# Vektoren

Vektoren bestehen aus einer festen Zahl von Komponenten desselben Datentyps, die über einen (ab 0 gezählten) Index angesprochen werden. Der Datentyp einer Vektorkomponente kann einfach oder zusammengesetzt sein.

In C gibt es unterschiedliche Vektordatentypen. Die direkt in die Sprache eingebauten Vektoren, die aus C übernommen wurden, haben eine konstante Länge, die bei ihrer Definition angegeben wird und nur eingeschränkt abfragbar ist. (Insbesondere bei der Übergabe an Funktionen geht die Längeninformation verloren.) Mit Hilfe von Zeigern lassen sich auch C-Vektoren variabler Länge realisieren.

Die in C++ üblicherweise verwendeten Vektoren sind in der Standard Template Library definiert, ihre Länge (Komponentenanzahl) ist veränderbar und kann abgefragt werden.

Bei den genannten Vektoren kann mit $a[i]$ auf die Komponente zum Index $i$ des Vektors $a$ der Länge $n$ zugegriffen werden. Darin muß $i$ ein ganzzahliger Ausdruck sein und $0 \leq i < n$ gelten.

Mit der Vereinbarung eines Vektors $a$ der Länge $n$ wird auch der Speicherplatz für die Komponenten $a[0]$, $a[1]$,...,$a[n-1]$ vereinbart.

## C-Vektoren

### Eindimensionale Felder

Eindimensionale Felder sind Vektoren, deren Komponenten aus einfachen Datentypen gebildet werden. Die Feldlänge muss mindestens 1 sein.

*Beispiel:*

```
          :
const int M=50;
double a[M],b[2*M];  // Vektorlaengen werden waehrend des Uebersetzens berechnet
for (int i=0; i<M; ++i) a[i]=b[2*i+1]=b[2*i]=i*i;
          :
```

### Mehrdimensionale Felder

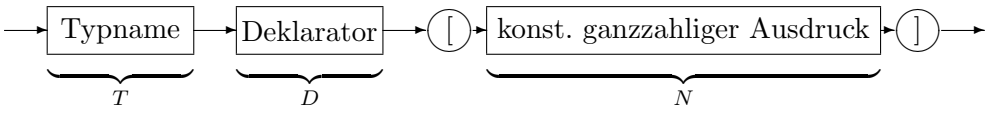
Zweidimensionale Felder werden als Vektoren von Vektoren aus einfachen Datentypen realisiert usw.

*Beispiel:* $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, $b = Ax \in \mathbb{R}^m$, *d.h.* $b_i = \sum_{j=0}^{n-1} a_{ij} x_j \;\; (i = 0, \ldots, m-1)$

```
#include <iostream>
#include <iomanip>
#include <cstdlib>

using namespace std;

int main()
{
  const int M=20,N=30;
  int m,n;
  double a[M][N],x[N],b[M];

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

```
  if (m>M || n>N)  { exit(1); /* Abbruch */ }

  /* Matrixeingabe (a) */
  cout << endl << "a:" << endl;
  for (int i=0; i<m; ++i)
    for (int j=0; j<n; ++j) cin >> a[i][j];

  // Vektoreingabe (x)
  cout << endl << "x:" << endl;
  for (int j=0; j<n; ++j) cin >> x[j];

  // Matrixvektormultiplikation (a*x)
  for (int i=0; i<m; ++i) {
    b[i] = 0;
    for (int j=0; j<n; ++j) b[i] += a[i][j]*x[j];
  }

  // Vektorausgabe (b)
  cout << endl << "b:" << endl;
  for (int i=0; i<m; ++i)
    cout << fixed << setprecision(2) << setw(6) << b[i] << endl;
  return 0;
}
```

## Vereinbarungssyntax




Falls $T\ D$ einen Namen als "Datenstruktur aus $T$" vereinbart, dann vereinbart $T\ D[N]$ denselben Namen als "Datenstruktur aus $N$-Vektor von $T$".

Diese Definition bewirkt, dass der Datentyp aus dem Deklarator durch Lesen von innen nach außen bestimmt werden kann.

*Beispiele:*

| | |
|---|---|
| `double a[M]` | M-Vektor von double |
| `double a[M][N]` | M-Vektor von N-Vektor von double |
| `int a[N1][N2]...[Nk]` | $N_1$-Vektor von $N_2$-Vektor ... von $N_k$-Vektor von int |

## Initialisierung

Falls $I_0$,...,$I_{N-1}$ zulässige Initialisierungen für Variablen eines Datentyps sind, dann ist $\{I_0, \ldots, I_{N-1}\}$ oder $\{I_0, \ldots, I_{N-1},\}$ eine Initialisierung für einen $N$-Vektor von diesem Datentyp.

Sind weniger Initialisierungen vorhanden als benötigt, so werden die restlichen Komponenten nach Voreinstellung initialisiert. Die Voreinstellung ist für arithmetische Datentypen und für Zeigertypen 0.

Die Initialisierungsregeln werden rekursiv angewendet.

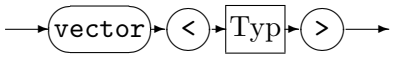
Falls die Dimension des Vektors unbekannt ist, wird sie durch die Zahl der Elemente der Initialisierungsliste bestimmt. (Nur die erste Felddimension darf fehlen.)

Mehrfache geschweifte Klammern können weggelassen werden, die einzelnen Elemente werden ggf. nach Reihenfolge initialisiert.

*Beispiele:*

| | |
|---|---|
| `double a[3] = {1.0,4.0,}` | $a_0 = 1$, $a_1 = 4$, $a_2 = 0$ |
| `double b[] = {1,4}` | vereinbart 2-Vektor mit $b_0 = 1$, $b_1 = 4$ |
| `int c[3][3] = {{1,2},{3}}` | $c_{00} = 1$, $c_{01} = 2$, $c_{10} = 3$, sonst 0 |
| `int d[][3] = {{4,2,1},{5,2,7}}` | vereinbart implizit `int d[2][3]` |
| `double e[10][10] = {0}` | $e_{ij} = 0$ |

## STL-Behälterklasse für Vektoren (`<vector>`)




*Beispiel:*

