3 Erster Überblick über Klassen

3.1 Klassenbegriff

3.2 Klassenkonstruktion

3.3 Copy-Konstruktor, Zuweisungs-operator und Destruktor

3.4 Vordefinierte Parameterwerte

3.5 Referenzen

3.6 Ein- und Ausgabeoperationen

3.7 Typwandlungen

3.8 friends

3.9 Auszug aus Template Klasse vector

3.10 Initialisierung

Eine Klasse führt einen neuen Typ ein.

Form:

```
class NEU {
public:
        //  Öffentliche Schnittstelle,  Typdefinition

private:
        //  Details der Implementation
}; //NEU
```

Grundoperationen, die für fast jeden neuen Typ T gelten:

Erschaffen neuer Objekte vom Typ T,
Zerstören alter Objekte vom Typ T,
Herstellen von Objektkopien,
Zuweisen von Objekten zu Variablen,
Bewegen von Objekten an anderen Ort,
Transformation von Objekten vom Typ TA zu Objekten vom Typ TB.

Bemerkungen:

(i) **Es gibt auch ein Schlüsselwort "protected" für die Zugriffskontrolle.**

(ii) **Es ist möglich, Typen zu erweitern durch Einführung neuer öffentlicher Operationen und durch Einführung zusätzlicher Implementations-details.**

(iii) **Die Spezifikation und die Implementation neuer Typen sollte getrennt erfolgen können.**

(iv) **Schlüsselwörter zur Erzeugung eines neuen Typs sind u. a.: class, struct, union.**

