34

Optimierte numerische Arrays (valarray)

Dieses Kapitel behandelt die folgenden Themen:

- valarray – Eine Klasse für optimierte numerische Arrays
- Beispiele für die verschiedenen Anwendungsmöglichkeiten

Der Header `<valarray>` schließt die Template-Klasse `valarray` ein, die für mathematische Vektor-Operationen gedacht ist. Ein Objekt der Klasse `vector` ist ein Container zur bequemen und flexiblen Verwaltung von Objekten, die ganz verschiedenen Typen angehören können. Ein `valarray`-Objekt hingegen ist ein für numerische Berechnungen optimierter Vektor – mit der Auswirkung, dass er für den Benutzer etwas umständlicher zu benutzen ist, um den Compiler-Herstellern möglichst viele Freiheiten bei der Optimierung zu ermöglichen. Im Header `<valarray>` sind auch die Klassen `slice` und `gslice` definiert, mit denen ein Ausschnitt aus einem Valarray bzw. die Struktur einer Matrix abgebildet werden kann, sowie weitere Template-Hilfsklassen (siehe Seite 864 ff.). Anwendungsbeispiele zur Klasse `valarray` finden sich in den Beispielen im Verzeichnis *cppbuch/k34*.

34.1 Konstruktoren

Die Konstruktoren ähneln teilweise denen der `vector`-Klasse.

- `valarray()`
  konstruiert ein Valarray der Größe 0.
- `explicit valarray(size_t n)`
  erzeugt ein Valarray mit `n` Elementen, die alle zu 0 initialisiert werden.
- `valarray(const T& val, size_t n)`
  erzeugt ein Valarray mit `n` Elementen, die alle zu `val` initialisiert werden. *Bei diesem Konstruktor ist die Reihenfolge der Argumente im Vergleich zum Vektor vertauscht.* Es kommt erst der Wert und dann die Anzahl.
- `valarray(const T* ptr, size_t n)`
  erzeugt ein Valarray mit `n` Elementen. Die Elemente werden mit den ersten `n` Werten initialisiert, auf die `ptr` zeigt. Dieser Konstruktor eignet sich daher gut zur Umwandlung eines C-Arrays in ein Valarray. Beispiel:

```
double a[] = {2.2, 3.3, 4.4, 9.9};
valarray<double> w(a, 4);
```

- `valarray(const valarray<T>&)` Kopierkonstruktor
- `valarray(const slice_array<T>&)`
  `valarray(const gslice_array<T>&)`
  `valarray(const mask_array<T>&)`
  `valarray(const indirect_array<T>&)`
  Die Argumente dieser Konstruktoren werden in den Abschnitten 34.5 bis 34.8 beschrieben (Seiten 866 – 870).

34.2 Elementfunktionen

- `valarray<T>& operator=(const valarray<T>& rhs)`
  Voraussetzung für die Zuweisung ist, dass die Größe des Arguments `rhs` mit der Größe des aufrufenden Valarrays übereinstimmt, andernfalls ist das Verhalten undefiniert.
- `valarray<T>& operator=(const T& t)`
  Jedem Element des Valarrays wird `t` zugewiesen. Beispiel:

```
valarray<double> vd(4);
vd = 100.123;
```

- `valarray<T>& operator=(const slice_array<T>&)`
  `valarray<T>& operator=(const gslice_array<T>&)`
  `valarray<T>& operator=(const mask_array<T>&)`
  `valarray<T>& operator=(const indirect_array<T>&)`

Die Wirkungsweise dieser Zuweisungsperatoren wird in den Abschnitten 34.5 bis 34.8 beschrieben (Seiten 866 – 870).

- `T operator[](size_t n) const`
  `T& operator[](size_t n)`
  Der Indexoperator verhält sich wie üblich: Eine Überprüfung auf Einhaltung des Bereichs findet nicht statt.
- Mit den folgenden Indexoperatoren können definierte Untermengen aus einem Valarray erzeugt werden:

```
valarray<T> operator[](slice) const
slice_array<T> operator[](slice)
valarray<T> operator[](const gslice&) const
gslice_array<T> operator[](const gslice&)
valarray<T> operator[](const valarray<bool>&) const
mask_array<T> operator[](const valarray<bool>&)
valarray<T> operator[](const valarray<size_t>&) const
indirect_array<T> operator[](const valarray<size_t>&)
```

