Auf Numerik optimierte Vektoren (`valarray`)

Objekte vom Datentyp `valarray<`T`>` speichen beliebig viele Elemente vom Typ T *dicht*, d.h. als in einem großen Speicherbereich direkt aufeinander folgend. Die Werte, die T annehmen darf sind eingeschränkt auf eingebaute Zahlentypen wie `double`, `float`, `int`, etc., Zeigertypen, `complex` und `valarray<`T'`>`.

`valarray<`T`>`s unterstützen arithmetische Operationen punktweise mit einem Skalar vom Typ T und punktweise mit einem weiteren `valarray<`T`>` *gleicher Größe*.

Zudem unterstützen `valarrays` das bilden von Teilvektoren mit Referenzsemantik, was sie insb. für die Verwendung als Datentyp zu Darstellung von Matrizen attraktiv macht. Teilvektoren wären in diesem Kontext z.B. Zeilen, Spalten und Untermatrizen.

Im Folgenden bezeichnet T einen Datentyp, t, $t_0, \ldots, t_{n-1}$ Werte vom Typ T, n, i natürliche Zahlen vom Typ `std::size_t`, i eine ganze Zahl vom Typ `int`, v, v' Variablen vom Typ `valarray<`T`>`, f ein Zeiger auf Funktion vom Typ $T'(T)$

`valarray<`T`> v`	Vereinbart Variable v vom Typ `valarray<`T`>` leer (Länge 0)
`valarray<`T`> v(`n`)`	Vereinbart Variable v vom Typ `valarray<`T`>` mit Länge n und allen Elementen initialisiert mit Standardkonstruktor
`valarray<`T`> v(`t`, `n`)`	Vereinbart Variable v vom Typ `valarray<`T`>` mit Länge n und allen Elementen initialisiert als Kopien von t[1]
`valarray<`T`> v{`t_0`, ..., `t_{n-1}`}`	Vereinbart Variable v vom Typ `valarray<`T`>` mit Elementen $t_0, \ldots, t_{n-1}$
`v[`i`]`	Liefert Referenz auf Element mit Index i in v

`v == `v' `v != `v' `v > `v' `v < `v'
`v >= `v' `v <= `v'
`t == v t != v t > v t < v t >= v`
`t <= v`
`v == t v != t v > t v < t v >= t`
`v <= t`
	Liefert Wert vom Typ `valarray<bool>` mit den Ergebnissen des komponentenweisen Vergleichs der Element von v mit denen von v' bzw. von t jeweils mit den Elementen von v

`v + `v' `v - `v' `v * `v' `v / `v' `v % `v'
`v & `v' `v | `v' `v ^ `v' `v << `v'
`v >> `v' `v && `v' `v || `v' `v += `v'
`v -= `v' `v *= `v' `v /= `v' `v %= `v'
`v &= `v' `v |= `v' `v ^= `v' `v <<= `v'
`v >>= `v'
`t + v t - v t * v t / v t % v`
`t & v t | v t ^ v t << v t >> v`
`t && v t || v`
`v + t v - t v * t v / t v % t`
`v & t v | t v ^ t v << t v >> t`
`v && t v || t v += t v -= t`
`v *= t v /= t v %= t v &= t`
`v |= t v ^= t v <<= t v >>= t`
	Liefert `valarray` mit Ergebnissen der komponentenweisen Anwendung des arith. bzw. bitweisen bzw. logischen Operators der Elemente von v mit denen von v' bzw. von t jeweils mit den Elementen von v

[1] Die Reihenfolge der Parameter ist hier anders als bei anderen container-Datentypen