Konstruktoren

```
//  Bruch 1
//  Beispiel eines einfachen Datentyps
//  Nur Konstruktion und simple Nutzung

#include <iostream>
#include <cstdlib>
using  std::cerr;                  // Elemente aus std
using  std::cout;
using  std::endl;
using  std::exit;



class Bruch {                 //  Deklaration eines neuen Typs
public:                       //  Operationenteil
      //  Mehrere Konstruktoren
      Bruch ();
      Bruch (int);
      Bruch (int, int);
      // Eine Funktion
      void print ();
private:
      int zaehler;            //  Datenteil
      int nenner;
};//Bruch
```

```
//  Implementation der Methoden


Bruch::Bruch () {
   // Bruch mit 0 initialisieren
   zaehler  =  0;
   nenner   =  1;
}

// Konstruktor aus ganzer Zahl
Bruch::Bruch (int z) {
   zaehler  =  z;
   nenner   =  1;
}

//Konstruktor aus Zähler und Nenner
Bruch::Bruch (int z, int n) {
   if (n == 0) {
            // Programm mit Fehlermeldung beenden
            cerr << "Fehler: Nenner ist 0!" << endl;
            exit (EXIT_FAILURE);
   }
   zaehler  =  z;
   nenner  =  n;
}

void Bruch::print () {
       //  Nur zur Illustration
       cout << zaehler << '/' << nenner << endl;
}//print
```

```
int main () {

      Bruch x;                        //  Default-Konstruktor
      Bruch a (7, 3);                 //  int/int-Konstruktor

      Bruch b  =  Bruch (8, 9);

      cout << "Bruch x = ";           //  Ausgabe
      x.print ();
      cout << "Bruch a = ";           //  Ausgabe
      a.print ();

      cout << "Bruch b = ";           //  Ausgabe
      b.print ();

      x  =  a;                        //  Zuweisungsoperator
                                      //  compilergeneriert

      a  =  Bruch (1000);

      cout << "Bruch x = ";  x.print ();
      cout << "Bruch a = ";  a.print ();
}//main


/*  Ausgabe:

Bruch x = 0/1
Bruch a = 7/3
Bruch b = 8/9
Bruch x = 7/3
Bruch a = 1000/1
*/
```

```
//  Vereinfachung von Bruch 1

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

class Bruch {              //  Deklaration eines neuen Typs
public:                    //  Operationenteil
      //  Mehrere Konstruktoren
      Bruch ();
      Bruch (int);
      Bruch (int, int);
      // Eine Funktion
      void print ();
private:                   //  Datenteil
      int zaehler = 0;
      int nenner  = 1;
};//Bruch

//  Implementation der Methoden

Bruch::Bruch () { }  // Default-Konstrutor
Bruch::Bruch (int z) {// Konstruktor aus ganzer Zahl
   zaehler  =  z;
}
//Konstruktor aus Zähler und Nenner
Bruch::Bruch (int z, int n) {
   if (n == 0) {
            // Programm mit Fehlermeldung beenden
            cerr << "Fehler: Nenner ist 0!" << endl;
            exit (EXIT_FAILURE);
   }
   zaehler  =  z;    nenner  =  n;
}
```

```
void Bruch::print () {
      //  Nur zur Illustration
      cout << zaehler << '/' << nenner << endl;
}//print


int main () {

      Bruch x;                        //  Default-Konstruktor
      Bruch a (7, 3);                 //  int/int-Konstruktor
      Bruch b  =  Bruch (8, 9);
      cout << "Bruch x = ";    x.print ();     //  Ausgabe
      cout << "Bruch a = ";    a.print ();     //  Ausgabe
      cout << "Bruch b = ";    b.print ();     //  Ausgabe
      x  =  a;        //  Zuweisungsoperator compilergeneriert
      a  =  Bruch (1000);
      cout << "Bruch x = ";  x.print ();
      cout << "Bruch a = ";  a.print ();

}//main


/*  Ausgabe:

Bruch x = 0/1
Bruch a = 7/3
Bruch b = 8/9
Bruch x = 7/3
Bruch a = 1000/1

*/
```

```
/*  Static variable in C                                        */
/*  Static Variable überleben Aufrufe von Funktionen. */

#include <stdio.h>

void buildsumme (void) {
    static int sum = 0;

    int num;
    printf ("Bitte Zahl:  ");
    scanf ("%d",  &num);
    sum  +=  num;
    printf ("Gesamtsumme bisher:  %d\n", sum);
}

int main (void) {
    buildsumme ();
    buildsumme ();
    buildsumme ();
    buildsumme ();
}

/*
Bitte Zahl: 32
Gesamtsumme bisher:  32
Bitte Zahl: 16
Gesamtsumme bisher:  48
Bitte Zahl: 128
Gesamtsumme bisher:  176
Bitte Zahl: 8
Gesamtsumme bisher:  184
*/
```

```
//  Erweiterung von Bruch 1
//  Destruktor,  Copy-Konstruktor und Zuweisung

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


class Bruch {
public:
	//  Default-Konstruktor
	Bruch ();
	Bruch (int,  int);
	//  Destruktor
	~Bruch () {
		cout << "Bruch No. " << kennzeichen
			<< " zerstört!" << endl;
	}
	//  Zuweisungsoperator
	Bruch&  operator =  (const Bruch&);
	//  Copy Konstruktor
	Bruch (const Bruch&);
	//  Weitere Methoden
	void print ();
	Bruch  operator * (Bruch);
	Bruch&  operator *= (Bruch);
private:
	int zaehler;
	int nenner;
	int kennzeichen;
	static int Nummer;
};//Bruch
```

```
//Initialisierung der Variablen mit Attribut static
int Bruch::Nummer = 0;

Bruch::Bruch () {
	kennzeichen = ++Nummer;
	cout  <<  "Bruch No. "  << kennzeichen
		<<  " geschaffen!"  <<  endl;
}

Bruch::Bruch (int z,  int n) {
	 //  Nur sinnvolle Nutzung
	zaehler = z;
	nenner  = n;
	kennzeichen = ++Nummer;
	cout << "Bruch No. " << kennzeichen
		<< " geschaffen!" << endl;
}

Bruch::Bruch (const Bruch& rs) {
	zaehler = rs.zaehler;
	nenner = rs.nenner;
	kennzeichen = ++Nummer;
	cout << "Bruch No. " << kennzeichen
		<< " in Copy-Konstruktor benannt!" << endl;
}

Bruch& Bruch::operator = (const Bruch& rs) {
	if  (this != &rs) {
		zaehler = rs.zaehler;
		nenner  = rs.nenner;
	}
	cout << "Zuweisung ausgeführt!" << endl;
return *this;
}
```

```
void Bruch::print () {
	cout << zaehler << '/' << nenner << endl;
}//print

Bruch Bruch::operator * (Bruch b) {
   return Bruch (zaehler * b.zaehler,  nenner * b.nenner);
}//*

Bruch& Bruch::operator *= (Bruch b) {
	this->zaehler *= b.zaehler;
	this->nenner  *= b.nenner;
return *this;
}//*=



int main() {
	Bruch x (2, 3);
	cout << "    x = ";  x.print ();

	x = x;
	Bruch a (7, 3);
	cout << "    a = ";  a.print ();

	Bruch y (x);
	cout << "    y = ";  y.print ();

	Bruch ya = a;                    //  Kopie erstellt
	cout << "    ya = ";  ya.print ();

	Bruch z = Bruch (5, 7) * Bruch (10, 7);

	cout <<"    z = ";  z.print ();
}//main
```

/* Ausgabe

Bruch No. 1 geschaffen!
**   x = 2/3**
Zuweisung ausgeführt!
Bruch No. 2 geschaffen!
**   a = 7/3**
Bruch No. 3 in Copy-Konstruktor benannt!
**   y = 2/3**
Bruch No. 4 in Copy-Konstruktor benannt!
**   ya = 7/3**
Bruch No. 5 geschaffen!
Bruch No. 6 geschaffen!
Bruch No. 7 geschaffen!
Bruch No. 5 zerstört!
Bruch No. 6 zerstört!
**   z = 50/49**
Bruch No. 7 zerstört!
Bruch No. 4 zerstört!
Bruch No. 3 zerstört!
Bruch No. 2 zerstört!
Bruch No. 1 zerstört!

