Modellierung und Programmierung

Dr. Martin Riplinger

5.12.2012

Zeiger und Felder

Zur Erinnerung:

- Der Name eines statischen Feldes ist zugleich ein Zeiger auf das erste Feldelement.
- Auf Zeiger kann der Index/Array-Operator `[]` angewandt werden, mit dem gleichen Effekt wie bei statischen Feldern.
- Statische Felder können anstelle von Zeigern auf den gleichen Typ an Funktionen übergeben werden.

  Beispiel: Der folgende Code ist semantisch korrekt und liefert wie erwartet die Ausgabe „`1. Element: 3.141500`“.

```
void zeigeErstesElement(
  double *array)
{
  printf("1. Element: %lf\n",
         array[0]);
  return;
}
```

```
int main(void)
{
  double x[2] = {3.1415, -1.0};
  zeigeErstesElement(x);

  return 0;
}
```

- Namen statischer Felder können als **konstante** Zeiger nicht umgebogen werden. Zuweisungen wie `feld1 = feld2;` schlagen fehl.

Call by reference: Beispiel

```
void vektor_ausgeben(double *vec,int n) // Funktion
{
  int i;
  printf("(");
  for(i=0;i<n-1;i++)
  printf(" %2.1lf ,",vec[i]);        //alternativ: *(vec+i)
  printf(" %2.1lf )\n",vec[n-1]);
}

void vektor_erhoehen(double *vec,int n,int summand) // Funktion
{
  int i;
  for(i=0;i<n;i++)
          vec[i] +=summand;
}

int main(void) // Hauptprogramm
{
  double vec[6]={1,1,1,1.5,1.5,1.5};
  vektor_ausgeben(vec,6);
  vektor_erhoehen(vec,6,2);
  vektor_ausgeben(vec,6);
}
```

Ausgabe:

Dynamische Speicherverwaltung

Problem: Mit den bisherigen Mitteln können nur Speicherobjekte angelegt werden, deren Größe **zur Compilezeit** bekannt ist.

Beispiel: Einfacher Texteditor

- **Umsetzung**: der aktuell bearbeitete Text wird zeilenweise in einem zweidimensionalen Feld (z. B. `char text[zeilen][spalten]`) gespeichert.
- **Frage 1**: Wie kann man im Vorfeld eine sinnvolle Zeilenbreite festlegen?
  Antwort: Gar nicht! Es sollte berücksichtigt werden, wie viele Zeichen *zum Zeitpunkt der Ausführung* auf dem Bildschirm in einer Zeile darstellbar sind.
- **Frage 2**: Wie viele Zeilen sollte das Feld umfassen, damit
  - einerseits ein Arbeiten mit Texten „gängiger" Länge möglich ist und
  - andererseits das Programm nicht unnötig viel Speicher verbraucht?

  Antwort: Offenbar lassen sich nicht beide Bedingungen gleichzeitig erfüllen!
- **Frage 3**: Ist es sinnvoll, diese Größe festzulegen, ohne zu berücksichtigen,
  - wie viel Speicher zur Verfügung steht,
  - wie lang der bearbeitete Text tatsächlich ist und
  - dass eventuell mehrere Texte gleichzeitig bearbeitet werden?

  Offensichtliche Antwort: Nein!

Fazit: Mit diesen Speicherobjekten kommt man nicht besonders weit!

Dynamische Speicherverwaltung

Naheliegende Lösung: Das *Programm* soll *zur Laufzeit* entscheiden, wann wieviel Speicher benötigt wird.

Programmiertechnische Umsetzung: mit Pointern!

