (*spUndef).hi();                 // Laufzeitfehler!
$
```

```
$ main2
hier ist B::hi()
Aufruf: perReferenz(smartPointer<T>&):hier ist B::hi()
B::Destruktor
A::Destruktor
undefinierter Zeiger!
main2: smartptr.t:35: T* smartPointer<T>::operator->()
const [with T = B]: Assertion `ZeigerAufObjekt' failed.
Abgebrochen
$
```

Wir werden smart-Pointer später nochmals an Hand der boost-Bibliothek kennen lernen !

10 Objekte als Funktionen (Funktoren)

Ein Funktionsaufruf innerhalb eines Ausdrucks bewirkt, dass die Funktion ausgeführt wird und das Resultat (ein Wert) an die Aufrufstelle kopiert wird.

Soll ein **Objekt** die **Aufgabe** einer **Funktion** übernehmen, damit man z.B. von Vererbung profitieren kann, so wird einfach der **Funktionsoperator** „()“ **überladen**. Solche Objekte, die sich wie Funktionen verhalten nennt man ***Funktoren***.

Funktoren werden häufig verwendet, da sie die Vorteile von Funktionen und Objekten vereinen:

- ein Funktor kann in einem Ausdruck vorkommen
- ein Funktor kann wie jedes Objekt seinen Zustand ändern (mit Funktionen eingeschränkt nur über static Variablen möglich)
- Funktoren sind in Klassenhierarchien möglich, somit sind Vererbungen erlaubt
- Funktoren können als Parameter an Funktionen übergeben werden; damit werden automatisch die internen Daten mit übergeben

Ein **Funktor** wird **realisiert**, indem eine Klasse eine Methode mit überladenem **Klammeroperator** **`operator()()`** implementiert.

Funktoren sollten eingesetzt werden, wenn man eigentlich Zeiger auf Funktionen benötigt und weiterhin die Vorteile der OOP ausgenutzt werden sollen.

Beispiel (Berechnung vom Sinus eines Winkels):

```
$ cat sinus.h
#ifndef sinus_h
#define sinus_h sinus_h
#include<cassert>
#include<cmath>   // sin(), Konstante  M_PI für pi

class SINUS {
    public:
      enum Modus { Bogenmass, Grad, Neugrad };
      SINUS(Modus M=Bogenmass)
      : Berechnungsart(M) {
      }
      const double operator()(double arg) {
        double erg;
        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    : assert(0);       // darf nicht vorkommen
        }
        return erg;
      }
    private:
       Modus Berechnungsart;
};
#endif
$
```

Damit kann man eine **Instanz** von sinus **wie eine Funktion** verwenden:

```
$ cat main.cpp
#include<iostream>
#include"sinus.h"
using namespace std;

void SinusAnzeigen(double arg, SINUS& Funktor) {
    cout << 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 <<" rad) = " << sinrad(M_PI/4)   << endl;
    cout << "sin(45 Grad)     = "         << sinGrad(45.0)    << endl;
    cout << "sin(50 Neugrad) = "          << sinNeuGrad(50.0) << endl;

    // Übergabe eines Funktors an eine Funktion
    SinusAnzeigen(50.0, sinNeuGrad);
}
$
```

```
$ main
sin(0.785398 rad) = 0.707107
sin(45 Grad)     = 0.707107
sin(50 Neugrad) = 0.707107
0.707107
```

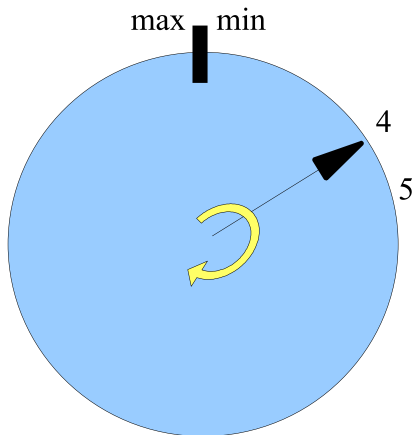
Hörsaalübung

Realisieren Sie ein Klasse I für Integer mit folgender Spezifikation:

- privates int-Attribut
- öffentliche Methoden set und get und Konstruktor, der Wert mit 0 initialisiert

Von I soll eine Klasse RI abgeleitet sein mit zwei ínt-Konstanten min und max als private Attribute, die den Range der Werte, die angenommen werden dürfen angibt. RI soll die Inkrementoperatoren überschreiben haben, so dass beim Überschreiten der oberen Grenze max der neue Wert dann min wird.

Weiterhin soll verwaltet werden, wie oft die obere Grenze überschritten wurde, ganz gleich bei welchen Instanz von RI.




```
int main(int argc, char** argv) {
    I *pi = new RI(0,10);
    pi->set(5);
    cout << pi->get() << endl;

    I i(10);
    I j;
    j = i;
    cout << j.get() << endl;

    RI r1(1,3);
    for (int i=1; i<8; i++) {
      cout << (r1++).get() << endl;
      cout << "r1 no= " << RI::no << endl;
    }
    return 0;
}
```