***/**

Weiteres zu Konstruktoren

Im wesentlichen gibt es nur vier Formen für den Kopier-Konstruktor: X::X(X&), X::X(const X&), X::X(volatile X&), X::X(const volatile X&).

```
//  Copy Constructor
#include <iostream>
using namespace std;


class X {
            // ...
public:
   X (int) {
      cout << "Aufruf Konstruktor!" << endl;
   }
   X (X&,  int = 1) {
      cout << "Aufruf Copy-Konstruktor 1!" << endl;
   }
   X (const X&,  int = 2) {
      cout << "Aufruf Copy-Konstruktor 2!" << endl;
   }
};

int main () {
    X a (1);            // Aufruf X (int);
    const X b (2);   // Aufruf X (int)
    X c (a, 0);         // Aufruf X (X&,  int);
    X d = b;            // Aufruf X (const X&,  int);
}//main
```

```
//   Form von Konstruktoren
#include <iostream>
using namespace std;

class CA {
public:
	CA (int a,  double b) {
		i  = a;
		d = b;
		cout << "In Konstruktor 1!" << endl;
	}

	CA (double b)  :
		d {b},  i {int (b/10)} {
		cout << "In Konstruktor 2!" << endl;
	}

	CA (CA c,  CA e) {
		i = c.i;
		d = 2.0 * e.d;
		cout << "In Konstruktor 3!" << endl;
	}

	void aus () {
		cout  << "Wert von i = "  << i
			<< "\nWert von d = "  << d
			<< endl;
	}
private:
	double  d;
	int	i;
};//CA
```

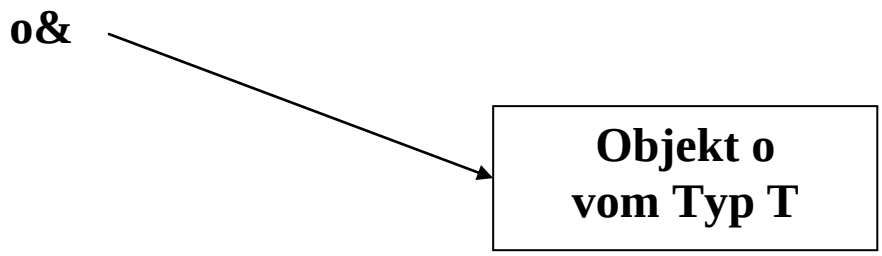
```
int main () {

        CA o1 (3, 7.2);
        o1.aus ();
        CA o2 (101.6);
        o2.aus ();
        CA o3 (o1, o2);
        o3.aus ();

}//main



/*  Ausgabe:

In Konstruktor 1!
Wert von i = 3
Wert von d = 7.2
In Konstruktor 2!
Wert von i = 10
Wert von d = 101.6
In Konstruktor 3!
Wert von i = 3
Wert von d = 203.2

*/
```

```
//  Erweiterung von Bruch 1
//  Gesetzte Parameter
//  Schlüsselwort inline

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


class Bruch {
public:
        //  Ein umfassender Konstruktor
        Bruch (int = 0,  int = 1);
        //  Einige Methoden
        Bruch  operator * (Bruch);
        Bruch&  operator *= (Bruch);
        bool operator < (Bruch);
        void print () {
                cout << zaehler << " / " << nenner << endl;
        }
private:
        int zaehler;
        int nenner;
};//Bruch


inline Bruch  Bruch::operator * (Bruch b) {
   return Bruch (zaehler * b.zaehler,  nenner * b.nenner);
}//*
```

```
Bruch::Bruch (int z,  int n) {
        if (n == 0) {
                cerr << "Fehler: Nenner ist 0!" << endl;
                exit (EXIT_FAILURE);
        }
        //  Erste Normalisierung
        if (n < 0) {
                zaehler  = -z;
                nenner  = -n;
        } else {
                zaehler  = z;
                nenner  = n;
        }
}


Bruch&  Bruch::operator *= (Bruch b) {
        zaehler  *=  b.zaehler;
        nenner   *=  b.nenner;
return *this;
}//*=


inline bool Bruch::operator < (Bruch b) {
    // Da Nenner > 0
    return   zaehler * b.nenner  <  b.zaehler * nenner;
}
```

```
int main () {
	Bruch a (7, 3);

	cout << "Bruch a = ";
	a.print ();

	Bruch x = a * a;
	cout << "Bruch x = ";
	x.print ();

	while (x < Bruch (100000)) {
		x *= a;
		cout << "Bruch x = ";
		x.print ();
	}
}//main


/*  Ausgabe:

Bruch a = 7 / 3
Bruch x = 49 / 9
Bruch x = 343 / 27
Bruch x = 2401 / 81
Bruch x = 16807 / 243
Bruch x = 117649 / 729
Bruch x = 823543 / 2187
Bruch x = 5764801 / 6561
Bruch x = 40353607 / 19683
Bruch x = 282475249 / 59049
Bruch x = 1977326743 / 177147

*/
```

// Referenzen



Bemerkung: Referenzen sind Aliasbezeichnungen für Objekte. Referenzen müssen daher initialisiert werden.