- Im Code wird explizit ein Speicherblock einer bestimmten Größe angefordert, indem eine speziell dafür vorgesehene Funktion aufgerufen wird.
- Die Funktion versucht, den Speicher zu reservieren („**allokieren**“) und liefert bei Erfolg einen **Zeiger auf die Startadresse** des Speicherblocks, den sogenannten **Basis-Zeiger** (engl. *base pointer*).
- Wird der Speicher nicht mehr benötigt, ruft man eine weitere Funktion auf, um Block **explizit** wieder freizugeben.
- Der Basis-Zeiger ist die **einzige Referenz** auf den allokierten Speicherblock(, solange man keinen anderen Pointer ebenfalls dorthin zeigen lässt). Man darf ihn *auf keinen Fall verlieren* (umbiegen, Lebenszeit ablaufen lassen,. . .), denn sonst kann der Block nicht mehr freigegeben werden.
  $\rightsquigarrow$ „speicherfressende“ Programme

Funktionen zur Speicherverwaltung

Header: `#include <stdlib.h>`

Anforderung von Speicherplatz

- `void *malloc(size_t size);` (**mem**ory **alloc**ation)

  Fordert einen (uninitialisierten!) Speicherblock der Größe `size` Byte an und gibt einen Zeiger auf die Startadresse zurück. Bei Scheitern wir `NULL` zurückgegeben.

- `void *calloc(size_t anzahl, size_t size);` (**c**leared memory **alloc**ation)

  Fordert einen Speicherblock für `anzahl` Elemente der Größe `size` an und initialisiert mit Nullen. Ansonsten wie `malloc`.

Vergrößerung / Verkleinerung eines Speicherblocks

- `void *realloc(void *ptr, size_t size);`

  Ändert die Größe des Blocks zum Basis-Zeiger `ptr` auf `size` Byte. Der Inhalt des alten Blocks bleibt unverändert, neu angeforderter Speicher bleibt uninitialisiert. Rückgabewert ist der zum neuen Speicherbereich gehörige Base-Pointer, welcher sich von `ptr` unterscheiden kann(!). Bei Scheitern wird nichts unternommen.

Freigabe eines Speicherblocks

- `void free(void *ptr);`

  Gibt den Speicher frei, auf den `ptr` zeigt.

Funktionen zur Speicherverwaltung

Beispiel

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int i;
  double *dyn_feld;

  dyn_feld = (double *) malloc(5 * sizeof(double));
             // Cast!        // Platz fuer 5 double-Eintraege

  for(i = 0; i < 5; i++)     // Initialisierung
    dyn_feld[i] = 2.0 * i;
  for(i = 0; i < 5; i++)     // Ausgabe
    printf("dyn_feld[%d] = %lf\n", i, dyn_feld[i]);

  dyn_feld = (double *) realloc(dyn_feld, 8 * sizeof(double));
                             // Vergroesserung auf 8 double-Eintraege
  for(i = 5; i < 8; i++)
    dyn_feld[i] = 2.0 * i;   // Initialisierung der neuen Eintraege
  for(i = 0; i < 8; i++)
    printf("dyn_feld[%d] = %lf\n", i, dyn_feld[i]);

  free(dyn_feld); dyn_feld = NULL;   // Vorsichtsmassnahme

  return 0;
}
```

Der `sizeof`-Operator

Der Speicherbedarf wird niemals direkt als „magische“ Zahl angegeben, sondern immer mit Hilfe des `sizeof`-Operators berechnet.

Erinnerung:

`sizeof(Datentyp)` oder `sizeof(Speicherobjekt)`

liefert den Speicherbedarf eines allgemeinen Datentyps oder eines deklarierten Speicherobjekts in Byte.

Achtung: Bei **statischen** Feldern liefert `sizeof` die **tatsächliche Größe des Feldes**, bei **dynamischen** Feldern hingegen nur den **Speicherbedarf der Pointervariable**. Mit `sizeof` lässt sich die Größe eines dynamischen Feldes nicht feststellen!

Bemerkung: Da Pointer Adressen speichern, ist `sizeof(Datentyp *)` unabhängig vom Datentyp immer gleich `sizeof(size_t)`.

Typische Fehler bei der Speicherverwaltung

- Nicht mehr benötigter Speicher wird vergessen freizugeben.