  Die Wirkungsweise dieser Indexoperatoren wird in den Abschnitten 34.5 bis 34.8 beschrieben (Seiten 864 – 870).
- `valarray<T> operator+() const`
  `valarray<T> operator-() const`
  `valarray<T> operator~() const`
  `valarray<bool> operator!() const`
  Diese unären Operatoren existieren nur, sofern sie für den Typ der Elemente sinnvoll anwendbar sind. Anwendungsbeispiel:

```
v = -u;  // v und u sind Valarrays.
```

- `valarray<T>& operator*=(const valarray<T>&)`
  `valarray<T>& operator/=(const valarray<T>&)`
  `valarray<T>& operator%=(const valarray<T>&)`
  `valarray<T>& operator+=(const valarray<T>&)`
  `valarray<T>& operator-=(const valarray<T>&)`
  `valarray<T>& operator^=(const valarray<T>&)`
  `valarray<T>& operator&=(const valarray<T>&)`
  `valarray<T>& operator|=(const valarray<T>&)`
  `valarray<T>& operator<<=(const valarray<T>&)`
  `valarray<T>& operator>>=(const valarray<T>&)`
  Diese Kurzformoperatoren bewirken, dass die jeweilige Operation mit den entsprechenden Elementen ausgeführt wird, d.h. `u += v;` ist dasselbe wie `u[i] += v[i];` für alle `i`.
- `valarray<T>& operator*=(const T&)`
  `valarray<T>& operator/=(const T&)`
  `valarray<T>& operator%=(const T&)`
  `valarray<T>& operator+=(const T&)`
  `valarray<T>& operator-=(const T&)`
  `valarray<T>& operator^=(const T&)`
  `valarray<T>& operator&=(const T&)`

```
valarray<T>& operator|=(const T&)
valarray<T>& operator<<=(const T&)
valarray<T>& operator>>=(const T&)
```

Diese Kurzformoperatoren bewirken, dass die jeweilige Operation mit allen Elementen ausgeführt wird, d.h. `u += t;` ist dasselbe wie `u[i] += t;` für alle `i`.

- `size_t size() const`
  gibt die Anzahl der Elemente zurück.
- `T sum() const`
  gibt die Summe aller Elemente zurück. Das Valarray muss mindestens ein Element besitzen.
- `T min() const`
  gibt das kleinste Element zurück.
- `T max() const`
  gibt das größte Element zurück.
- `valarray<T> shift(int n) const`
  gibt ein Valarray der Länge `size()` zurück, dessen Elemente um `n` Positionen verschoben sind. Anwendungsbeispiele:

```
u = v.shift(2);
// Ergebnis: u[i] = v[i+2] für 0 ≤ i < size()-2, sowie
// u[size()-2] = u[size()-1] = T(), d.h. 0.
```

```
u = v.shift(-2);
// Ergebnis: u[i] = v[i-2] für 1 < i < size(), sowie
// u[0] = u[1] = T(), d.h. 0.
```

- `valarray<T> cshift(int n) const`
  gibt ein Valarray der Länge `size()` zurück, dessen Elemente um `n` Positionen verschoben sind. Im Unterschied zu `shift()` werden die Elemente rotiert, d.h. durch das Verschieben herausfallende Elemente werden am anderen Ende eingefügt. Beispiel:

```
int n = ...     // irgendeine Zahl
u = v.cshift(n); // Ergebnis: u[i] = v[(i+n)%size()]
```

- `valarray<T> apply(T func(T)) const` und
  `valarray<T> apply(T func(const T&)) const`
  geben ein Valarray der Länge `size()` zurück, wobei die Elemente aus den Ergebnissen der Funktion `func()` angewendet auf die entsprechenden Elemente des aufrufenden Valarrays bestehen:

```
u = v.apply(f); // Ergebnis: u[i] = f(v[i]) für alle i
```

  Leider lassen sich nur (Zeiger auf) Funktionen übergeben, keine Funktionsobjekte!
- `void resize(size_t sz, T c = T())`
  Die Funktion ändert die Größe eines Valarrays und initialisiert alle Elemente. Nach dem Aufruf gilt `size() == sz` und `operator[](i) == c` für alle `i`.

