Matrixtyp mit Elementzugriff per [ ][ ] - slice-basiert

Slice_arrays sind als Hilfsklassen im Zusammenhang mit Valarrays und Slices konzipiert, jedoch sind die zugrundeliegenden Valarrays und Slices nicht direkt aus dem Slice_array ablesbar. Diese Informationen sind aber erforderlich, um auf das k-te Element eines Slice_arrays z.B. mit einem Indexoperator zuzugreifen. Entsprechende Probleme treten im Zusammenhang mit Matrizen auf, die als lange Vektoren gespeichert werden. Für die Definition von Matrixoperationen kann man sich mit selbstdefinierten Slice_array-Varianten behelfen (vgl. Stroustrup 22.6.4, a.a.O).

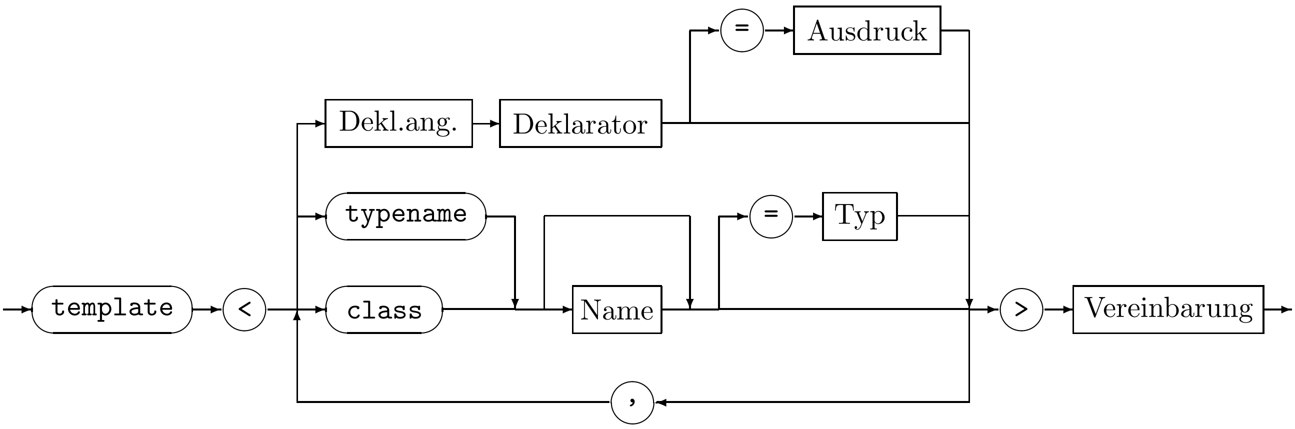
Bsp.: Matrixtyp mit selbstdefinierten Slicearray-Varianten

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

using namespace std;

template <class T> class myslice_array {
  private:
    slice s;
    valarray<T>& v;
  public:
    myslice_array<T> (int i0, int n, int h, valarray<T>& v_):
      s(slice(i0,n,h)), v(v_) {}
    T& operator[] (int j)
      { return v[s.start()+j*s.stride()]; }
};

template <class T> class mycslice_array {
  private:
    slice s;
    const valarray<T>& v;
  public:
    mycslice_array<T> (int i0, int n, int h, const valarray<T>& v_):
      s(slice(i0,n,h)), v(v_) {}
    T operator[] (int j) const
      { return v[s.start()+j*s.stride()]; }
};

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

    myslice_array<T> operator[] (int i)
      { return myslice_array<T>(i*n,n,1,v); }
    mycslice_array<T> operator[] (int i) const
      { return mycslice_array<T>(i*n,n,1,v); }
```

```
    myslice_array<T> zeile(int i)
      { return myslice_array<T>(i*n,n,1,v); }
    mycslice_array<T> zeile(int i) const
      { return mycslice_array<T>(i*n,n,1,v); }

    myslice_array<T> spalte(int j)
      { return myslice_array<T>(j,m,n,v); }
    mycslice_array<T> spalte(int j) const
      { return mycslice_array<T>(j,m,n,v); }
};

int main()
{
  int i,j,m,n;
  cout << "m n: "; cin >> m >> n;
  cout << "i j: "; cin >> i >> j;

  matrix<double> a(m,n);

  // Einlesen der Matrix a
  cout << endl << "Einlesen von Matrix a: " << endl;
  for (int i=0; i<m; ++i)
    for (int j=0; j<n; ++j)
       cin >>  a[i][j];

