Für Numerik optimierte Vektoren (`<valarray>`)

Vereinbarung und Operationen

Im folgenden steht T für den Valarraykomponententyp, t für ein Objekt vom Datentyp T, *ptr* für einen Zeiger auf konstantes T (`const T*`), i und n sind *vorzeichenlose* ganze Zahlen (vom Typ `size_t`), k eine ganze Zahl vom Typ `int` und f eine Funktion mit Parameter vom Typ T.

Operation	*Wirkung*
`valarray<T> a`	vereinbart leeres Valarray
`valarray<T> a(n)`	vereinbart Valarray aus n Komponenten, jeweils nach Voreinstellung für T initialisiert
`valarray<T> a(t,n)`	vereinbart Valarray aus n Komponenten, jeweils mit t (vom Datentyp T) initialisiert
`valarray<T> a(ptr,n)`	vereinbart Valarray aus n Komponenten, initialisiert mit den ersten n Werten, auf die *ptr* zeigt
`valarray<T> a=`$\{t_0 \ldots, t_{n-1}\}$[*]	vereinb. Valarray a mit Dim. n und Komp.werten $t_0, \ldots, t_{n-1}$
`valarray<T> a`$\{t_0 \ldots, t_{n-1}\}$[*]	“
`valarray<T> a(b)`	vereinb. Valarray a mit Dim. und Komp.werten von b
`a[i]`	Komp.wert zum Index i (ganzzahlig, vorzeichenlos)
`a=`$\{t_0 \ldots, t_{n-1}\}$[*]	Zuweisung: $a_i = t_i$ $(i = 0, \ldots, n-1)$, n Dimension von a
`a=b a=t`	Zuweisung (komponentenweise)
`a==b a!=b a>b a<b a<=b a>=b`	Vergleiche komponentenweise
`a==t a!=t a>t a<t a<=t a>=t`	wie oben
`t==a t!=a t>a t<a t<=a t>=a`	wie oben
`a+b a-b a*b a/b a%b +a -a`	arithmetische Operationen komponentenweise
`a+t a-t a*t a/t a%t`	wie oben
`t+a t-a t*a t/a t%a`	wie oben
`a+=b a-=b a*=b a/=b a%=b`	wie oben
`a+=t a-=t a*=t a/=t a%=t`	wie oben
`a&b a|b a^b ~a a<<b a>>b a&&b a||b !a`	Bitoperatoren und logische Operatoren komp.weise
`a&t a|t a^t a<<t a>>t a&&t a||t`	wie oben
`t&a t|a t^a t<<a t>>a t&&a t||a`	wie oben
`a&=b a|=b a^=b a>>=b a<<=b`	wie oben
`a&=t a|=t a^=t a>>=t a<<=t`	wie oben
`sin(a) cos(a) tan(a)`	trigonometrische Funktionen komp.weise
`asin(a) acos(a) atan(a)`	trigonometrische Umkehrfunktionen komp.weise
`exp(a) sinh(a) cosh(a) tanh(a)`	Exp.fkt. und hyperbolische Funktionen komp.weise
`abs(a) sqrt(a) log(a) log10(a)`	Betrag-, Wurzel- und logarithm. Funktionen komp.weise
`pow(a,b) pow(a,t) pow(t,a)`	Potenzfunktion komponentenweise
`atan2(a,b) atan2(a,t) atan2(t,a)`	Argumentfunktion komponentenweise
`a.size()`	Anzahl der Komponenten
`a.min() a.max()`	kleinster bzw. größter Komponentenwert
`a.sum()`	Summe (eines nichtleeren Valarrays)
`a.shift(k)`	liefert um k Positionen verschobenes Valarray nachgeschoben werden nach Voreinst. für T init. Werte
`a.cshift(k)`	liefert um k Positionen rotiertes Valarray
`a.apply(f)`	liefert Valarray, auf das komp.weise f angewendet wurde
`a.resize(n)`	ändert Komponentenzahl, initialisiert *alle* Komp. neu (!!)
`a.resize(n,t)`	wie `a.resize(n)`, jedoch Initialisierung mit t

Vorsicht: Andere Argumentreihenfolge in `valarray<T> a(t,n)` als in `vector<T> a(n,t)`!

[*] C++11

Beispiel:

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

using namespace std;