`abs(`v`)` `exp(`v`)` `log(`v`)` `log10(`v`)` `sqrt(`v`)` `sin(`v`)` `cos(`v`)` `tan(`v`)` `asin(`v`)` `acos(`v`)` `atan(`v`)` `sinh(`v`)` `cosh(`v`)` `tanh(`v`)`	Liefert `valarray` mit Ergebnissen der komponentenweisen Anwendung der arith. Funktion auf die Elemente von v
`pow(`v`,` v'`)` `pow(`v`,` t`)` `pow(`t`,` v`)` `atan2(`v`,` v'`)` `atan2(`v`,` t`)` `atan2(`t`,` v`)`	Liefert `valarray` mit Ergebnissen der komponentenweisen Anwendung der arith. Funktion auf die Elemente von v und die Elemente von v' bzw. von t jeweils mit den Elementen von v
v`.sum()` v`.min()` v`.max()`	Liefert Summe bzw. Minimum bzw. Maximum der Elemente von v
v`.shift(`i`)`	Liefert `valarray<`T`>` mit Kopien der Elemente von v, verschoben um i Stellen; füllt von den Enden her mit 0 auf.
v`.cshift(`i`)`	Liefert `valarray<`T`>` mit Kopien der Elemente von v, verschoben um i Stellen; verschiebt zyklisch.
v`.apply(`f`)`	Liefert `valarray<`T'`>` mit den Ergebnissen der komponentenweisen Anwendung von f auf die Elemente von v
v`.resize(`n`)` v`.resize(`n`,` t`)`	Ersetzt v durch ein neues valarray der Länge n mit allen Elementen initialisiert mit Standardkonstruktor bzw. als Kopie von t
`begin(`v`)` `end(`v`)`	Liefert Iterator nicht näher spezifizierten Typs

Beispiel. Wir berechnen das Skalarprodukt eines Vektors von `double` dargestellt als `valarray<double>`.

valarray_scalar.cpp

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

using namespace std;

int main() {
  valarray<double> a(5), b{1.0, 2.0, 3.0, 4.0, 5.0};

  cout << "a: ";
  for (double& x: a) cin >> x;

  cout << "skalar(a, b) = " << (a*b).sum() << endl;
}
```

```
a: 5 4 3 2 1
skalar(a, b) = 35
```

Indexmengen (`slice` etc.) & Teilvektoren (`slice_array` etc.)

Objekte der Klassen `slice`, `gslice` und `valarray<bool>` und `valarray<size_t>` können als Indexmengen fungieren. Der Subscriptoperator `[]` ist für `valarray` geeignet überladen, sodass auch Indexmengen als Parameter fungieren können. Zurückgegeben wird dann ein Teilvektor von einem mit `valarray` verwanden Typ (`slice_array`, `gslice_array`, `mask_array` bzw. `indirect_array`) der Referenzsemantik hat auf die Elemente des `valarray`. D.h. Zuweisungsoperatoren mit einem Teilvektor auf der linken Seite verändern die im an den Subscriptoperater übergebenen `valarray` gespeicherten Werte.

Zudem können Teilvektoren implizit konvertiert werden in Werte vom Typ `valarray` sodass z.B. auch die Verwendung auf der rechten Seite von Zuweisungsoperatoren möglich ist. Es wird jedoch bei der Konvertierung zu `valarray` jedes mal eine Kopie aller Elemente gemacht, was einen großen performance-Einfluß haben kann.

Bitmasken (`mask_array`) & Indirekte Indizierung (`indirect_array`)

Bei Verwendung eines `valarray<bool>`, einer *Bitmaske*, als Indexmenge muss der `valarray<bool>` von selber Größe sein wie das `valarray` auf das der Subscriptoperator später angewandt wird. Der zurückgegebene Typ `mask_array` enthält eine Referenz auf genau jene Werte bei denen am selben Index in der Indexmenge der Wert `true` steht.

Bei Verwendung eines `valarray<size_t>` als Indexmenge enthält der zurückgegebene Typ Referenzen auf genau jene Stellen im `valarray`, auf das der Subscriptoperator angewandt wurde, die im `valarray<size_t>` stehen in der Reihenfolge in der sie in `valarray<size_t>` stehen. Das mehrfache Vorkommen des selben Index in dem `valarray<size_t>` ist nicht zulässig.

Beispiel. Wir setzen mithilfe einer Bitmaske die vier Ecken einer Matrix (zeilenweise gespeichert in einem `valarray<double>`) auf 0 und geben die Diagonale der resultierenden Matrix aus.

misc_slices.cpp

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

using namespace std;

void ausgabe_vektor(const valarray<double>& a) {
  for (double x: a)
    cout << setw(3) << x;
}
void ausgabe_matrix(const valarray<double>& a, size_t n) {
  for (size_t k = 0; k < a.size(); k++) {
    cout << setw(3) << a[k];
    if ((k+1) % n == 0)
      cout << endl;
  }
}

int main() {
```

```
  size_t m, n;
  cout << "m n: "; cin >> m >> n;

  valarray<double> a(m*n);

  for (size_t k = 0; k < a.size(); k++) a[k] = k;

  valarray<bool> bv(m*n);
  bv = false;
  bv[0] = bv[n - 1] = bv[n * (m-1)] = bv[m*n - 1] = true;
  a[bv] = 0;
  cout << "a:" << endl;
  ausgabe_matrix(a, n); cout << endl;

  valarray<size_t> iv(min(m, n));
  for (size_t i = 0; i < min(m, n); i++)
    iv[i] = i*(n + 1);
  cout << "diag(a):";
  ausgabe_vektor(a[iv]); cout << endl;
}
```

```
m n: 4 3
a:
  0  1  0
  3  4  5
  6  7  8
  0 10  0

diag(a):  0  4  8
```

