Bsp.: Komplexe Zahlen (sehr rudimentär)

```
#include <iostream>

using namespace std;

class Complex {
  private:
    double re,im;

  public:
    Complex (double Re=0, double Im=0): re(Re), im(Im) { } // Konstruktor

    double real () { return re; }   // Komponentenfunktion
    double imag () { return im; }   // Komponentenfunktion

    friend Complex conj(Complex z)  // keine Komponentenfunktion
      { z.im = -z.im; return z; }
};


int main()
{
  Complex z1(1.0,2.0), z2;
  z2 = conj(z1);

  double x3,y3;
  cout << "Re z3, Im z3: ";
  cin  >> x3 >> y3;
  Complex z3(x3,y3);

  cout << "Re z1 = " << z1.real() << "   Im z1 = " << z1.imag() << endl;
  cout << "Re z2 = " << z2.real() << "   Im z2 = " << z2.imag() << endl;
  cout << "Re z3 = " << z3.real() << "   Im z3 = " << z3.imag() << endl;

  return 0;
}
```

Ausgabe:

```
Re z3, Im z3: 3 4
Re z1 = 1   Im z1 = 2
Re z2 = 1   Im z2 = -2
Re z3 = 3   Im z3 = 4
```

Anmerkungen zum Programm:

- In die Konstruktordefinition kann zwischen der Parameterliste und dem Funktionsblock eine Konstruktorinitialisierungsliste eingeschoben sein. Sie dient der Initialisierung der Datenkomponenten vor Ausführung des Funktionsblocks des Konstruktors.
- Parametervoreinstellungen sind auch für Konstruktordefinitionen möglich.

Überladen von Operatoren

Ein Operator @ kann (mit wenigen Ausnahmen) mittels einer Funktion `operator@` für nicht eingebaute Datentypen definiert werden, der binäre Ausdruck $x@y$ wird dann in `operator@(`x`,`y`)` umgesetzt. (Überladen ist auch für unäre Operatoren möglich, als Funktionen können auch Komponentenfunktionen eingesetzt werden). Vorrang und Syntax entspricht der für die eingebauten Operanden.

Bekanntestes Beispiel für einen überladenen Operator ist die Ein/Ausgabe mittels `>>` bzw. `<<` in der Standardbibliothek.

Bsp.: Addition und Ein/Ausgabe für komplexe Zahlen

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

class Complex {
  private:
    double re,im;
  public:
    Complex (double Re=0, double Im=0): re(Re), im(Im) { }

    friend Complex operator+(Complex z1, Complex z2)
      { return Complex(z1.re+z2.re, z1.im+z2.im); }

    friend ostream& operator<<(ostream& stream, Complex z)
      { stream << "(" << z.re << "," << z.im << ")" ;
        return stream; }

    friend istream& operator>>(istream& stream, Complex& z)
      { char c1,c2,c3;
        double x,y;
        stream >> c1 >> x >> c2 >> y >> c3;
        if (c1!='(' || c2!=',' || c3!=')')
          stream.setstate(ios::failbit);
        z = Complex(x,y);
        return stream; }
};

int main()
{
  Complex z1,z2;
  cout << "z1 z2: ";
  cin  >> z1 >> z2;
  cout << "z1+z2 = " << z1+z2 << endl;
  return 0;
}
```

Ausgabe:

```
z1 z2: (1,2) (3,4)
z1+z2 = (4,6)
```

Überladen von Funktionen

Neben Operatoren können auch Funktionen überladen werden, d.h. diejenige wird ausgewählt, deren Parametertypen am besten zu den Argumenttypen passen.

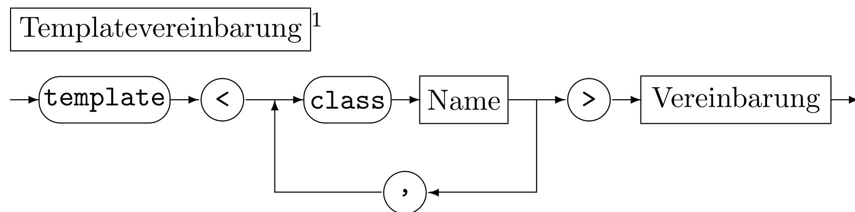
Bsp.: Komplexe Wurzel

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

class Complex {
  private:
    double re,im;

  public:
    Complex (double Re=0, double Im=0): re(Re), im(Im) { }

    friend ostream& operator<<(ostream& stream, Complex z)
      { stream << "(" << z.re << "," << z.im << ")" ;
        return stream; }

    friend Complex operator-(Complex z)
      { return Complex(-z.re,-z.im); }

    friend Complex sqrt(Complex z)  // keineswegs optimal
      { double r   = pow(z.re*z.re+z.im*z.im,0.25),
               phi = atan2(z.im,z.re)/2.0;
        return Complex(r*cos(phi),r*sin(phi)); }
};

int main()
{
  double  two=2.0;
  Complex zwei(2.0);

  cout << "sqrt(two)   = "  << sqrt(two)   << endl
       << "sqrt(zwei)  = "  << sqrt(zwei)  << endl
       << "sqrt(-two)  = "  << sqrt(-two)  << endl
       << "sqrt(-zwei) = "  << sqrt(-zwei) << endl;

  return 0;
}
```

Ausgabe:

```
sqrt(two)   = 1.41421
sqrt(zwei)  = (1.41421,0)
sqrt(-two)  = -nan
sqrt(-zwei) = (8.65956e-17,-1.41421)
```

Parametrisierte Datentypen (Templates)



Mit der Templatevereinbarung können Datentypen und Funktionen vereinbart werden, die als Parameter Typen besitzen. Der auf `class` folgende Name steht für einen allgemeinen Typparameter, nicht nur für einen Klassentypparameter.

Voriges Beispiel mit Templates:

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

template <class T> class complex {
  private:
    T re,im;

  public:
    complex<T>(T Re=0, T Im=0): re(Re),im(Im) { }

    friend ostream& operator<<(ostream& stream, complex<T> z)
      { stream << "(" << z.re << "," << z.im << ")" ;
        return stream; }

    friend complex<T> operator-(complex<T> z)
      { return complex<T>(-z.re,-z.im); }

    friend complex<T> sqrt(complex<T> z)  // keineswegs optimal
      {  T r = pow(z.re*z.re+z.im*z.im,(T)0.25),
               phi = atan2(z.im,z.re)/2.0;
         return complex<T>(r*cos(phi),r*sin(phi)); }
};

int main()
{
  complex<float> two(2.0f,0.0f);
  complex<double> zwei(2.0,0.0);

  cout << "sqrt(-two)="  << sqrt(-two)  << "  "
       << "sqrt(-zwei)=" << sqrt(-zwei) << endl;

  return 0;
}
```

Ausg.: `sqrt(-two)=(-6.18172e-08,-1.41421)  sqrt(-zwei)=(8.65956e-17,-1.41421)`

[1] vereinfacht