34.3 Binäre Valarray-Operatoren

Zunächst werden alle binären Operatoren und Funktionen beschrieben, ehe auf Elementfunktionen und die Hilfsklassen eingegangen wird.

Binäre arithmetische Valarray-Operatoren

Für `valarray`-Objekte sind die folgenden arithmetischen Operationen definiert, sofern sie mit dem Typ der Elemente verträglich sind (zum Beispiel existiert der Operator '%' nicht für `float`-Zahlen): `*`, `/`, `%`, `+` und `-`. Alle Operatoren treten in drei überladenen Varianten auf, hier gezeigt am Beispiel des Multiplikationsoperators:

```
template<typename T>
valarray<T> operator*(const valarray<T>& v, const valarray<T>& w);
// Anwendung zum Beispiel:
valarray<double> u = v*w; // Ergebnis: u[i] = v[i]*w[i] für alle i

template<typename T>
valarray<T> operator*(const valarray<T>& v, const T& t);
// Anwendung zum Beispiel:
valarray<double> u = v*t; // Ergebnis: u[i] = v[i]*t für alle i

template<typename T>
valarray<T> operator*(const T& t, const valarray<T>& v);
// Anwendung zum Beispiel:
valarray<double> u = t*v; // Ergebnis: u[i] = t*v[i] für alle i
```

Binäre bitweise Valarray-Operatoren

Für `valarray`-Objekte sind die folgenden binären Bit-Operationen definiert, sofern sie mit dem Typ der Elemente verträglich sind: `^`, `&`, `|`, `<<` und `>>`. Auch hier treten die Operatoren in drei überladenen Varianten auf, gezeigt am Beispiel des Und-Operators:

```
template<typename T>
valarray<T> operator&(const valarray<T>& v, const valarray<T>& w);
// Anwendung zum Beispiel:
valarray<int> u = v&w; // Ergebnis: u[i] = v[i] & w[i] für alle i

template<typename T>
valarray<T> operator&(const valarray<T>& v, const T& t);
// Anwendung zum Beispiel:
valarray<int> u = v&t; // Ergebnis: u[i] = v[i] & t für alle i

template<typename T>
valarray<T> operator&(const T& t, const valarray<T>& v);
// Anwendung zum Beispiel:
valarray<int> u = t&v; // Ergebnis: u[i] = t & v[i] für alle i
```

Binäre logische Valarray-Operatoren

Für `valarray`-Objekte sind die folgenden binären logischen Operationen definiert: `&&` und `||`. Auch hier treten die Operatoren in drei überladenen Varianten auf, gezeigt am Beispiel des logischen Und-Operators:

```
template<typename T>
valarray<bool> operator&&(const valarray<T>& v,
                          const valarray<T>& w);

// Anwendung zum Beispiel:
valarray<bool> u = v&&w;  // Ergebnis: u[i] = v[i] && w[i] für alle i

template<typename T>
valarray<bool> operator&&(const valarray<T>& v, const T& t);

// Anwendung zum Beispiel:
valarray<bool> u = v&&t;  // Ergebnis: u[i] = v[i] && t für alle i

template<typename T>
valarray<bool> operator&&(const T& t, const valarray<T>& v);

// Anwendung zum Beispiel:
valarray<bool> u = t&&v;  // Ergebnis: u[i] = t && v[i] für alle i
```

Relationale Valarray-Operatoren

Für `valarray`-Objekte sind die folgenden binären Operationen definiert, sofern sie mit dem Typ der Elemente verträglich sind: `==`, `!=`, `<`, `>`, `<=` und `>=`. Auch hier treten die Operatoren in drei überladenen Varianten auf, gezeigt am Beispiel des Gleichheitsoperators:

```
template<typename T>
valarray<bool> operator==(const valarray<T>& v,
                          const valarray<T>& w);

// Anwendung zum Beispiel:
valarray<bool> u = v==w;  // Ergebnis: u[i] = (v[i] == w[i]) für alle i

template<typename T>
valarray<bool> operator==(const valarray<T>& v, const T& t);

// Anwendung zum Beispiel:
valarray<bool> u = v==t;  // Ergebnis: u[i] = (v[i] == t) für alle i

template<typename T>
valarray<bool> operator==(const T& t, const valarray<T>& v);

// Anwendung zum Beispiel:
valarray<bool> u = t==v;  // Ergebnis: u[i] = (t == v[i]) für alle i
```