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

void f (int i,  int& j) {
   ++i;
   ++j;
}//f


int main () {
   int a  {};
   int b  {};
   f (a, b);
   cout << "a = " << a << "   b = " << b << endl;
   //  a = 0   b = 1
}//main
```

```
//  Referenzen


#include<iostream>
using namespace std;

int f (const long& i) {
   return i+1;
}//f

void g (long& i) {
  f (i);
}//g


int main () {
   long la = 0;
   int a = 0;

   // g (a);       // Fehler bei Parameteranpassung
   // g (7);       // Fehler bei Parameteranpassung

   g (la);

   int   b  =  f (a);
   long c  =  f (7);
   la  =  f (la);       // la = 1

   cout << "la = " << la << "   a = " << a << endl;
   //  la = 1   a = 0
   cout << "b =  " << b << "   c = " << c << endl;
   //  b =  1   c = 8
}//main
```

```
//  Referenzen,  Beispiel aus Standard

#include <iostream>
using namespace std;

int g (int) {
   cout << "in g ()!" << endl;
return 0;
}//g

void f () {
   int i;
   int& r = i;         // r refers to i
   r = 1;              // the value of i becomes 1
   int* p = &r;        // p points to i
   int& rr = r;        // rr refers to what r refers to, that is, to i
   int (&rg) (int) = g;       // rg refers to the function g
   rg (i);                    // calls function g
   int a[3];
   int (&ra)[3] = a;          // ra refers to the array a
   ra [1] = i;                // modifies a[1]
}//f

int main () {
   const double&  dd  =  3.1;
   cout << "in main ()!" << endl;
   cout << "dd = " << dd << endl;
   f ();
}//main

/*
       in main ()!
       dd = 3.1
       in g ()!  */
```

```
//  Weiteres Beispiel aus Standard

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

int main () {
   int  i  =  101;
   typedef  int&     LRI;
   typedef  int&&  RRI;

   LRI&            r1 = i;             //  Typ:  int&
   const LRI&    r2 = i;             //  Typ:  int&
   const LRI&& r3 = i;             //  Typ:  int&

   cout << "Typ (r1) = " << typeid(r1).name() << endl;
   cout << "Typ (r2) = " << typeid(r2).name() << endl;
   cout << "Typ (r3) = " << typeid(r3).name() << endl;

   r1 = 111;
   cout << "r1 = " << r1 << "   i = " << i<< endl;
   r2 = 112;
   cout << "r2 = " << r2 << "   i = " << i<< endl;
   r3 = 113;
   cout << "r3 = " << r3 << "   i = " << i << endl << endl;

   RRI&      r4 = i;             // Typ:  int&
   RRI&&  r5 = 5;             // Typ:  int&&
   cout << "Typ (r4) = " << typeid(r4).name() << endl;
   cout << "Typ (r5) = " << typeid(r5).name() << endl;
   r4  =  r3 + 10;
   cout << "r4 = " << r4 << "   i = " << i<< endl;
```

```
    i   +=   r5;
    cout << "r5 = " << r5 << "   i = " << i << endl << endl;

    decltype(r2)&      r6 = i;          //  Typ:  int&
    decltype(r2)&&  r7 = i;          //  Typ:  int&
    cout << "Typ (r6) = " << typeid(r6).name() << endl;
    cout << "Typ (r7) = " << typeid(r7).name() << endl;
    i = 1024;
    cout << "r6 = " << r6 << "   i = " << i << endl;
    r7 = 117;
    cout << "r7 = " << r7 << "   i = " << i << endl;

}//main

/*
Typ (r1) = int
Typ (r2) = int
Typ (r3) = int
r1 = 111   i = 111
r2 = 112   i = 112
r3 = 113   i = 113

Typ (r4) = int
Typ (r5) = int
r4 = 123   i = 123
r5 = 5      i = 128

Typ (r6) = int
Typ (r7) = int
r6 = 1024   i = 1024
r7 = 117     i = 117

*/
```

```
//  Referenzen
//  Erweiterung von Bruch 1
//  Schlüsselwort const

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


class Bruch {
public:
   Bruch (int = 0,  int = 1);
   const Bruch& operator *= (const Bruch&);
   void print () const {
      cout << zaehler << " / " << nenner << endl;
   }
private:
   int  zaehler;
   int  nenner;
};//Bruch

Bruch::Bruch (int z,  int n) {
   if (n == 0) {
      cerr << "Fehler: Nenner ist 0!" << endl;
      exit (EXIT_FAILURE);
   }
   if  (n < 0)  {
      zaehler  =  -z;     nenner  = -n;
   } else  {
      zaehler  =  z;     nenner  =  n;
   }}// Bruch
```

```
inline
const Bruch& Bruch::operator  *=  (const Bruch& b) {
   zaehler *= b.zaehler;    nenner  *= b.nenner;
return *this;                //  nur zur Verkettung
}//*=


int main () {
   Bruch a (7, 3),   b (5, 2),   c (1, 4),   d (11, 6);
   cout << "Bruch  a = ";    a.print ();
   cout << "Bruch  b = ";    b.print ();
   cout << "Bruch  c = ";    c.print ();
   cout << "Bruch  d = ";    d.print ();
   Bruch e;
   e *= d *= c *= b *= a;
   cout << "Bruch  a  = ";   a.print ();
   cout << "Bruch  b  = ";   b.print ();
   cout << "Bruch  c  = ";   c.print ();
   cout << "Bruch  d  = ";   d.print ();
   cout << "Bruch  e  = ";   e.print ();
}//main