`slice`

`slice(`i_0`, `n`, `h`)` liefert ein Objekt vom Typ `slice`, das die Indexmenge $(i_0 + kh)_{k=0,\dots,n-1}$ darstellt.

Die Klasse `slice` implementiert die Methoden `start()`, `size()` und `stride()` um die im Konstruktor gegebenen Werte wieder auslesen zu können.

Beispiel. Für eine Matrix $(a_{ij})_{\substack{i=0,\dots,m-1\\ j=0,\dots,n-1}}$, die zeilenweise in einer Variable a vom Typ `valarray` gespeichert ist, d.h. `a[`$i \cdot n + j$`]` entspricht a_{ij}:

i-te Zeile Die Index-Folge $(i \cdot n + j)_{j=0,\dots,n-1}$ entspricht `slice(`$i \cdot n$`, `n`, 1)`.

j-te Spalte Die Index-Folge $(i \cdot n + j)_{i=0,\dots,n-1}$ entspricht `slice(`j`, `m`, `n`)`.

Beispiel. Wir geben die i-te Zeile und die j-te Spalte einer Matrix aus. Danach addieren wir die l-te Zeile auf die j-te Zeile und geben die resultierende Matrix aus.

slice.cpp

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

using namespace std;

slice zeile(size_t i, size_t m, size_t n)
  { return slice(i * n, n, 1); }
slice spalte(size_t j, size_t m, size_t n)
  { return slice(j, m, n); }

void ausgabe_vektor(const valarray<double>& a) {
  for (double x: a)
    cout << setw(3) << x;
}
void ausgabe_matrix(const valarray<double>& a, size_t n) {
  for (size_t k = 0; k < a.size(); k++) {
    cout << setw(3) << a[k];
    if ((k+1) % n == 0)
      cout << endl;
  }
}

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

  valarray<double> a(m*n);

  for (size_t k = 0; k < a.size(); k++) a[k] = k;

  cout << "Zeile  " << i << ": ";
  ausgabe_vektor(a[zeile(i, m, n)]); cout << endl;
  cout << "Spalte " << j << ": ";
  ausgabe_vektor(a[spalte(j, m, n)]); cout << endl;

  a[zeile(i, m, n)] += a[zeile(l, m, n)];