34.4 Mathematische Funktionen

Es gibt die folgenden mathematischen Funktionen, die ein Valarray als Argument nehmen und ein Valarray mit dem Ergebnis zurückgeben:

```
template<typename T> valarray<T> abs (const valarray<T>&)
template<typename T> valarray<T> acos (const valarray<T>&)
template<typename T> valarray<T> asin (const valarray<T>&)
template<typename T> valarray<T> atan (const valarray<T>&)
template<typename T> valarray<T> cos (const valarray<T>&)
template<typename T> valarray<T> cosh (const valarray<T>&)
template<typename T> valarray<T> exp (const valarray<T>&)
template<typename T> valarray<T> log (const valarray<T>&)
template<typename T> valarray<T> log10(const valarray<T>&)
template<typename T> valarray<T> sin (const valarray<T>&)
template<typename T> valarray<T> sinh (const valarray<T>&)
template<typename T> valarray<T> sqrt (const valarray<T>&)
template<typename T> valarray<T> tan (const valarray<T>&)
template<typename T> valarray<T> tanh (const valarray<T>&)
```

Als Anwendung sei hier die Sinusfunktion gezeigt:

```
valarray<double> dieWinkel(100); // 100 Werte
// ... Berechnung der Winkel fehlt hier
// Berechnung der Sinuswerte:
valarray<double> dieSinusWerte = sin(dieWinkel);
// Ergebnis: dieSinusWerte[i] = sin(dieWinkel[i]) für alle i
```

Die Funktionen `atan2()` und `pow()` treten in überladenen Variationen auf:

```
template<typename T>
valarray<T> atan2(const valarray<T>& v, const valarray<T>& w);
// Beispielanwendung mit dem Ergebnis u[i] = atan(v[i]/w[i]) für alle i:
valarray<double> u = atan2(v, w);
```

```
template<typename T>
valarray<T> atan2(const valarray<T>& v, const T& t);
// Beispielanwendung mit dem Ergebnis u[i] = atan(v[i]/t) für alle i:
valarray<double> u = atan2(v, t);
```

```
template<typename T>
valarray<T> atan2(const T& t, const valarray<T>& v);
// Beispielanwendung mit dem Ergebnis u[i] = atan(t/v[i]) für alle i:
valarray<double> u = atan2(t, v);
```

```
template<typename T>
valarray<T> pow(const valarray<T>& v, const valarray<T>& w);
// Beispielanwendung mit dem Ergebnis u[i] = v[i]^w[i] für alle i:
valarray<double> u = pow(v, w);
```

```
template<typename T>
valarray<T> pow(const valarray<T>& v, const T& t);
// Beispielanwendung mit dem Ergebnis u[i] = v[i]^t für alle i:
valarray<double> u = pow(v, t);
```

```
template<typename T>
valarray<T> pow(const T& t, const valarray<T>& v);
// Beispielanwendung mit dem Ergebnis u[i] = t^v[i] für alle i:
valarray<double> u = pow(t, v);
```

34.5 slice und slice_array

Die Klasse `slice` (dt. Scheibe, Querschnitt, Teil) ist eine Abstraktion, die es erlaubt, jede Zeile oder Spalte eines Valarrays, das als Matrix interpretiert wird, zu beschreiben oder andere Teilmengen eines Valarrays anzusprechen. Die Klasse ist wie folgt definiert:

```
class slice {
  public:
    slice();
    slice(size_t start, size_t size, size_t stride);
    size_t start() const; // Index des ersten Elements
    size_t size() const; // Anzahl der Elemente
    size_t stride() const; // Schrittweite N
  private:
    size_t start_, size_, stride_;
};
```

Ein `slice`-Objekt definiert eine Folge von Zahlen, die die Position eines jeden N-ten Elements eines Valarrays angeben, indem die Zahlen *Index des ersten Elements* und *Schrittweite* auf die Position im Valarray abgebildet werden. Zum Beispiel definiert `slice(3, 8, 2)` die Folge 3, 5, 7, 9, 11, 13, 15, 17. Damit lassen sich Untermengen einer Matrix definieren, zum Beispiel Zeilen, Spalten oder Untermatrizen. Im folgenden Beispiel werden die Indizes aller Zeilen und aller Spalten einer Matrix mit 3 Zeilen und 4 Spalten mithilfe von `slice`-Objekten ausgegeben.

