Bsp.: Matrixelementzugriff mittels `()`

```
#include <iostream>
#include <valarray>
#include <iomanip>

using namespace std;

template <class T> class matrix {
  private:
    int m,n;
    valarray<T> a;
  public:
    matrix<T> (int m_=0, int n_=0): m(m_), n(n_), a(m_*n_) {}
    size_t nrow() const { return m; }
    size_t ncol() const { return n; }

  T  operator() (int i, int j) const
     { return a[i*n+j]; }
  T& operator() (int i, int j)
     { return a[i*n+j]; }
};

int main()
{
  const int m=3,n=4;
  matrix<double> a(m,n);
  cout << "nrow=" << a.nrow() << " ncol=" << a.ncol() << endl;
  for (int i=0; i<m; ++i)
    for (int j=0; j<n; ++j)
      a(i,j)=i*i+j;
  for (int i=0; i<m; ++i) {
    for (int j=0; j<n; ++j)
      cout << setw(4) << a(i,j);
    cout << endl;
  }
  cout << endl;
  return 0;
}
```

Ausgabe:

```
nrow=3 ncol=4
   0   1   2   3
   1   2   3   4
   4   5   6   7
```

Funktionsobjekte

Objekte einer Klasse, in der der Operator `()` überladen ist, werden oft als *Funktionsobjekte* oder *Funktoren* bezeichnet. Diese Bezeichnung wird vor allem verwendet, wenn der Hauptzweck der Klasse darin besteht, funktionsartige Aufrufe bereitzustellen.

Funktionsobjekte können zur Darstellung von Funktionen benutzt werden, die von verborgenen Parametern abhängen. Auf diese Weise können globale Variablen bzw. zusätzliche Parameter vermieden werden.

Bsp.: Normalverteilung $N(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$

```
#include <iostream>
#include <cmath>
#include <iomanip>

using namespace std;

class Normalverteilung {
  private:
    double m;
    double s;
  public:
    explicit Normalverteilung(double m_=0, double s_=1): m(m_), s(s_) {}
    double& mw() {return m;}
    double& stdabw() {return s;}
    double operator() (double x)
      { return exp(-0.5*(x-m)*(x-m)/(s*s))/(sqrt(2*M_PI)*s); }
};

int main()
{
  Normalverteilung gv, nv(2,3);
  cout << "gv:  mw=" << gv.mw() << " std=" << gv.stdabw() << " gv(0)=" << gv(0) << endl
       << "nv:  mw=" << nv.mw() << " std=" << nv.stdabw() << " nv(0)=" << nv(0) << endl;
  return 0;
}
```

Ausgabe:

```
gv:  mw=0 std=1 gv(0)=0.398942
nv:  mw=2 std=3 nv(0)=0.106483
```

Tatsächlich ist es sogar möglich, nur mit dem Konstruktor der Funktorklasse zu arbeiten.

```
int main()
{
  cout << "gv:    mw="   << Normalverteilung().mw()
       <<    "   std="   << Normalverteilung().stdabw()
       <<    " gv(0)="   << Normalverteilung()(0)          << endl
       << "nv:    mw="   << Normalverteilung(2,3).mw()
       <<    "   std="   << Normalverteilung(2,3).stdabw()
       <<    " nv(0)="   << Normalverteilung(2,3)(0)       << endl;
  return 0;
}
```

Spezielle Komponentenfunktionen

Typumwandlungen mittels Konstruktoren

Konstruktoren einer Klasse C, die für ein einzelnes Argument vom Typ T aufgerufen werden können, haben die Wirkung einer Typumwandlung $T \to C$.

Bsp.: `Complex(double x=0, double y=0) : re(x), im(y) { }  // double -> Complex`

Benutzerdefinierte Typumwandlungen werden auch implizit durchgeführt, in einer Folge von Umwandlungen jedoch nur eine davon.

Durch implizite Typumwandlungen können überraschende Effekte auftreten.

Bsp.:

```
         :
int main()
{
  fvektor a(3);
  a[1]=2;
  a.ausgeben("a");
  a=2;
  a.ausgeben("a");
  return 0;
}
```

Ausgabe:

```
a[1]=2 a[2]=0 a[3]=0
a[1]=0 a[2]=0
```

Das kann dadurch verhindert werden, daß der entsprechende Konstruktor als `explicit` definiert wird.

Bsp.: `explicit fvektor(int n_=1, int m_=1, double x=0.0) ...`

Auch die Standardbibliothek enthält aus diesem Grund häufig als `explicit` vereinbarte Konstruktoren.

Bsp.: `explicit vector<T>(size_type n) // verhindert implizites int -> vector<T>`

Typumwandlungen mittels Operatorfunktionen

Mit Konstruktoren sind Typumwandlungen von einer Klasse C in einen eingebauten Datentyp *nicht* möglich. Außerdem erfordern Typumwandlungen von einer neuen Klasse in eine bereits bestehende Klasse Änderungen an der bereits bestehenden Klasse.

Abhilfe schafft die Komponentenfunktion `operator` T`()`, für die *kein* Rückgabetyp (auch nicht `void`) angegeben und *kein* Parameter definiert werden darf. T steht hier für einen Typ- oder Klassennamen, es können noch ein oder mehrere `*` oder `&` folgen.

Benutzerdefinierte Typumwandlungen mittels Konstruktoren oder Operatorfunktionen müssen eindeutig sein. Auf jeden Wert werden sie jedoch höchstens einmal angewendet.

Bsp.: Typumwandlungen zwischen `fvektor` *und* `vector<double>`

```
            :
#include <vector>
#include <stdexcept>
            :

class fvektor {
  private:
    int m;      // min. Index
    int n;      // max. Index
    double *ap; // Zeiger auf die Komponenten
  public:
            :
    fvektor(const vector<double>& v)  // Konstruktor fuer Typumwandlung
      : m(0), n(v.size()-1)           // vector<double> -> fvektor
      { ap = new double[n+1];
        for (int i=0; i<=n; ++i)
          ap[i]=v[i]; }

    operator vector<double> ()        // Operatorfunktion fuer Typumwandlung
      { vector<double> v(n+1,0.0);    // fvektor -> vector<double>
        for (int i=m; i<=n; ++i)
          v[i]=ap[i-m];
        return v; }
};


int main()
{
  fvektor a;
  vector<double> b(4,2.0),c;
  a=b;                                // Typumw. von b: vector<double> -> fvektor
  a.ausgeben("a");
  a[2]=3;
  c=a;                                // Typumw. von a: fvektor -> vector<double>
  for (vector<double>::size_type i=0; i<c.size(); ++i)
    cout << "c[" << i << "]=" << c[i] << " ";
  cout << endl;
  return 0;
}
```

Ausgabe:

```
a[0]=2 a[1]=2 a[2]=2 a[3]=2
c[0]=2 c[1]=2 c[2]=3 c[3]=2
```

In der Standardbibliothek ist für Ein/Ausgabeströme die Funktion `operator void* () const` definiert, die die Umwandlungen `istream`→`void*` bzw. `ostream`→`void*` vornimmt. Diese liefert genau dann 0 als Zeigerwert, wenn ein Fehler oder Dateiende aufgetreten ist und ermöglicht so die Schleifen `while(cin>>s) { ... }` . Analog liefert `bool operator! () const` im Fall eines Fehlers oder Dateiendes `false` und sonst `true`.