```
double berechnung(double *vektor, int n)
{
  double ergebnis,
         *zw_speicher = (double *) malloc(n*n * sizeof(double));
  //  Hier wird etwas berechnet
  return ergebnis;
}
```

Folge: Speicher bleibt bis zum Ende des Programms belegt, schlimmstenfalls wird der gesamte Speicher aufgebraucht und das System geht in die Knie.

⇝ Speicher **so früh wie möglich** und **in jedem Fall** wieder freigeben

- Auf bereits freigegebenen Speicher wird zugegriffen.

```
int *arr = (int *) malloc(2 * sizeof(int));
free(arr);

arr[0] = -24; arr[1] = 42;
printf("ptr2[0] = %d\n", arr[0]);
printf("ptr2[1] = %d\n", arr[1]);
```

Folge: Möglicherweise stehen dort noch immer die gleichen Werte, doch dafür gibt es keine Garantie. Vor einem solchen Zugriff wird **nicht gewarnt**.

⇝ Basis-Pointer von freigegebenen Blöcken auf `NULL` setzen

Typische Fehler bei der Speicherverwaltung

- Ein umgebogener Base-Pointer wird in `free()` verwendet.

```
int i, *arr = (int *) malloc(20 * sizeof(int));

for (i = 0; i < 20; i++)
  *(arr++) = 5 * i;

free(arr);
```

Folge: Speicherzugriffsfehler

- Es werden statische statt dynamischer Felder verwendet.

```
int feld[5], *ptr;

ptr = feld;    // Alternativer Zugriff
ptr[4] = 42;

// Code

free(ptr);
```

Folge: Speicherzugriffsfehler

Dynamische Speicherverwaltung

Speicherverwaltung während der Programmlaufzeit:

Speicherbereich	Verwendung
Code	Maschinencode des Programms
Daten-Segment	Statische und globale Variablen
Stack (dt. Stapel)	Funktionsaufrufe und lokale Variablen
Heap (dt. Halde)	Dynamisch reservierter Speicher

Variablentypen

Lokale, globale und statische Variablen

- Name und Typ werden im Programmcode durch eine Deklaration festgelegt.
- Variablen werden direkt über ihren Namen angesprochen.
- Scope und Lifetime sind durch die statische Struktur des Programms festgelegt.
- Speicherverwaltung erfolgt implizit:
  - Lokale Variable werden im **Stack** angelegt.
  - Globale und statische Variablen werden im **Daten-Segment** angelegt.

Dynamische Variablen

- Erscheinen in keiner Deklaration, tragen keinen Namen.
- Variablen werden über ihre Adressen mit einem Pointer angesprochen.
- Scope und Lifetime folgen nicht den Regeln statischer Variablen.
- Speicherverwaltung im **Heap** geschieht explizit im Programm.

Dynamische Implementierung von Vektoren

Vorüberlegung: Vektoren $x \in \mathbb{R}^n$ lassen sich offenbar als Felder `double x[n]` implementieren. Um flexibler zu sein, bietet sich eine Implementierung als dynamisches Feld `double *x` in Verbindung mit den Speicherverwaltungsfunktionen an.

Erzeugung eines neuen Vektors:

```
double *neuerVektor(int laenge)
{
  double *v;
  v = (double *) malloc(laenge * sizeof(double));

  return v; // Bei Scheitern automatisch NULL
}
```

Erzeugung eines Nullvektors:

```
double *neuerNullvektor(int laenge)
{
  double *v;
  v = (double *) calloc(laenge, sizeof(double));

  return v; // Bei Scheitern automatisch NULL
}
```

Dynamische Implementierung von Vektoren

Ausgabe am Bildschirm:

```
void zeigeVektor(double *vektor, int laenge)
{
  int i;

  for (i = 0; i < laenge; i++)
    printf("%lf\n", vektor[i]);

  return;
}
```

Löschung:

