§4 Vektoren - Allgemeines

Leitideen: *Vektoren bestehen aus einer festen Zahl von Komponenten desselben Datentyps, die über einen Index ansprechbar sind. Diese Komponenten werden direkt aufeinanderfolgend in einem zusammenhängenden Speicherbereich gespeichert.*

„C-Vektoren“ bestehen nur aus Vektorkomponenten, ihre Länge gehört zwar zum Datentyp, ist aber nur bedingt abfragbar. Sie wird zur Übersetzungszeit festgelegt und in Funktionsaufrufen nicht übergeben.

Dynamisch allozierte Speicherbereiche können als Vektoren variabler Länge mit der C-Vektorsyntax genutzt werden.

Die leichter benutzbaren STL-Vektoren speichern ihre Länge, die veränderbar ist. Sprachmechanismen sorgen dafür, dass die dynamische Allokation und Deallokation des von den Komponenten genutzten Speicherbereichs automatisch erfolgt.

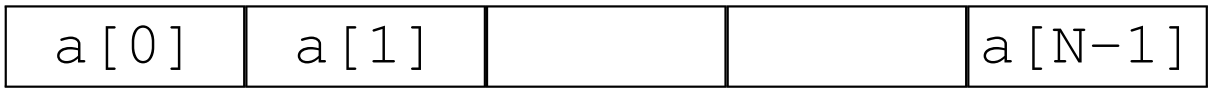
§4 Vektoren - Themenübersicht

- C-Vektoren - Überblick I,II
- C-Vektoren - Vereinbarungssyntax I,II
- C-Vektoren - Initialisierung I,II
- Vektoren in der STL I,II,III,IV

C-Vektoren - Überblick

- „C-Vektoren“: Bezeichnung für die aus C übernommenen Vektoren. Werden auch in C++ verwendet, allerdings vor allem bei der Implementierung von Klassen und der Benutzung von C-Funktionen.
- Vektorindizes beginnen immer mit 0.
  Ein Vektor a aus N Komponenten („N-Vektor“) hat daher folgende Komponenten: a[0], a[1],. . . ,a[N-1]
- Es gibt C-Vektoren konstanter Länge und dynamisch allozierte Speicherbereiche (Zugriff über Vektorsyntax)
  Bsp.:
  ```
  const int N=10
  double a[N];     // a Vektor mit double-Komp.
                   // a[0],. . . ,a[9]
  double b[2*N]; // auch erlaubt
  ```
  Festlegung der Vektorlänge zur Übersetzungszeit

Skizze:



C-Vektor a aus N Komponenten

C-Vektoren - Überblick II

- Für den Zugriff auf eine Komponente genügt prinzipiell die Anfangsadresse des Speicherbereichs und der über den Index bestimmbare Offset.
- Das Kopieren von C-Vektoren wird in C++ nicht direkt unterstützt.
- Insbesondere wird bei der Parameterübergabe in Funktionen der Vektor *nicht* kopiert, sondern seine Anfangsadresse übergeben *(später!)*
- Matrizen werden als C-Vektoren von C-Vektoren gebildet:
  Bsp.:
  ```
  const int M=20,N=30
  double a[M][N];
  // a M-Vektor von N-Vektor von double
  // ≙ M×N-Matrix von double
  ```

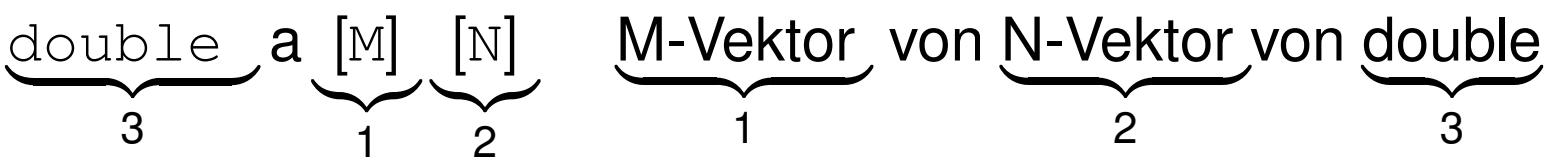
C-Vektoren - Vereinbarungssyntax

`a[i]`	Komponente i (ab 0 gezählt) des Vektors a
`double a[N]`	N-Vektor aus double-Zahlen
	N Komponenten: a[0], a[1], . . . , a[N-1]

Ziel der Vereinbarungssyntax: Ähnlichkeit zu Ausdrücken

`a[i]`	double-Zahl
`double a[N]`	N-Vektor von double
`a[i][j]`	double-Zahl
`double a[M][N]`	M-Vektor von N-Vektor von double

Mechanismus: Lesen von innen nach außen



Dieser Mechanismus ist grundlegend für die Vereinbarungssyntax von C++ !