  // Ausgabe der Zeile i
  cout << endl << "Zeile  " << i << ": ";
  for (int k=0; k<n; ++k) cout << setw(3) << a.zeile(i)[k];

  // Ausgabe der Spalte j
  cout << endl << "Spalte " << j << ": ";
  for (int k=0; k<m; ++k) cout << setw(3) << a.spalte(j)[k];

  // Ausgabe der Matrix b (Kopie von a)
  const matrix<double> b(a);
  cout << endl << endl << "Matrix b (Kopie von a): " << endl;
  cout << "nzeil=" << b.nzeil() << " nspalt=" << b.nspalt() << endl;
  for (int i=0; i<m; ++i) {
    for (int j=0; j<n; ++j)
      cout << setw(4) << b[i][j];
    cout << endl;
  }

  return 0;
}
```

Parametrisierte Datentypen (Templates)

Template-Vereinbarung [1]



Sie dient zur Vereinbarung parametrisierter Klassen oder Funktionen, je nachdem, ob in der Vereinbarung am Ende des Syntaxdiagramms eine Klasse oder eine Funktion vereinbart wird. Der Name in der Vereinbarung am Ende des Syntaxdiagramms ist der Template-Name.

Die Template-Vereinbarung ist außerhalb von Funktionen und mit Einschränkungen innerhalb von Klassen möglich.

Bsp.: Datentyp array aus C++98/TR1 `<tr1/array>` *(Auszug)*

```
using namespace std;
// array verkapselt C-Vektoren fester Laenge und
// stellt einige STL-Behaelter-Funktionen bereit

template <typename T, size_t N> struct array {

  typedef size_t size_type;

  typedef T value_type;
  typedef value_type& reference;
  typedef const value_type& const_reference

  typedef value_type* iterator;
  typedef const value_type* const_iterator;
                      :
  // Interner C-Vektor instance hat mindestens Laenge 1 (vorgeschrieben)
  value_type instance[N ? N : 1];

  size_type size() const { return N; }
  bool empty() const { return size() == 0; }
                      :
  // Indexoperatoren
  reference operator[](size_type n) { return instance[n]; }
  const_reference operator[](size_type n) const { return instance[n]; }

  // Iteratoren
```

[1] vereinfacht

```
  iterator begin() { return iterator(&instance[0]); }
  iterator end()   { return iterator(&instance[N]); }
                       :
};

// Vergleichsfunktionen
template<typename T, size_t N> inline bool operator==(const array<T,N>& a,
                                                       const array<T,N>& b)
  { return equal(a.begin(),a.end(),b.begin()); }
template<typename T, size_t N> inline bool operator!=(const array<T,N>& a,
                                                       const array<T,N>& b)
  { return !(a==b) ; }
                       :
```

Der auf `class` oder `typename` folgende Name steht für einen allgemeinen Typparameter, *nicht nur* für einen Klassentypparameter. Die Verwendung von `typename` anstelle von `class` macht keinen Unterschied.

Andere Template-Parameter müssen einen ganzzahligen oder Zeigertyp besitzen.

Innerhalb einer Template-Definition genügt der Template-Name (ohne Template-Parameter) als Typangabe.

Komponentenfunktionen in Klassentemplates sind im Unterschied zu befreundeten Funktionen automatisch Funktionstemplates und werden *ohne* Template-Parameter vereinbart.

Bsp.: `size() const {...}` statt `size<T,N>() const {...}`

Optionale Zuweisungen legen Voreinstellungen für die Template-Parameter fest.

Allerdings muss in Template-Vereinbarungen den Typnamen, die von einem Template-Parameter abhängen oder mit Hilfe von `::` gebildet werden, das Schlüsselwort `typename` vorangestellt werden (vgl. Funktion `two_norm` unten)

Bsp.: Funktionstemplate für Datentyp array

```
#include <iostream>
#include <cmath>
#include <array>

using namespace std;

template<typename T,size_t N> T two_norm(const array<T,N>& x)
{
  T s = 0;
  for (typename array<T,N>::const_iterator pos=x.begin(); pos!=x.end(); ++pos)
    s += (*pos)*(*pos);
  return sqrt(s);
}

int main()
{
  array<float,2>  x = {3,4};
  array<double,4> y = {1,3,5,1};
  cout << "two_norm(x) = " << two_norm(x) << endl;
  cout << "two_norm(y) = " << two_norm(y) << endl;
  return 0;
}
```