```
void loescheVektor(double *vektor)
{
  free(vektor);

  return;
}
```

Dynamische Implementierung von Vektoren

Kopie anlegen:

```
void kopiereVektor(double *ziel, double *quelle, int laenge)
{
  int i;
  for (i = 0; i < laenge; i++)
    ziel[i] = quelle[i];

  return;
}
```

Weitere Funktionen:

- Arithmetische Operationen mit Vektoren (elementweise)
- Skalierung, Addition einer Konstanten
- (Betragsmäßig) maximales / minimales Element finden
- Untervektoren: Block, jedes n-te Element
- Permutationen

Minimales Verwendungsbeispiel

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  double *neuerVektor(int laenge);
5  void zeigeVektor(double *vektor, int laenge);
6  void loescheVektor(double *vektor);
7  void kopiereVektor(double *ziel, double *quelle,
8                     int laenge);
9
10 int main(void)
11 {
12   int i, N = 10;
13   double *v = neuerVektor(N), *w = neuerVektor(N);
14
15   if (v==NULL || w==NULL)
16     {
17      printf("Zu wenig freier Speicher!");
18      free(v);free(w);
19      return 1;
20     }
21
22   for (i = 0; i < N; i++)
23     v[i] = 2.0 * i;
24
25   kopiereVektor(w, v, N);
26   printf("w =\n");
27   zeigeVektor(w, N);
28   loescheVektor(v); loescheVektor(w);
29   return 0;
30 }
31 // Hier kommen noch die Funktionsdefinitionen
```

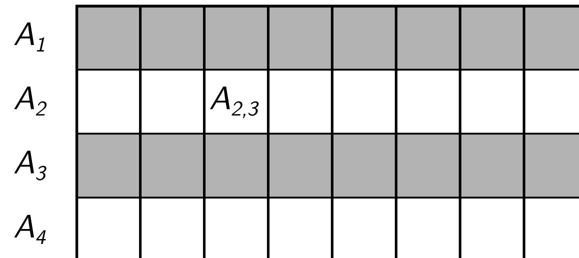
Ausgabe:

```
w =
0.000000
2.000000
4.000000
6.000000
8.000000
10.000000
12.000000
14.000000
16.000000
18.000000
```

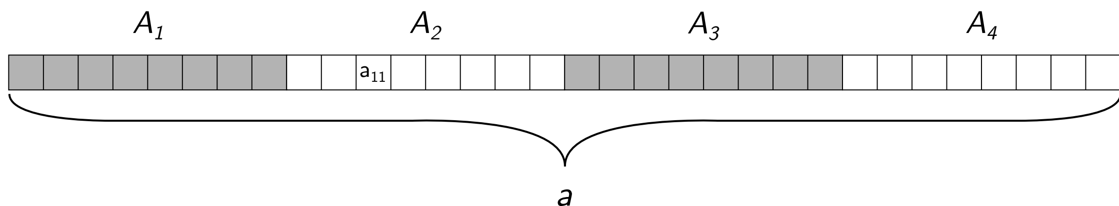
Dynamische Implementierung von Matrizen

Vorüberlegungen:

- Felder in C sind eindimensional, Matrizen jedoch (mindestens) zweidimensional.
- Jede Zeile einer 2D-Matrix kann als 1D-Vektor aufgefasst werden.
- **Idee 1**: Eine Matrix $A \in \mathbb{R}^{m \times n}$ wird als **Feld von Vektoren** $A_i \in \mathbb{R}^n$ interpretiert.