/*   Ausgabe:
Bruch a  = 7 / 3
Bruch b  = 5 / 2
Bruch c  = 1 / 4
Bruch d  = 11 / 6
Bruch a  = 7 / 3
Bruch b  = 35 / 6
Bruch c  = 35 / 24
Bruch d  = 385 / 144
Bruch e  = 0 / 144
*/
```

```
//  Erweiterung von Bruch 1
//  Ein- und Ausgabe mit Streams

#include <cstdlib>
#include <iostream>
#include <iomanip>
using namespace std;

class Bruch {
public:
   Bruch (int = 0, int = 1);
   Bruch operator * (const Bruch&) const;
   const Bruch& operator *= (const Bruch&);
   bool operator < (const Bruch&) const;
   void print (ostream& = cout) const;
   void scan (istream& = cin);
   bool korrpruefen ();
private:
   int zaehler   = 0;
   int nenner   = 1;
};//Bruch


bool Bruch::korrpruefen () {
       if  (nenner == 0)
             return false;
       if  (nenner < 0) {
             zaehler = -zaehler;
             nenner = - nenner;
       }
return true;
}//korrpruefen
```

```
inline Bruch Bruch::operator * (const Bruch& b) const {
   return Bruch (zaehler * b.zaehler, nenner * b.nenner);
}


// Globales Überladen von <<
ostream& operator << (ostream& strm,  const Bruch& b){
       b.print (strm);
return strm;                 //  Verkettung ermöglichen
}


// Globales Überladen von >>
istream& operator >> (istream& strm,  Bruch& b) {
       b.scan (strm);
return strm;                 //  Verkettung ermöglichen
}


Bruch::Bruch (int z,  int n) {
   if (n == 0) {
      cerr << "Fehler: Nenner ist 0" << endl;
      exit (EXIT_FAILURE);
   }
   if (n < 0) {
      zaehler  = -z;
      nenner  = -n;
   } else {
      zaehler  =  z;
      nenner  =  n;
   }
}
```

```
const Bruch& Bruch::operator *= (const Bruch& b) {
   zaehler  *= b.zaehler;
   nenner   *= b.nenner;
return *this;
}


bool Bruch::operator < (const Bruch& b) const {
   return   zaehler * b.nenner  <  b.zaehler * nenner;
}//<


void Bruch::print (ostream& strm) const {
   strm << zaehler << " / " << nenner;
}


void Bruch::scan (istream& strm) {
   strm >> zaehler;
   // optionales Trennzeichen '/' einlesen
   if ((strm>>ws).peek () == '/') {
      strm.get ();
   }
   // Nenner einlesen
   strm >> nenner;
}
```

```
int main () {

   const Bruch a (7, 3);
   Bruch x;

   cout << a << endl;

   cout << "Bitte Bruch (Zaehler/Nenner): ";
   if (! (cin >> x)) {
      // Eingabefehler: Programmabbruch
      cerr << "Fehler beim Lesen eines Bruchs" << endl;
      exit (EXIT_FAILURE);
   }
   if (!x.korrpruefen ()) {
      cerr << "Bruch mit Nenner Null entdeckt!" << endl;
      exit (EXIT_FAILURE);
      }
   cout << "Gelesen: " << x << endl;

   while (x < Bruch (1000)) {
      x *= a;
      cout << x << endl;
   }
}//main
```

/* Ausgabe zweier Läufe:

Lauf 1:

7 / 3
Bitte Bruch (Zaehler/Nenner): 7 / 10
Gelesen: 7 / 10
49 / 30
343 / 90
2401 / 270
16807 / 810
117649 / 2430
823543 / 7290
5764801 / 21870
40353607 / 65610
282475249 / 196830

Lauf 2:

7 / 3
Bitte Bruch (Zaehler/Nenner): 7 10
Gelesen: 7 / 10
49 / 30
343 / 90
2401 / 270
16807 / 810
117649 / 2430
823543 / 7290
5764801 / 21870
40353607 / 65610
282475249 / 196830

Typwandlungen

In einer objektorientierten Welt sollte es möglich sein, daß aus Objekten des Typs TA Objekte des Typs TB gebildet werden können.

In C++ beschreibt TA (TB) einen Konstruktor für Objekte vom Typ TA. Diesen Konstruktor kann man auch als Typwandler lesen. Neben diesem Typwandler bietet C++ auch die Konstruktion
operator Anderer_Typ () als Typwandler.

```
// Typwandlung
#include <cstdlib>
#include <iostream>
using namespace std;

class Bruch {
public:
   //  Typwandlung Zahl nach Bruch
   explicit Bruch (int = 0, int = 1);
   Bruch& operator *= (const Bruch&);
   bool operator < (const Bruch& b);
   // Konvertierung nach double
   operator double () const;
private:
   int zaehler  = 0;
   int nenner  = 1;
};
```

```
inline Bruch::operator double () const {
   return (double)zaehler / (double)nenner;
}

Bruch::Bruch (int z,  int n){
    if (n == 0) {
       cerr << "Fehler: Nenner ist 0" << endl;
       exit (1);
   }
   if (n < 0) {
       zaehler = -z;
       nenner  = -n;
   }
   else {
       zaehler = z;
       nenner  = n;
   }
}