Listing 34.1: slice

```
// Auszug aus cppbuch/k34/slices.cpp
#include <iostream>
#include "valarray"
using namespace std;

void printIndizes(slice s) {
   for(size_t i = 0; i < s.size(); ++i)
     cout << s.start()+i*s.stride() << ' ';
   cout << endl;
}
```

```
int main() {
  const size_t ZEILEN = 3;
  const size_t SPALTEN = 4;
  cout << "Zeige Indizes von Zeilen und Spalten:\n";
  for(size_t zeile = 0; zeile < ZEILEN; ++zeile) {
    cout << "Zeile " << zeile << ": ";
    size_t start_index = zeile * SPALTEN;
    printIndizes(slice(start_index, SPALTEN, 1));
  }
  cout << endl;
  for(size_t spalte = 0; spalte < SPALTEN; ++spalte) {
    cout << "Spalte " << spalte << ": ";
    printIndizes(slice(spalte, ZEILEN, SPALTEN));
  }
}
```

Das Ergebnis dieses Programms ist:

```
Zeile 0:  0  1  2  3
Zeile 1:  4  5  6  7
Zeile 2:  8  9  10  11

Spalte 0: 0  4  8
Spalte 1: 1  5  9
Spalte 2: 2  6  10
Spalte 3: 3  7  11
```

Der Indexoperator `valarray<T>::operator[](slice)` tritt in zwei Varianten auf:

```
valarray<T> valarray<T>::operator[](slice) const
slice_array<T> valarray<T>::operator[](slice)
```

Er erlaubt es, eine mit einem `slice`-Objekt definierte Untermenge zu extrahieren und zu verändern:

```
valarray<double> v(ZEILEN * SPALTEN); // Platz für 3x4-Matrix
                                      // Konstanten wie oben
// ... (hier Werte zuweisen)
// Zeile 1 (Zählung ab 0) als Valarray extrahieren:
slice z1(4, 4, 1); // Zeile 1 einer 3x4 Matrix definieren
valarray<double> zeile1 = v[z1];
```

Das Valarray `zeile1` besteht aus 4 Elementen, die Kopien der Elemente `v[4]` bis `v[7]` sind. Veränderungen einer Valarray-Untermenge sind auf folgende Art möglich:

```
v[z1] = 0.0;   // Wirkung: v[4] = v[5] = v[6] = v[7] = 0.0
// auf Zeile 2 die oben erzeugte zeile1 addieren, d.h
// v[8] += zeile1[0] usw. bis v[11] += zeile1[3]:
v[slice(2*4, 4, 1)] += zeile1;
```

Der auf der linken Seite eines binären Operators stehende Indexoperator liefert ein `slice_array` zurück, für das der binäre Operator aufgerufen wird. Die möglichen Operatoren sind im folgenden Abschnitt 34.5 aufgelistet. Die Klasse `slice_array` tritt im obigen Beispiel nicht explizit auf. Man muss sie nicht selbst kennen, sondern nur ihre Operatoren, die verändernd auf Untermengen von Valarrays wirken.

slice_array

Ein `slice_array` ist eine Hilfsklasse, die eine Referenz auf ein Valarray und ein `slice`-Objekt zur Definition einer Untermenge für dieses Valarray enthält. Die Klasse beschreibt also nur eine Untermenge, und sie ermöglicht für diese Untermenge die in der Klassendefinition beschriebenen Operationen:

```
template<typename T>
class slice_array {
  public:
   typedef T value_type;
   void operator= (const valarray<T>&) const;
   void operator*= (const valarray<T>&) const;
   void operator/= (const valarray<T>&) const;
   void operator%= (const valarray<T>&) const;
   void operator+= (const valarray<T>&) const;
   void operator-= (const valarray<T>&) const;
   void operator^= (const valarray<T>&) const;
   void operator&= (const valarray<T>&) const;
   void operator|= (const valarray<T>&) const;
   void operator<<=(const valarray<T>&) const;
   void operator>>=(const valarray<T>&) const;
   void operator=(const T&);
  private:
   // ...
};
```