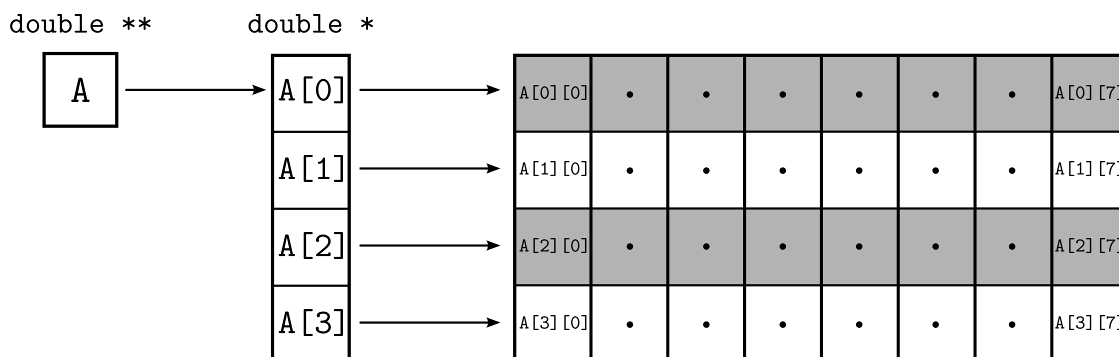


- **Idee 2**: Die Zeilen einer Matrix $A \in \mathbb{R}^{m \times n}$ werden aneinandergereiht, so dass ein langer Vektor $a \in \mathbb{R}^{m \cdot n}$ entsteht. Es gilt der Zusammenhang $A_{i,j} = a_{(i-1)n+(j-1)}$.



Dynamische Implementierung von Matrizen

Umsetzung in C – Variante 1: Beispiel $A \in \mathbb{R}^{4 \times 8}$



- Ein Vektor `v` ist ein (dynamisches) Feld vom Typ `double`, implementiert als `double *v`.
- Folglich ist eine Matrix `A` ein (dynamisches) **Feld von Vektoren** `A[i]` vom Typ `double *`.
- Technisch gesehen ist `A` ein **Zeiger auf** `double *`, und jedes `A[i]` ist ein **Zeiger auf** `double`.
- Deklaration: `double **A;`
- Zugriff auf Feldelemente: $A_{i,j} \triangleq$ `A[i-1][j-1]`

Dynamische Implementierung von Matrizen: Funktionen

Erzeugung einer Matrix:

```
double **neueMatrix(int n_zeilen, int n_spalten)
{
  int i, j;
  double **matrix;

  matrix = (double **) malloc(n_zeilen * sizeof(double *));
  if (matrix == NULL) // Speicheranforderung gescheitert
    return NULL;

  for (i = 0; i < n_zeilen; i++)
  {
    matrix[i] = (double *) malloc(n_spalten * sizeof(double));
    if (matrix[i] == NULL)
    {
      for (j = 0; j < i; j++)
        free(matrix[j]);

      free(matrix);
      return NULL;
    }
  }

  return matrix;
}
```

Dynamische Implementierung von Matrizen: Funktionen

- Bei der Speicheranforderung für die Matrix (`double **`) muss der Größenmultiplikator `sizeof(double *)` sein, denn jeder Eintrag in `matrix` ist schließlich ein `double *`!

```
matrix = (double **) malloc(N_zeilen * sizeof(double *));
```

- Scheitert die Anforderung, wird `NULL` zurückgegeben.

```
if (matrix == NULL)
  return NULL;
```

- Ist die Speicheranforderung für eine Zeile `A[i]` erfolglos, so muss der bisher reservierte Speicher - also sämtliche Zeilenvektoren `A[j] (j < i)` sowie `matrix` selbst - wieder freigegeben werden, bevor die Funktion mit Rückgabewert `NULL` endet.

```
if (matrix[i] == NULL)
{
  for (j = 0; j < i; j++)
    free(matrix[j]);

  free(matrix);
  return NULL;
}
```

Dynamische Implementierung von Matrizen: Funktionen

Ausgabe einer Matrix auf dem Bildschirm:

```
1   void zeigeMatrix(double **matrix, int n_zeilen, int n_spalten)
2   {
3     int i, j;
4
5     for (i = 0; i < n_zeilen; i++)
6     {
7       for (j = 0; j < n_spalten; j++)
8         printf("%.4lf ", matrix[i][j]);
9
10      printf("\n");
11    }
12
13    return;
14  }
```

Beachte:

- Die Ausgabe auf der Kommandozeile funktioniert nur zeilenweise.
- Nach dem Ende der Zeile den Newline-Character `'\n'` nicht vergessen!