Bruch& Bruch::operator *= (const Bruch& b) {
   zaehler *= b.zaehler;
   nenner  *= b.nenner;
return *this;
}

bool Bruch::operator < (const Bruch& b)
{
   // Da die Nenner nicht negativ sein koennen,  reicht:
   return   zaehler * b.nenner  <  b.zaehler * nenner;
}
```

```
int main () {
   const Bruch a (7,3);              // Konstante a deklarieren
   Bruch x (3, 4);                   // Variable x deklarieren
   // Bruch a ausgeben
   cout << "Bruch a = " << a << endl;

   //  solange x < 1000
   //  Hier erfolgt Wandlung von x in eine double-Größe.
   while (x < 1000) {
      // x mit a multiplizieren und ausgeben
      x  *=  a;
      //  Auch hier Wandlung von x in double!
      cout << x << endl;
   }
}//main



/*  Ausgabe:

Bruch a = 2.33333

1.75
4.08333
9.52778
22.2315
51.8735
121.038
282.422
658.985
1537.63

*/
```

```
//  friends
//  Erweiterung zu Bruch 1
#include <cstdlib>
#include <iostream>
#include  <iomanip>
using namespace std;


class Bruch {

   //  Globale Friend-Funktion ermöglicht symmetrische
   //  Behandlung beider Operanden.
   friend Bruch operator * (const Bruch&, const Bruch&);

   friend bool operator<(const Bruch& a, const Bruch& b)
   {    return  a.zaehler * b.nenner  <  b.zaehler * a.nenner;
   }

   //  Ein- und Ausgabe
   friend ostream& operator << (ostream& strm,
                                const Bruch& b);
   friend istream& operator >> (istream& strm,
                                Bruch& b);
public:
   Bruch (int = 0, int = 1);
   const Bruch& operator *= (const Bruch&);

private:
   int zaehler   {};
   int nenner   {1};

};//Bruch
```

```
// globale Friend-Funktion inline
inline Bruch operator * (const Bruch& a, const Bruch& b)
{    return Bruch (a.zaehler * b.zaehler,
                   a.nenner * b.nenner);
}

inline ostream& operator << (ostream& strm,
                             const Bruch& b) {
   strm << b.zaehler << '/' << b.nenner;
return strm;       // Stream zur Verkettung zurückliefern
}//<<

istream& operator >> (istream& strm, Bruch& b) {
   int  z, n;

   // Zaehler einlesen
   strm >> z;
   // optionales Trennzeichen '/' überlesen
   if ((strm>>ws).peek () == '/')) {
      strm.get (); }
   // Nenner einlesen
   strm >> n;
    // Nenner == 0 ?
   if (n == 0) {
      // Fail-Bit setzen
      strm.clear (strm.rdstate() | ios::failbit);
   return strm;  }
   if (n < 0) {
            b.zaehler = -z;      b.nenner  = -n;
   } else {  b.zaehler = z;      b.nenner  = n;
   }
return strm;
}//>>
```

```
Bruch::Bruch (int z, int n) {
   if (n == 0) {
      cerr << "Fehler: Nenner ist 0" << endl;
      exit (EXIT_FAILURE);
   }
   if (n < 0) {
      zaehler = -z;
      nenner  = -n;
   }
   else {
      zaehler = z;
      nenner  = n;
   }
}


const Bruch& Bruch::operator *= (const Bruch& b) {
   zaehler *= b.zaehler;
   nenner  *= b.nenner;
   return *this;
}



int main () {
   const Bruch a (7, 3);    // Bruch-Konstante a deklarieren
   Bruch b = a * 10;

   cout << "a = " << a << endl;
   cout << "b = " << b << endl;
   Bruch x;
```

```
    // Bruch x einlesen
    cout << "Bruch eingeben (zaehler/nenner): ";
    if (! (cin >> x)) {
        // Eingabefehler: Programmabbruch
        cerr << "Fehler bei Bruch-Eingabe!" << endl;
        exit (EXIT_FAILURE);
    }
    cout << "Eingabe war: " << x << endl;

    // solange x < 1000
    while (x < 1000) {              // Typwandlung
        // x mit a multiplizieren und ausgeben
        x *= a;
        cout << x << endl;
    }
}//main


/*  Ausgabe:
a = 7/3
b = 70/3
Bruch eingeben (zaehler/nenner): 3/8
Eingabe war: 3/8