[∗] C-Vektoren - Vereinbarungssyntax II

- Vereinbarung einer Variablen (vereinfacht):

  $\underbrace{\textit{Typname}}_{T} \quad \underbrace{\textit{Deklarator}}_{D}$

  z.B.: $\underbrace{\texttt{double}}_{T} \ \underbrace{\texttt{a}}_{D}$

- Im allg.: *Deklarator* $\neq$ *Variablenname*

  z.B.: $\underbrace{\texttt{double}}_{T} \ \underbrace{\texttt{a[M][N]}}_{D}$

- *Regel:* $T \ D$ Datenstruktur von T

  $\Rightarrow$ $T \ D[N]$ Datenstruktur von N-Vektor von T

  z.B.:

  `double a[M]` M-Vektor von double

  $\Rightarrow$ `double a[M][N]` M-Vektor von N-Vektor von double

  Mit dem Anhängen von `[N]` an `a[M]` verlängert sich “M-Vektor“ zu “M-Vektor von N-Vektor“. Usw.

C-Vektoren - Initialisierung

- Initialisierungliste in geschweiften Klammern, Komma als Trennzeichen, Komma nach letztem Element optional

  Bsp.: `double a[3]={1.0,4.0,0.0}`

  $a_0 = 1, a_1 = 4, a_2 = 0$

- Falls zu wenig Elemente angegeben, Auffüllen mit 0

  Bsp.: `double a[3]={1.0,4.0}`

  $a_0 = 1, a_1 = 4, a_2 = 0$

- Fehlende erste Vektordimension wird aus Initialisierungsliste bestimmt. (Nur die erste Vektordimension darf fehlen!)

  Bsp.: `double b[]={1.0,4.0}`

  b 2-Vektor mit $b_0 = 1, b_1 = 4$

- Rekursive Anwendung der Initialisierungsregeln

  Bsp.: `int c[3][3]={{1,2},{3}}`

  $c[0] \leftarrow \{1, 2\}$, d.h. $c_{00} = 1, c_{01} = 2, c_{02} = 0$

  $c[1] \leftarrow \{3\}$, d.h. $c_{10} = 3, c_{11} = 0, c_{12} = 0$

  $c[2] \leftarrow \{0\}$, d.h. $c_{20} = 0, c_{21} = 0, c_{22} = 0$

C-Vektoren - Initialisierung II

Bsp.: `int d[][3]={{4,2,1},{5,2,7}}`
$d[0] \leftarrow \{4,2,1\}$, d.h. $d_{00} = 4, d_{01} = 2, d_{02} = 1$
$d[1] \leftarrow \{5,2,7\}$, d.h. $d_{10} = 5, d_{11} = 2, d_{12} = 7$
d 2-Vektor von 3-Vektor von int

- Mehrfache geschweifte Klammern können weggelassen werden, Initialisierung dann nach Reihenfolge

  Bsp.: `int d[][3]={4,2,1,5,2,7}`
  d wie zuvor

- Initialisierung mit 0 als Spezialfall

  Bsp.: `double e[10][10]={0}`
  $e_{ij} = 0$ für $i, j = 0, \ldots, 9$

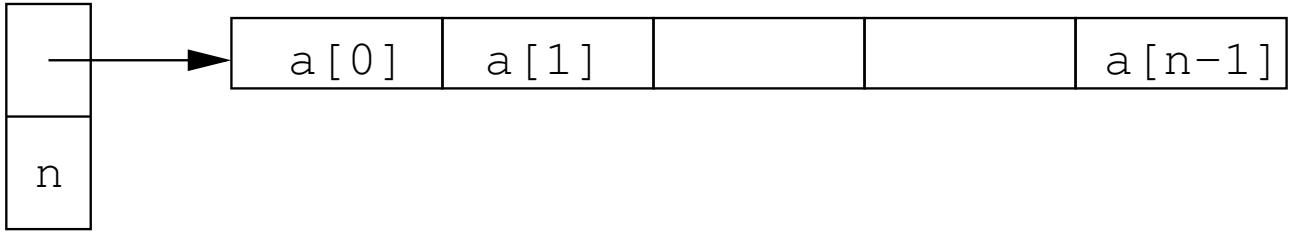
Bem.: Initialisierung von STL-Vektoren mittels geschweifter Klammern in C++98 *nicht* möglich.
Ab C++11 rekursive Initialisierungsregeln, Bestimmung der Vektorlänge(n) jeweils durch Elementanzahl