34.6 gslice und gslice_array

Ein `slice`-Objekt kann eine Zeile oder Spalte einer Matrix beschreiben. Manchmal wird aber eine Untermatrix benötigt, im allgemeinsten Fall eine m-dimensionale Untermatrix einer n-dimensionalen Matrix ($m \leq n$). Dazu dienen `gslice`-Objekte. Die Klasse `gslice` (Abk. für *generalized slice*) enthält alle dafür notwendigen Informationen, nämlich die `slice`-Werte für Anzahl und Schrittweite, gespeichert in Valarrays, und den gemeinsamen Startpunkt:

```
class gslice {
  public:
    gslice();
    gslice(size_t start,
           valarray<size_t>& lengths, valarray<size_t>& strides);
    size_t start() const;
    valarray<size_t> size() const;
    valarray<size_t> stride() const;
  private:
    size_t start_;
    valarray<size_t> sizes_, strides_;
};
```

Die Wirkung soll an einer 4x5-Matrix gezeigt werden, zu der die Indizes von Submatrizen ausgegeben werden. Dazu werden zunächst zwei Hilfsfunktionen definiert, die in `main()` benutzt werden:

```
// mit einem gslice definierte Indizes ausgeben
void printIndizes(const gslice& gs) {
   for(size_t r = 0; r < gs.size()[0]; ++r) { // Zeilen
     for(size_t c = 0; c < gs.size()[1]; ++c) // Spalten
        cout << '\t'
             << gs.start()
                + r * gs.stride()[0]
                + c * gs.stride()[1];
     cout << endl;
   }
}
```

```
// gslice einer zweidimensionalen Untermatrix aus einer durch gs
// definierten zweidimensionalen Matrix zurückgeben
gslice submatrix_gslice(const gslice& gs, // 2-D Matrix
                        size_t position, // Startposition
                        size_t rows,     // Zeilen
                        size_t cols) { // Spalten
  size_t sz[] = {rows, cols};
  size_t str[] = {gs.size()[1], 1};
  valarray<size_t> sizes(sz, 2);
  valarray<size_t> strides(str, 2);
  return gslice(position, sizes, strides);
}
```

```
// Auszug aus cppbuch/k34/gslices.cpp
int main() {
   const size_t ZEILEN = 4, SPALTEN = 5; // 4x5 Matrix
   valarray<size_t> sizes(2);          // 2 Dimensionen
   valarray<size_t> strides(2);
   sizes [0] = ZEILEN;
   sizes [1] = SPALTEN;
   strides[0] = SPALTEN;
   strides[1] = 1;
   // gslice in der Ecke oben links konstruieren
   gslice g(0, sizes, strides);
   cout << endl << "Indizes der 4x5 Matrix:\n";
   printIndizes(g);
```

Ergebnis:

```
      0      1      2      3      4
      5      6      7      8      9
      10     11     12     13     14
      15     16     17     18     19
```

Die Indizes einer 3x4 Untermatrix, die an Position 1 beginnt, werden mit

```
   printIndizes(submatrix_gslice(g, 1, 3, 4));
```

ausgegeben und führen zu dem Ergebnis

```
1       2       3       4
6       7       8       9
11      12      13      14
```

Ähnlich wie bei der `slice`-Klasse lassen sich Untermengen eines als Matrix interpretierten Valarrays verändern und extrahieren. Der Indexoperator `valarray<T>:: operator[](const gslice&)` dient zum Ermitteln der Untermenge und tritt in zwei Varianten auf:

```
valarray<T> valarray<T>::operator[](const gslice&) const
gslice_array<T> valarray<T>::operator[](const gslice&)
```

Die folgende Fortsetzung des `main()`-Programms zeigt die mögliche Anwendung:

```
// 4x5 Matrix erzeugen und initialisieren
valarray<double> v(ZEILEN*SPALTEN);
for(size_t i = 0; i < v.size(); ++i)
    v[i] = i;                   // Wert = Index

// gslice für 2x3 Untermatrix an Position 6 erzeugen
gslice gs(submatrix_gslice(g, 6, 2, 3));

// Kopie der Untermatrix erzeugen
valarray<double> sub = v[gs];
```

Das Valarray `sub` enthält die folgenden Elemente:

```
 6       7       8
11      12      13
```