21/24
147/72
1029/216
7203/648
50421/1944
352947/5832
2470629/17496
17294403/52488
121060821/157464
847425747/472392
```

Weitere Läufe:

a = 7/3
b = 70/3
Bruch eingeben (zaehler/nenner): a
Fehler bei Bruch Eingabe!

a = 7/3
b = 70/3
Bruch eingeben (zaehler/nenner): 4 / b
Fehler bei Bruch Eingabe!

a = 7/3
b = 70/3
Bruch eingeben (zaehler/nenner): 4 / 0
Fehler bei Bruch Eingabe!

***/**

Template-Klasse vector

Konstruktoren

- **vector <Typ> m**
- **vector <Typ> m1 (m2)**
- **vector <Typ> m (10)**
- **vector <Typ> m (10, elem)**
- **vector <Typ> m (anf, end)**

Destruktor

- **~vector <Typ> ()**

Funktionen

- **size()**
- **empty()**
- **max_size()**
- **capacity()**
- **reserve()**
- **==**
- **!=**
- **<**
- **>**
- **<=**
- **>=**

Zuweisungen

- **= m2**
- **assign (10)**
- **assign (10, elem)**
- **assign (anfang, ende)**
- **swap (m2)**
- **swap (m1, m2)** **globale Funktion**

Elementzugriff

- **at ()**
- **[]**
- **front ()**
- **back ()**
- **data ()**

Iteratoren

- **begin ()**
- **end ()**
- **rbegin ()**
- **rend**
- **cbegin ()**
- **cend ()**
- **crbegin ()**
- **crend ()**

Einfügen und Löschen

- **insert (pos)**
- **insert (pos, elem)**
- **insert (pos, anf, end)**
- **push_back (elem)**
- **erase (pos)**
- **erase (anf, end)**
- **pop_back ()**
- **resize (anzahl)**
- **resize (anzahl, elem)**
- **clear ()**

```
// Vergleiche von Vektoren
// Vektoren werden lexikographisch verglichen

#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;



int main () {
   vector<int> v1 {7, 8, 9};
   vector<int> v2 {2, 3, 4, 5, 6, 7};
   vector<int> v3 {1, 10, 11, 12};
   bool v2v1 = v2 < v1;
   bool v3v1 = v3 < v1;
   bool v3v2 = v3 < v2;

   cout << boolalpha
        << "v2v1 <?  " << v2v1 << '\n'
        << "v3v1 <?  " << v3v1 << '\n'
        << "v3v2 <?  " << v3v2 << endl;
}


/*
v2v1 <?  true
v3v1 <?  true
v3v2 <?  true
*/
```

```
//  swap vectors
//  lokale Funktion

#include <iostream>
#include <vector>


int main () {
   std::vector<int> foo (3, 100);
   std::vector<int> bar (5, 200);

   foo.swap (bar);

   std::cout << "foo enthaelt:  ";
   for (auto i : foo)
      std::cout << ' ' << i;
   std::cout << std::endl;

   std::cout << "bar enthaelt:  ";
   for (auto i : bar)
      std::cout << ' ' << i;
   std::cout << std::endl;
}

/*

foo enthaelt:   200 200 200 200 200
bar enthaelt:   100 100 100

*/
```

```
// vectors
// verschiedene Konstruktoren

#include <iostream>
#include <vector>



int main () {
  std::vector<int> first;
  std::vector<int> second (4,100);
  std::vector<int> third (second.begin(), second.end());
  std::vector<int> fourth (third);

  // Konstruktion über Arrays
  int a [] = {16, 2, 77, 29};
  std::vector<int> fifth (a, a + sizeof(a) / sizeof(int));

  std::cout << "Inhalt von first :  ";
  for (auto i : first)
     std::cout << ' ' << i;
  std::cout << std::endl;

std::cout << "Inhalt von second : ";
  for (auto i : second)
       std::cout << ' ' << i;
  std::cout << std::endl;

std::cout << "Inhalt von third :  ";
  for (auto i : third)
    std::cout << ' ' << i;
  std::cout << std::endl;
```

```
  std::cout << "Inhalt von forth :  ";
  for (auto i : fourth)
     std::cout << ' ' << i;
  std::cout << std::endl;

  std::cout << "Inhalt von fifth :  ";
  for (std::vector<int>::iterator it = fifth.begin(); it !=
fifth.end(); ++it)
     std::cout << ' ' << *it;
  std::cout << std::endl;
}//main


/*

Inhalt von first :
Inhalt von second :  100 100 100 100
Inhalt von third :     100 100 100 100
Inhalt von forth :     100 100 100 100
Inhalt von fifth :      16  2  77  29

*/
```

```
// vector::cbegin/cend
// c ist ein Hinweis auf constant


#include <iostream>
#include <vector>


int main () {
   std::vector<int> va = {10, 20, 30, 40, 50};

  s  td::cout << "va enthaelt : ";

   for (auto it = va.cbegin(); it != va.cend(); ++it)
       std::cout << '  ' << *it;
   std::cout << std::endl;

   std::vector<int> vb = {110, 120, 130, 140, 150};

   // "vb wird geaendert!";
   for (auto it = vb.cbegin(); it != vb.cend(); ++it)
       // *it += 12; //  Compiler moniert Veränderung von
                     //  Nur-Lese-Daten
}//main


/*
va enthaelt :  10  20  30  40  50
*/
```

```
//  push_back (const T&)
//  push_back (T&&)

#include <iostream>
#include <utility>
#include <vector>
#include <string>
using namespace std;

int main() {
   string str1 = "Hallo";
   vector<string> v;

   // Nutzung von push_back (const T&)
   v.push_back (str1);
   cout << "Nach Eingabe, str1 = " << str1 << endl;