Dynamische Implementierung von Matrizen: Funktionen

Löschung einer Matrix:

```
1   void loescheMatrix(double **matrix, int n_zeilen)
2   {
3     int i;
4
5     for (i = 0; i < n_zeilen; i++)
6       free(matrix[i]);
7
8     free(matrix);
9   }
```

Beachte:

- Es genügt nicht, das übergeordnete Feld `matrix` zu befreien. Jedes Feld `matrix[i]` muss einzeln an `free()` übergeben werden.
- Sicherheitshalber sollte nach dem Aufruf `loescheMatrix(A, n);` noch die Zuweisung `A = NULL;` vorgenommen werden, um zukünftige Zugriffe auf den freigegebenen Bereich zu verhindern.

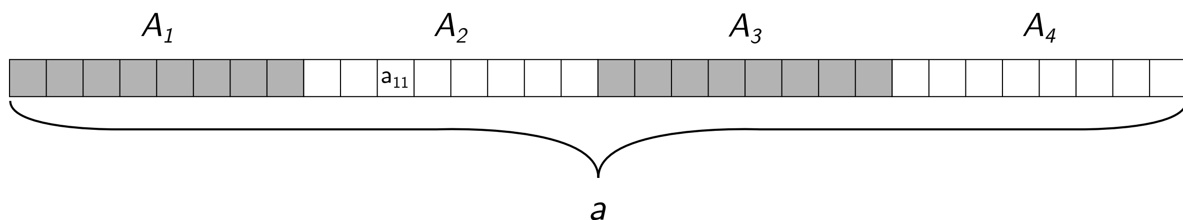
  Frage: Wie könnte man diese Zuweisung in die Funktion `loescheMatrix()` einbauen?

Anwendungsbeispiel

```
#include<stdio.h>
#include<stdlib.h>

double **neueMatrix(int n_zeilen, int n_spalten);
void zeigeMatrix(double** matrix, int n_zeilen, int n_spalten);
void loescheMatrix(double** matrix, int n_zeilen);

int main(void)
{
  int i, j, n = 10;
  double **A;

  A = neueMatrix(n, n);

  for (i = 0; i < n; i++)
  {
    for (j = 0; j < n; j++)
      A[i][j] = i+0.1*j;
  }

  zeigeMatrix(A, n, n);
  loescheMatrix(A, n);

  return 0;
}

// Ab hier noch die Funktionsdefinitionen
```

Dynamische Implementierung von Matrizen

Umsetzung in C – Variante 2:



- Deklaration: `double *a;`
- Als Funktionen zum Erzeugen und Löschen einer Matrix können die Vektor-Funktionen `neuerVektor()` und `loescheVektor()` verwendet bzw. umbenannt werden.
- Zum Zugriff auf das Element $A_{i,j}$ verwendet man den Ausdruck `a[i * n_spalten + j]`.
- Umgekehrt lassen sich Zeilen- und Spaltenindex in `a[index]` berechnen mittels `i = index / n_spalten; j = index % n_spalten;`
- Vorteil gegenüber Doppelpointer-Variante: Vermeidung von Doppelschleifen → schneller bei Operationen auf der ganzen Matrix und bei Multiplikation. Viele numerische Programmbibliotheken (GSL, BLAS, NumPy,. . .) verwenden intern diese Darstellung.
- Nachteil: Operationen basierend auf Zeilen- und Spaltenindex (z. B. transponieren) sind wegen des zusätzlichen Berechnungsschritts aufwändiger.