Vektoren in der Standard Template Library (STL)

Eigenschaften

- Vereinbarung: `vector<T> a(n)`
  a Vektor aus *n* Komponenten vom Typ *T*
- Dimension *n* (Komponentenzahl) zur Laufzeit änderbar
- Komponentenzugriff: `a[i]`
- Index *i*: vorzeichenlos Niedrigster Index: 0
  Indextyp: `vector<T>::size_type`
- Zuweisungen und Größenvergleiche möglich, falls in Datentyp *T* vorhanden
- Nur effizient realisierbare Funktionen implementiert, deshalb *kein* `push_front` und `pop_front`.

Skizze:



STL-Vektor aus n Komponenten

Vektoren in der Standard Template Library II

Caveat

- *Falsch:* `vector<T> a[n];`
  (*a* C-Vektor aus *n* leeren *T*-Vektoren)
  Richtig: `vector<T> a(n);`
- *Falsch:* `vector<T> a; a[i]=t;`
  (*a* leerer T-Vektor, führt in der Regel zu Programmabsturz)
  Richtig: `vector<T> a(n); a[i]=t; //` $n > i \geq 0$
  bzw.: `vector<T> a; a.resize(n); a[i]=t;`
- *Falsch:* `vector<T> a(n); a[n]=t;`
  (a hat die Komponenten a[0],. . . ,a[n-1])
  Richtig: `vector<T> a(n+1); a[n]=t;`
- *Keine* Überprüfung, ob beim Komponentenzugriff der Index innerhalb der Feldgrenzen liegt!
  Kann zu Programmabstürzen führen (segmentation fault, bus error) $\rightarrow$ Debuggereinsatz zur Fehlersuche

Vektoren in der Standard Template Library III

Matrizen

- STL-Vektoren können zum Aufbau von $m \times n$-Matrizen genutzt werden:
  `vector<vector<double>> a(m);`
  (m leere double-Vektoren)
  `for (int i=0; i<m; ++i) a[i].resize(n)`
  (m double-Vektoren der Länge n)
- Kürzere Alternative:
  `vector<vector<double>> a(m,vector<double>(n));`
- C++98: `vector<vector<double> > a` erforderl.
  (>> interpretiert als Shiftoperator, geändert ab C++11!)
- Zugriff auf Matrixelemente:
  `a[i][j]` // $0 \leq i < m$, $0 \leq j < n$
- Die so definierten Matrizen sind für numerische Zwecke nur bedingt geeignet:
  1. Dynamische Allozierung oft in Blöcken von Zweierpotenzen
  2. Matrixelemente nicht zusammenhängend gespeichert
  3. Optimierung des zweifach indirekten Zugriffs auf Matrixelemente erforderlich

Vektoren in der Standard Template Library IV

Initialisierung

- Initialisierunglisten mit geschweiften Klammern für STL-Vektoren erst ab C++11 möglich
- Bestimmung der Vektorlänge durch Elementanzahl

  Bsp.: `vector<double> a={1.0,4.0,0.0}`

  $a_0 = 1, a_1 = 4, a_2 = 0$
- Rekursive Anwendung der Initialisierungsregeln

  Bsp.: `vector<vector<int>> c={{1,2},{3}}`

  $c[0] \leftarrow \{1, 2\}$, d.h. $c_{00} = 1, c_{01} = 2$

  $c[1] \leftarrow \{3\}$, d.h. $c_{10} = 3$

  c *keine* Implementierung einer Rechtecksmatrix

  Bsp.: `vector<vector<int>> d={{4,2,1},{5,2,7}}`

  $d[0] \leftarrow \{4, 2, 1\}$, d.h. $d_{00} = 4, d_{01} = 2, d_{02} = 1$

  $d[1] \leftarrow \{5, 2, 7\}$, d.h. $d_{10} = 5, d_{11} = 2, d_{12} = 7$

  d Implementierung einer 2×3-Matrix