  cout << "a:" << endl;
  ausgabe_matrix(a, n); cout << endl;
}
```

```
m n: 3 4
i j l: 1 2 0
Zeile  1:   4  5  6  7
Spalte 2:   2  6 10
a:
  0  1  2  3
  4  6  8 10
  8  9 10 11
```

gslice

Bei `gslice` handelt es sich um die mehrdimensionale Verallgemeinerung von `slice`. Im Gegensatz zu `slice` nimmt `gslice` als zweiten und dritten Parameter jeweils ein `valarray<size_t>` von Längen und Schrittweiten.

Ein Aufruf von `gslice(`i_0`, {`n_0`, `n_1`, ...}, {`h_0`, `h_1`, ...})` liefert die Indexmenge

$$(i_0 + k_0 h_0 + \ldots + k_{L-1} h_{L-1})_{\substack{k_{L-1}=0,\ldots,n_{L-1} \\ \vdots \\ k_0=0,\ldots,n_0-1}}$$

wobei k_{L-1} „am schnellsten variiert“ und k_0 „am langsamsten“.

Anders ausgedrückt genügt `gslice` der folgenden Rekursionsbeziehung:

- `gslice(`i_0`, {`n`}, {`h`})` liefert: `slice(`i_0`, `n`, `h`)`
- `gslice(`i_0`, {`n_0`, `n_1`, ...}, {`h_0`, `h_1`, ...})` liefert:
  `gslice(`$i_0 + 0 \cdot h_0$`, {`n_1`, ...}, {`h_1`, ...})` $\circ$ `gslice(`$i_0 + 1 \cdot h_0$`, {`n_1`, ...}, {`h_1`, ...})` $\circ \ldots \circ$
  `gslice(`$i_0 + (n_0 - 1) \cdot h_0$`, {`$n_1$`, ...}, {`$h_1$`, ...})`

wobei $\circ$ die Konkatenation von Folgen von Indizes bezeichne.

Beispiel. Wir wandeln ein `gslice_array`, das den oberen linken Teil einer 2×2-Blockmatrix referenziert, implizit in ein `valarray` um und geben dieses aus.

gslice.cpp

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

using namespace std;

gslice submatrix(size_t mp, size_t np, size_t n)
  { return gslice(0, {mp, np}, {n, 1}); }

void ausgabe_matrix(const valarray<double>& a, size_t n) {
  for (size_t k = 0; k < a.size(); k++) {
    cout << setw(3) << a[k];
    if ((k+1) % n == 0)
      cout << endl;
  }
}

int main() {
  size_t m, n, mp, np;
  cout << "m n: "; cin >> m >> n;
  cout << "mp np: "; cin >> mp >> np;

  valarray<double> a(m*n);
  for (size_t k = 0; k < a.size(); k++) a[k] = k;

  cout << "a:" << endl;
  ausgabe_matrix(a, n); cout << endl;
```

```
  cout << "sub a:" << endl;
  ausgabe_matrix(a[submatrix(mp, np, n)], np); cout << endl;
}
```

```
m n: 3 4
mp np: 2 3
a:
  0  1  2  3
  4  5  6  7
  8  9 10 11

sub a:
  0  1  2
  4  5  6
```

Matrix-Datentypen mit `valarray`

`valarray` ist zwar a priori kein Matrix-Datentyp, eignet sich aber sehr gut zur Implementierung eben dieser.

Beispiel. Wir vereinbaren einen eigenen Matrix-Datentyp der insb. den Zugriff auf einzelne Elemente durch doppelte Anwendung des Subscriptoperators erlauben soll. D.h. für A eine Matrix soll $A[i][j]$ eine Referenz auf den j-ten Eintrag der i-ten Zeile der Matrix A liefern.

Es genügt hierfür nicht den vordefinierten Typ `slice_array` für Zeilen der Matrix zu verwenden, da dieser nicht wiederum selbst den Subscriptoperator überlädt.

Es ist zudem keine gute Idee die Klasse `matrix<T>` von `valarray<T>` abzuleiten, da `valarray<T>` die Methode `resize` (nicht virtuell) implementiert und diese dazu verwendet werden könnte einen bzgl. der Größe des enthaltenen `valarray` und den beiden anderen Attributen `m` und `n` invaliden Zustand zu erreichen.

Wir beginnen mit den Deklarationen für den Matrix-Typ, die wir in eine separate Header-Datei auslagern:

matrix.h

```
#pragma once

#include <valarray>

template<class T> class matrix {
  private:
    std::valarray<T> v;
    std::size_t m, n;

  public:
    matrix(std::valarray<T>&&, std::size_t, std::size_t);
    matrix(std::size_t m_ = 0, std::size_t n_ = 0, T val_ = T());

    size_t rows() const; size_t columns() const;
```

```
    class matrix_slice : public std::slice {
      friend class matrix;
      private:
        std::valarray<T>& v;

        matrix_slice(std::valarray<T>& v_, std::size_t start, std::size_t size, std::size_t
        ↪  stride);

      public:
        T& operator[](std::size_t i);
        T operator[](std::size_t i) const;
        operator std::valarray<T>() const;
    };

    matrix_slice operator[](std::size_t i);
    matrix_slice row(std::size_t i);
    matrix_slice column(std::size_t j);

    operator std::valarray<T>() const;

    T* begin();
    T* end();
};
```