Mit verschiedenen Operatoren kann eine Untermatrix innerhalb der Matrix verändert werden. Die möglichen Operatoren sind im folgenden Abschnitt 34.6 aufgelistet. Hier wird das Nullsetzen der durch `gs` definierten Untermatrix gezeigt:

```
// Untermatrix = 0 setzen
v[gs] = 0.0;
```

Nach dieser Anweisung enthält Matrix `v` die folgenden Werte:

```
0       1       2       3       4
5       0       0       0       9
10      0       0       0       14
15      16      17      18      19
```

Die mathematischen Operationen setzen ein Valarray als Argument voraus, zum Beispiel

```
v1[gs] *= v2;  // v1 und v2 sind Valarrays
```

Die Wirkung dieser Anweisung besteht darin, dass diejenigen Elemente von `v1`, die durch das `gslice`-Objekt `gs` indiziert werden, mit den entsprechenden Elementen von `v2` multipliziert werden. Bei der oben gegebenen Definition von `gs` ist die Wirkung

```
// Wirkung der Anweisung v1[gs] *= v2;
v1[6] *= v2[0];
v1[7] *= v2[1];
v1[8] *= v2[2];
v1[11] *= v2[3];
```

```
v1[12] *= v2[4];
v1[13] *= v2[5];
```

gslice_array

Ein `gslice_array` ist eine Hilfsklasse, die eine Referenz auf ein Valarray und ein `gslice`-Objekt zur Definition einer Untermenge für dieses Valarray enthält. Die Klasse `gslice_array` ermöglicht für diese Untermenge dieselben Operationen wie die Klasse `slice_array`, also `=` für ein `const T&`-Argument und `*=`, `/=`, `%=`, `+=`, `-=`, `^`, `&=`, `|=`, `<<=`, `>>=`, `=` für Valarray-Argumente.

34.7 mask_array

Ein `mask_array` spielt eine ähnliche Rolle für Operationen auf Untermengen von Valarrays wie ein `slice_array`. Der Unterschied besteht in der Auswahl der Untermenge, die hier durch ein Valarray mit `bool`-Elementen geschieht. Die Untermenge besteht aus denjenigen Elementen eines Valarrays, für die das entsprechende Element des `bool`-Arrays den Wert `true` hat. Der Indexoperator `valarray<T>::operator[](const valarray<bool>&)` dient zum Ermitteln der Untermenge und tritt in zwei Varianten auf:

```
valarray<T> valarray<T>::operator[](const valarray<bool>&) const
mask_array<T> valarray<T>::operator[](const valarray<bool>&)
```

Die zweite Variante wird implizit benutzt, wenn die durch das `bool`-Array ausgewählte Untermenge geändert werden soll. Wie die Klassen `slice_array` und `gslice_array` tritt die Klasse `mask_array` nicht direkt in Erscheinung, man muss nur die möglichen Operationen kennen. Beispiele:

```
// Valarray anlegen und initialisieren
valarray<double> v(12);
for(size_t i = 0; i < v.size(); ++i)
   v[i] = i;

// bool-Array mit einer Größe ≤ v.size() anlegen.
valarray<bool> maske(10);

// Jedes 3. Element = true setzen
for(size_t i = 0; i < maske.size(); i+= 3)
  maske[i] = true;

// Untermenge auswählen
const valarray<double> u(v[maske]);
// Ergebnis: u =  0 3 6 9

// Elementen der Untermenge einen Wert zuweisen
v[maske] = 20;
// Ergebnis: v =  20 1 2 20 4 5 20 7 8 20 10 11
```

```
// Zweites Valarray vmult erzeugen
valarray<double> vmult(8); //vmult.size() ≥ Anzahl der trues in maske
for(size_t i = 0; i < vmult.size(); ++i)
  vmult[i] = 0.1*i; // 0.0 0.1 0.2 ... 0.7

// Untermenge mit allen Elementen von vmult multiplizieren
v[maske] *= vmult;
// Ergebnis: v =  0 1 2 2 4 5 4 7 8 6 10 11,
// d.h. 20*0.0, 1, 2, 20*0.1, 4, 5, 20*0.2, 7, 8, 20*0.3, 10, 11
```

Die Klasse `mask_array` ermöglicht für die durch das boolesche Array ausgewählte Untermenge dieselben Operationen wie die Klasse `slice_array`, also `=` für ein `const T&`-Argument und `*=`, `/=`, `%=`, `+=`, `-=`, `^=`, `&=`, `|=`, `<<=`, `>>=`, `=` für Valarray-Argumente.