int main()
{
  size_t i;
  valarray<double> a(5), b={1.0,2.0,3.0,4.0,5.0};
  cout << "a: ";
  for (i=0; i<5; ++i)
    cin >> a[i];
  cout << "skalar(a,b) = " << (a*b).sum() << endl;
  return 0;
}
```

Indexmengen (`<valarray>`)

Die Datentypen `slice` und `gslice` dienen zur Beschreibung von Indexmengen von Valarrays. Für diesen Zweck können auch boolesche Valarrays und Valarrays vom Indextyp (`size_t`) eingesetzt werden.

Im folgenden bezeichnet a ein Valarray, s ein Slice, g ein verallgemeinertes Slice, bv ein Valarray vom Komponententyp `bool` und iv ein Valarray vom Komponententyp `size_t`. $i0$, h und n sind vorzeichenlose ganze Zahlen vom Typ `size_t`, hv und nv Valarrays vom Komponententyp `size_t` der Länge L.

Operation	*Wirkung*
`slice s(i0,n,h)`	erzeugt die Beschreibung der Indexmenge $\{i_0 + kh : k = 0, \ldots, n-1\}$
`s.start()`	liefert i_0
`s.size()`	liefert n
`s.stride()`	liefert h
`a[s]`	liefert Valarray zur Indexmengenbeschreibung s bzw. ermöglicht Lese/Schreibzugriff auf die durch s bezeichneten Komp. von a
`gslice g(i0,nv,hv)`	erzeugt die Beschreibung der Indexmenge $\{i_0 + k_0h_0 + \ldots + k_{L-1}h_{L-1} : k_{L-1} = 0, \ldots, n_{L-1}-1; \ldots ; k_0 = 0, \ldots, n_0 - 1\}$ k_{L-1} variiert am schnellsten, k_0 am langsamsten!
`g.start()`	liefert i_0
`g.size()`	liefert nv (Komp.: $n_0,\ldots,n_{L-1}$)
`g.stride()`	liefert hv (Komp.: $h_0,\ldots,h_{L-1}$)
`a[g]`	liefert Valarray zur Indexmengenbeschreibung g bzw. ermöglicht Lese/Schreibzugriff auf die durch g bezeichneten Komp. von a
`a[bv]`	liefert Valarray zu den Indizes, die die auf `true` gesetzten Komp. von bv bezeichnen bzw. ermöglicht Lese/Schreibzugriff auf die entsprechenden Komponenten von a
`a[iv]`	liefert Valarray zu den Indizes, die als Komponenten in `iv` gespeichert sind bzw. ermöglicht Lese/Schreibzugriff auf die entsprechenden Komponenten von a

Matrixtyp mit Elementzugriff per [ ][ ] ohne Slices

Bsp.: Matrixtyp ohne Slices

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

using namespace std;

template<class T> class zeile {
  private:
    int i,n;                        // Zeilenindex, Spaltenzahl
    valarray<T>& v;
  public:
    zeile<T>(int i_,int n_,valarray<T>& v_) : i(i_),n(n_),v(v_) { }
    T& operator[](int j)       { return v[i*n+j]; }
};

template<class T> class konstzeile {
  private:
    int i,n;                        // Zeilenindex, Spaltenzahl
    const valarray<T>& v;
  public:
    konstzeile<T>(int i_,int n_,const valarray<T>& v_) : i(i_),n(n_),v(v_) { }
    T  operator[](int j) const { return v[i*n+j]; }
};

template<class T> class matrix {
  private:
    int m,n;                        // Zeilenzahl, Spaltenzahl
    valarray<T> v;
  public:
    matrix<T>(int m_=0, int n_=0): m(m_), n(n_), v(m_*n_) {}
    zeile<T>      operator[](int i)       { return      zeile<T>(i,n,v); }
    konstzeile<T> operator[](int i) const { return konstzeile<T>(i,n,v); }
    size_t nzeil()  const { return m; }
    size_t nspalt() const { return n; }
};

int main()
{
  const int m=3,n=4;
  matrix<double> a(m,n);
  cout << "nzeil=" << a.nzeil() << " nspalt=" << a.nspalt() << 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;}

  const matrix<double> b(a);
  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;
}
```

Hilfsklasse slice_array

Slice_arrays sind Hilfsklassen, um auf die durch Slices spezifierten Komponenten von Valarrays zuzugreifen. Insbesondere wird damit die Zuweisung `a[s] = b` für Valarrays `a`, `b` und Slices `s` möglich. Ein Slice_array enthält typischerweise die Datenkomponenten eines Slices und einen Verweis auf ein Valarray (vgl. Stroustrup - Die C++-Programmiersprache (2000) 22.6.4).
Für Slice_arrays gibt es *keinen* Standardkonstruktor, statt dessen werden Sie durch `a[s]` (oder über andere Slice_arrays) erzeugt.

Deklarationen für Valarrays, Slice_arrays und Slices

```
template<class T> class valarray {
  public:
    valarray<T>();                                 // Standardkonstruktor
    explicit valarray<T>(size_t);                  // Konstruktor:  valarray<T> a(n);
    valarray<T>(const valarray<T>&);               // Kopierkonstruktor
    valarray<T>(const slice_array<T>&);            // Typumw. slice_array->valarray
          :
    valarray<T>& operator=(const valarray<T>&);    // a = b
    valarray<T>& operator=(const slice_array<T>&); // a = s_b
          :
    T  operator[](size_t) const;                   // a[i] fuer konst. a
    T& operator[](size_t);                         // a[i]
    valarray<T> operator[](slice) const;           // a[s] fuer konst. a
    slice_array<T> operator[](slice);              // a[s]
          :
};

class slice {
  public:
    slice();                                       // Standardkonstruktor
    slice(size_t, size_t, size_t);                 // Konstruktor: slice s(x0,n,h);
    slice(const slice&);                           // Kopierkonstruktor
    size_t start()  const;                         // Liefert x0
    size_t size()   const;                         // Liefert n
    size_t stride() const;                         // Liefert h
};

template<class T> class slice_array {
  public:
    slice_array<T>() = delete;                     // Kein Standardkonstruktor
    slice_array<T>(const slice_array<T>&);         // s_a(t_b)
    slice_array<T>& operator=(const slice_array<T>&) const; // s_a = t_b, aendert die
                                                   // Komponenten, auf die s_a verweist
    void operator= (const valarray<T>&) const;     // s_a =  b
    void operator= (const T&);                     // s_a =  t
    void operator+=(const valarray<T>&) const;     // s_a += b
    void operator*=(const valarray<T>&) const;     // s_a *= b
          :
};
```