Die Definitionen für den Matrix-Typ:

matrix.cpp

```
#include <valarray>
#include <stdexcept>

#include "matrix.h"

using namespace std;

template<class T>
  matrix<T>::matrix(std::valarray<T>&& v_, std::size_t m_, std::size_t n_)
    : v(v_), m(m_), n(n_) {
      if (v.size() != m * n)
        throw invalid_argument("valarray of wrong size");
}
template<class T>
  matrix<T>::matrix(size_t m_, size_t n_, T val_)
    : v(val_, m_ * n_),m(m_),n(n_) {}

template<class T>
  size_t matrix<T>::rows() const { return m; }
template<class T>
  size_t matrix<T>::columns() const { return n; }

template<class T>
  matrix<T>::matrix_slice::matrix_slice
    (valarray<T>& v_, size_t start, size_t size, size_t stride)
    : slice(start, size, stride), v(v_) {}

template<class T>
```

```
  T& matrix<T>::matrix_slice::operator[](size_t i)
    { return v[start() + i*stride()]; }
template<class T>
  T matrix<T>::matrix_slice::operator[](size_t i) const
    { return v[start() + i*stride()]; }
template<class T>
  matrix<T>::matrix_slice::operator valarray<T>() const {
    valarray<T> v(size());
    for (size_t i = 0; i < size(); i++)
      v[i] = (*this)[i];
    return v;
  }

template<class T>
  typename matrix<T>::matrix_slice
    matrix<T>::operator[](size_t i) { return row(i); }
template<class T>
  typename matrix<T>::matrix_slice matrix<T>::row(size_t i)
    { return matrix<T>::matrix_slice{v, i*n, n, 1}; }
template<class T>
  typename matrix<T>::matrix_slice matrix<T>::column(size_t j)
    { return matrix<T>::matrix_slice{v, j, m, n}; }

template<class T>
  matrix<T>::operator valarray<T>() const {
    return v;
  }

template<class T>
  T* matrix<T>::begin()
    { return std::begin(v); }
template<class T>
  T* matrix<T>::end()
    { return std::end(v); }

template class matrix<double>;
```

Wir können den so eingeführten Matrix-Typ verwenden um auf einzelne Elemente der Matrix (auch schreibend) zuzugreifen:

matrix_basic.cpp

```
#include <iostream>

#include "matrix.h"

using namespace std;

int main() {
  size_t m, n, i, j;
  cout << "m n: "; cin >> m >> n;
  cout << "i j: "; cin >> i >> j;
  matrix<double> a{m, n};
  cout << "Matrix a: " << endl;
  for (double& x: a) cin >> x;
  cout << (a[i][j] = 13) << endl;
```

```
}
```

```
m n: 3 4
i j: 1 2
Matrix a:
  1  2  3  4
  5  6  7  8
  9 10 11 12
13
```

Wir können die Multiplikation von Werten unseres Matrix-Typs (naiv) implementieren:

matrixmul.cpp

```cpp
#include <iostream>
#include <iomanip>

#include "matrix.h"

using namespace std;

int main() {
  size_t m, n;
  cout << "m n: "; cin >> m >> n;
  matrix<double> a{m, n};
  cout << "Matrix a: " << endl;
  for (double& x: a) cin >> x;
  cout << endl;

  size_t k, l;
  cout << "k l: "; cin >> k >> l;
  matrix<double> b{k, l};
  cout << "Matrix b: " << endl;
  for (double& x: b) cin >> x;
  cout << endl;

  matrix<double> c{m, l};
  for (size_t i = 0; i < m; i++)
    for (size_t j = 0; j < l; j++) {
      valarray<double> as = a.row(i), bs = b.column(j);
      c[i][j] = (as * bs).sum();
    }

  for (size_t i = 0; i < m; i++) {
    for (size_t j = 0; j < l; j++)
      cout << setw(3) << c[i][j];
    cout << endl;
  }
}
```

```
m n: 4 4
Matrix a:
  3  2  1  4
  1  0  2  3
```

```
  3  2  1  2
  3  2  1  4

k l: 4 4
Matrix b:
  1  2  1  4
  0  1  0  3
  4  0  4  2
  1  2  1  4

 11 16 11 36
 12  8 12 20
  9 12  9 28
 11 16 11 36
```