34.8 indirect_array

Ein `indirect_array` spielt eine ähnliche Rolle für Operationen auf Untermengen von Valarrays wie ein `slice_array`. Der Unterschied besteht in der Auswahl der Untermenge, die hier durch ein Index-Array geschieht (indirekte Adressierung). Ein Index-Array ist ein Valarray mit Elementen vom Typ `size_t`, die Indizes, also Adressen von Array-Elementen, darstellen. Die Untermenge besteht aus denjenigen Elementen eines Valarrays, auf die das entsprechende Element des Index-Arrays verweist. Der Indexoperator `valarray<T>::operator[](const valarray<size_t>&)` dient zum Ermitteln der Untermenge und tritt in zwei Varianten auf:

```
valarray<T> valarray<T>::operator[](const valarray<size\_t>&) const
indirect_array<T> valarray<T>::operator[](const valarray<size\_t>&)
```

Die zweite Variante wird implizit benutzt, wenn die durch das Index-Array ausgewählte Untermenge geändert werden soll. Wie die Klassen `slice_array` und `gslice_array` tritt die Klasse `indirect_array` nicht direkt in Erscheinung, man muss nur die möglichen Operationen kennen. Beispiele:

```
// Valarray anlegen und initialisieren
valarray<double> v(10);
for(size_t i = 0; i < v.size(); ++i) {
  v[i] = 0.1 * i; // 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
}

// indirektes Valarray (Index-Array) anlegen und initialisieren
valarray<size_t> i_arr(v.size()-4);
for(size_t i = 0; i < i_arr.size(); ++i) {
  i_arr[i]= i_arr.size() - i + 2; // 8 7 6 5 4 3
}
```

Für ein Index-Array müssen zwei Bedingungen gelten, hier mit den beiden Valarrays `v` und `i_arr` formuliert:

`i_arr.size()` $\leq$ `v.size()`

und

`i_arr[i]` < `v.size()`, 0 $\leq$ `i` < `i_arr.size()`

Andernfalls ist das Verhalten des Programms undefiniert. Eine Untermenge kann wie folgt erzeugt werden:

```
const valarray<double> u(v[i_arr]); // 0.8 0.7 0.6 0.5 0.4 0.3
```

Es gilt `v[i_arr[i]] == u[i]` für alle `i` im Bereich 0 $\leq$ `i` < `i_arr.size()`. Die Klasse `indirect_array` ermöglicht für die durch das Index-Array ausgewählte Untermenge dieselben Operationen wie die Klasse `slice_array`, also `=` für ein `const T&`-Argument und `*=`, `/=`, `%=`, `+=`, `-=`, `^`, `&=`, `|=`, `<<=`, `>>=`, `=` für Valarray-Argumente. Beispiele:

```
v[i1_arr] = 20;
// Ergebnis: v =  0.0 0.1 0.2 20 20 20 20 20 20 0.9

valarray<double> vb(10);
for(size_t i = 0; i < vb.size(); ++i)
  vb[i] = 0.01*i; // .0 .01 .02 .03 .04 .05 .06 .07 .08 .09

v[i1_arr] = vb;
// Ergebnis: v =  0.0 0.1 0.2 0.05 0.04 0.03 0.02 0.01 0.0 0.9

v[i1_arr] *= vb;
// Ergebnis: v =  0 0.1 0.2 0.0025 0.0016 0.0009 0.0004 0.0001 0 0.9
```

Für solche Operationen gilt `v[i_arr[i]] *= vb[i]`, hier also

```
v[8] *= vb[0]; v[7] *= vb[1]; v[6] *= vb[2];
v[5] *= vb[3]; v[4] *= vb[4]; v[3] *= vb[5];
```

Außer den gezeigten Beispielen zur Klasse `valarray` finden sich weitere in den Beispielen, siehe Verzeichnis *cppbuch/k34*. Diese Beispiele demonstrieren auf einfache Art die beschriebenen Konzepte. Als Ergänzung gibt es ein Beispiel, das nicht so einfach und daher eher für Mathematiker interessant ist: Ein Programm zur rekursiven Matrixmultiplikation (Unterverzeichnis *cppbuch/k34/rekursiveMatrixMult*).