   // Nutzung von push_back (T&&)
   v.push_back (std::move(str1));
   cout << "Nach Eingabe, str1 = " << str1 << endl;;

   cout << "Inhalt von vector  v = " << v[0]
        << " , " << v[1] << endl;

   string str2 = "Danke!";
   cout << "Vor  move von str2, str2 = " << str2 << endl;
   v[0] = std::move (str2);
   cout << "Nach move von str2, str2 = " << str2 << endl;
   cout << "Inhalt von vector v = " << v[0]
        << " , " << v[1] << endl;

}
```

/*
Ausgabe nach Nutzung von g++ 4.8.2

Nach Eingabe, str1   = Hallo
Nach Eingabe, str1   =
Inhalt von vector v = Hallo , Hallo
Vor move von str2, str2 = Danke!
Nach move von str2, str2 = Hallo
Inhalt von vector v = Danke! , Hallo

Ausgabe nach Nutzung von VS2012

Nach Eingabe, str1 = Hallo
Nach Eingabe, str1 =
Inhalt von vector v = Hallo , Hallo
Vor move von str2, str2 = Danke!
Nach move von str2, str2 =
Inhalt von vector v = Danke! , Hallo

***/**

```
//  Initialisierung von Klasseninstanzen

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

class D {
public:
    D () {};
    D (double b) {x = 5.5;  y = b;}
    D (double a, double b) {x = a; y = b;}
    void aus (string s) {
        cout << s << ":  x = " << x << "  \ty = " << y <<endl;
    }
private:
    double x = 101.101;
    double y = 202.303;
};


int main () {
   D d (33.33);
   d.aus ("d ");
   D da = d;
   da.aus ("da");
   D db = D (17.7, 18.8);
   db.aus ("db");
   D dc;
   dc.aus ("dc");
   D dd = 6;
   dd.aus ("dd");
   D df = {10, 12};
   df.aus ("df");
```

```
   struct ST {
      int         ii;
      double  dd;
      D          d;
    } st = {101, 102.3, 103.4};

    cout << "st.ii = " << st.ii << "   \tst.dd = " << st.dd
         << endl;
    st.d.aus ("st.d");
    cout << endl;
}//main



/*
d :   x = 5.5     y = 33.33
da:  x = 5.5     y = 33.33
db:  x = 17.7   y = 18.8
dc:   x = 101.101        y = 202.303
dd:  x = 5.5     y = 6
df:   x = 10      y = 12
st.ii = 101       st.dd = 102.3
st.d:  x = 5.5         y = 103.4
*/
```

```
//  Anfangswerte für Klassenelemente
#include <iostream>
using namespace std;



struct A {              // Initialisierung im Körper
   char     a;
   int      b;
   double   c;
   A (char ca,  int in,  double da) {
      a = ca;  b = in;  c = da;
   }
   void aus (char cc []) {
      cout << "Inhalt von " << cc << " = " << a << "   "
           << b << "   " << c << endl;
   }
};



struct B {              //  Initialisierung mittels Liste
   char     a;
   int      b;
   double   c;
   B (char ca,  int in,  double da) : a (ca), b (in), c (da) {}
   void aus (char cc []) {
      cout << "Inhalt von " << cc << " = " << a << "   "
           << b << "   " << c << endl;
   }
};
```

```
struct C {   //  Initialisierung mittels Liste
   char      a;
   int       b;
   double  c;
   C (char a,  int b,  double c) : a (a), b (b), c (c) {}
   void aus (char cc []) {
      cout << "Inhalt von " << cc << " = " << a << "  "
           << b << "  " << c << endl;
   }
};


struct D {   //  Verteilung
   char      a;
   int       b;
   double  c;
   D (char a,  int b,  double c) : b (b) {
        D::a = a;  this->c = c;
   }
   void aus (char cc []) {
      cout << "Inhalt von " << cc << " = " << a << "  "
           << b << "  " << c << endl;
   }
};
```

```
int main () {

    A a ('A', 12, 34.56);
    a.aus ("a");
    B b ('B', 67, 78.90);
    b.aus ("b");
    C c ('C', 34, 45.89);
    c.aus ("c");
    D d ('D', 77, 3.123);
    d.aus ("d");

}//main



/* Ausgabe:

Inhalt von a = A  12  34.56
Inhalt von b = B  67  78.9
Inhalt von c = C  34  45.89
Inhalt von d = D  77  3.123

*/
```

```
// Zur Delegation in einer Konstruktorenschar



#include <iostream>
using namespace std;


class c1 {
public:
   int max = 10;
   int min = 1;
   int middle = 5;

   c1 () {}
   c1 (int mmax) {
      max = mmax > 0 ? mmax : 10;
      cout << "class (int)" << endl;
   }
   c1 (int mmax, int mmin) : c1 (mmax) {
      min = mmin > 0 && mmin < max ? mmin : 1;
      cout << "class (int, int)" << endl;
   }
   c1 (int mmax, int mmin, int mmiddle) :
        c1 (mmax, mmin) {
      middle =       mmiddle < max
                  && mmiddle > min ? mmiddle : 5;
      cout <<"class (int, int, int)" << endl;
   }
};
```

```
int main() {

    c1 ca {1, 3, 2 };
    cout << ca.max << "   " << ca.min << "   "
         << ca.middle << endl;
}//main


/*

class (int)
class (int, int)
class (int, int, int)
1  1  5
*/
```