```
#include <vector>
using namespace std;
      :
vector<double> a(n);    // nicht a[n]!
```

Im folgenden steht $T$ für den Vektorkomponententyp, $t$ für ein Objekt vom Datentyp $T$, $i$ und $n$ sind *vorzeichenlose* ganze Zahlen (vom Typ `vector<T>::size_type`).

| *Operation* | *Wirkung* |
|---|---|
| `vector<T> a` | vereinbart leeren Vektor |
| `vector<T> a(n)` | vereinbart Vektor aus $n$ Komponenten, jeweils nach Voreinstellung für $T$ initialisiert |
| `vector<T> a(n,t)` | vereinbart Vektor aus $n$ Komponenten, jeweils mit $t$ (vom Datentyp $T$) initialisiert |
| `vector<T> a(b)` | vereinbart Vektor $a$ mit Dimension und Komponentenwerten von $b$ |
| `vector<T> a={`$t_0 \ldots, t_{n-1}$`}`* | vereinbart Vektor $a$ mit Dimension $n$ und Komp.werten $t_0, \ldots, t_{n-1}$ |
| `vector<T> a{`$t_0 \ldots, t_{n-1}$`}`* | " |
| `a[i]` | Komponentenwert zum Index $i$ (ganzzahlig, vorzeichenlos) |
| `a.at(i)` | Komponentenwert (bei Bereichsverletzung wird die Ausnahme `out_of_range` erzeugt) |
| `a.front()` `a.back()` | erste (Index 0) bzw. letzte Komponente |
| `a=b` | Zuweisung: $a$ wird zu einer Kopie von $b$ |
| `a={`$t_0 \ldots, t_{n-1}$`}`* | Zuweisung: $a_i = t_i$ $(i = 0, \ldots, n-1)$, $n$ Dimension von $a$ |
| `a==b a!=b a>b a<b a<=b a>=b` | Vergleiche |
| `a.size()` | Zahl der Komponenten |
| `a.resize(n)` | ändert Komponentenzahl, ggf. Erzeugung und Initialisierung neuer Komponenten bzw. Löschen vorhandener Komponenten |
| `a.resize(n,t)` | wie `a.resize(n)`, jedoch ggf. Initialisierung mit $t$ |
| `a.push_back(t)` | Komponente mit Wert $t$ anhängen |
| `a.pop_back()` | letzte Komponente löschen |
| `a.clear()` | alle Komponenten löschen |

*Beispiel.: Initialisierung ab C++11*

```
vector<double> a = {1.0,4.0,0.0};        // vector<double> a(3) = {...} unzulaessig
vector<double> b = {1,4}
vector<vector<int>> c = {{1,2},{3}};     // Keine Rechtecksmatrix
vector<vector<int>> d = {{4,2,1},{5,2,7}};
vector<vector<double>> e(10);            // Keine Matrix (Zeilen leer)
                                         // Voellig falsch mit e(10,10) statt e(10)
```

*C++11

*Beispiel: Euklidische Norm eines double-Vektors*

```
#include <iostream>
#include <cmath>
#include <vector>

using namespace std;

double euklidische_norm(vector<double> a)
{
  double s=0;
  // Vermeidet Warnungen wegen des Vergleichs von signed und unsigned
  for (vector<double>::size_type i=0; i<a.size(); ++i)
    s += a[i]*a[i];
  return sqrt(s);
}

int main()
{
  int n;
  vector<double> a;

  cout << "n: "; cin  >> n;
  a.resize(n); // Alternativ a erst hier definieren:  vector<double> a(n);

  cout << "a[0] a[1] .. a[" << n-1 << "]: ";
  for (int i=0; i<n; ++i) cin >> a[i];

  cout << "euklidische_norm(a) = " << euklidische_norm(a) << endl;
  return 0;
}
```

Es gibt keinen vordefinierten Matrixtyp bzw. Datentyp für mehrdimensionale Felder in der STL. Matrizen können als Vektoren aus Vektoren gebildet werden. Bei Verwendung des STL-Datentyps `vector` ist allerdings ein erhöhter Speicherbedarf einzukalkulieren, weil in der Regel mehr Speicherplatz als notwendig alloziert wird, um Vergrößerungen ohne interne Umspeicherung durchführen zu können. Außerdem ist der Zugriff auf die Matrixelemente weniger effizient.

*Beispiel: Vereinbarung einer Matrix* $A \in \mathbb{R}^{m \times n}$

```
#include <vector>
      :
vector<vector<double>> a(m);    // m-Vektor aus leeren Vektoren aus double
for (int i=0; i<m; ++i)         // m-Vektor aus n-Vektoren aus double
  a[i].resize(n);               // durch Vergroessern der "Zeilenvektoren"
```

*Alternative:*

```
#include <vector>
      :
vector<vector<double>> a(m,vector<double>(n));
```

*Anmerkung:* In C++98 ist zur Vermeidung der Fehlinterpretation von `>>` als Shiftoperator Zwischenraum erforderlich: `vector<vector<double> >`.