Einführung in die Programmierung für Physiker

Die Programmiersprache C – Zeiger und Arrays

Marc Wagner

Institut für theoretische Physik
Johann Wolfgang Goethe-Universität Frankfurt am Main

WS 2013/14

Adressen

- Der Speicher eines Computers besteht (im einfachsten Fall) aus fortlaufend nummerierten Speicherzellen (jeweils 1 Byte = 8 Bits).
- Die Nummerierung einer Speicherzelle bezeichnet man als ihre **Adresse**.
- Oft werden Speicherzellen auch in zusammenhängenden Gruppen verarbeitet, z.B. häufig 4 Byte für eine **int**-Variable oder 8 Byte für eine **double**-Variable; die Adresse einer Variable entspricht der Nummerierung ihrer ersten Speicherzelle.
- Der **Adressoperator &** liefert auf eine Variable angewendet deren Adresse.

```
 1. #include<stdio.h>
 2.
 3. int main(void)
 4. {
 5.   int i1, i2;
 6.   double d1, d2;
 7.
 8.   printf("sizeof(int) = %zd\n", sizeof(int));
 9.   printf("sizeof(double) = %zd\n", sizeof(double));
10.
11.   printf("Adresse von i1: %p   Adresse von i2: %p\n", &i1, &i2);
12.   printf("Adresse von d1: %p   Adresse von d2: %p\n", &d1, &d2);
```

```
13.
14.   printf("Adresse von i1: %lu   Adresse von i2: %lu\n", (unsigned long)&i1, (unsigned long)&i2);
15.   printf("Adresse von d1: %lu   Adresse von d2: %lu\n", (unsigned long)&d1, (unsigned long)&d2);
16. }
```

```
sizeof(int) = 4
sizeof(double) = 8
Adresse von i1: 0x7fff15f11b68 Adresse von i2: 0x7fff15f11b6c
Adresse von d1: 0x7fff15f11b70 Adresse von d2: 0x7fff15f11b78
Adresse von i1: 140733561510760 Adresse von i2: 140733561510764
Adresse von d1: 140733561510768 Adresse von d2: 140733561510776
```

```
sizeof(int) = 4
sizeof(double) = 8
Adresse von i1: 0x7fff210e0058 Adresse von i2: 0x7fff210e005c
Adresse von d1: 0x7fff210e0060 Adresse von d2: 0x7fff210e0068
Adresse von i1: 140733747953752 Adresse von i2: 140733747953756
Adresse von d1: 140733747953760 Adresse von d2: 140733747953768
```

- **`sizeof(`***`type`***`)`** liefert die Anzahl der Bytes, die eine Variable oder ein Wert vom Datentyp *type* belegt; das Ergebnis ist vom Datenttyp **`size_t`** (häufig äquivalent zu **`unsigned long`**), dem im Formatstring **`%zd`** entspricht.
- Zur Ausgabe von Adressen dient im Formatstring **`%p`**; die Darstellung ist compilerabhängig (im obigen Beispiel **hexadezimal**).

Zeiger

- **Zeiger** sind Variablen, die Adressen enthalten.
- Zeiger stellen eigene Datentypen dar, z.B.
  - ist ein Zeiger auf eine **int**-Variable vom Typ **int ***,
  - ist ein Zeiger auf eine **double**-Variable vom Typ **double ***,
  - ist ein Zeiger auf einen **int**-Zeiger (d.h. auf eine **int ***-Variable) vom Typ **int ****,
  - ...

```
 1. #include<stdio.h>
 2.
 3. int main(void)
 4. {
 5.   int i;
 6.   int *pi;  // int-Zeiger.
 7.   int **ppi;  // int *-Zeiger.
 8.   double d;
 9.   double *pd;  // double-Zeiger.
10.
11.   pi = &i;
12.   ppi = &pi;
```

```
13.
14.   pd = &d;
15.
16.   printf("Adresse von i:  %p\n", pi);
17.   printf("Adresse von pi: %p\n", ppi);
18.   printf("Adresse von d:  %p\n", pd);
19. }
```

```
Adresse von i:  0x7fffcc161bfc
Adresse von pi: 0x7fffcc161c00
Adresse von d:  0x7fffcc161c08
```

- Der **Inhaltsoperator** * liefert, auf einen Zeiger angewendet, die Variable, auf die der Zeiger verweist.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   int i;
6.   int *pi;
7.
8.   pi = &i;  // Der Zeiger pi erhaelt die Adresse von i.
9.
10.  i = 5;
11.  printf("i und *pi sind aequivalent: %d %d\n", i, *pi);
12.
13.  *pi = 7;
```

```
14.   printf("i und *pi sind aequivalent: %d %d\n", i, *pi);
15. }
```

```
i und *pi sind aequivalent: 5 5
i und *pi sind aequivalent: 7 7
```

Zeiger als Parameter von Funktionen

- Zeiger als Parameter von Funktionen sind notwendig, wenn eine Funktion Variablen verändern soll.
- **Beispiel:** eine Funktion, die die Werte der `int`-Variablen **`a`** und **`b`** vertauscht ...
  - Erster Versuch (**fehlerhaftes Vorgehen, d.h. Funktion macht nicht das, was sie soll**) ...

```
 1. #include<stdio.h>
 2.
 3. void swap(int a, int b)
 4. {
 5.   int tmp = a;
 6.   a = b;
 7.   b = tmp;
 8. }
 9.
10. int main(void)
11. {
12.   int i1 = 3;
13.   int i2 = 7;
14.
15.   printf("i1 = %d, i2 = %d\n", i1, i2);
```

```
16.   swap(i1, i2);
17.   printf("i1 = %d, i2 = %d\n", i1, i2);
18. }
```

```
i1 = 3, i2 = 7
i1 = 3, i2 = 7
```

- Funktioniert nicht, weil **C** bei Funktionsaufrufen nicht Variablen, sondern Werte von Variablen (bzw. Konstanten) übergibt (**call by value**).
- Im obigen Beispiel wird beim Funktionsaufruf von **swap** der Wert von **i1** der Variable **a** und der Wert von **i2** der Variable **b** zugewiesen; die Werte von **a** und **b** werden in der Funktion **swap** vertauscht; **i1** und **i2** bleiben davon aber unberührt.

- Zweiter Versuch (**korrektes Vorgehen, d.h. Funktion macht das, was sie soll**)

  ...

```
1. #include<stdio.h>
2.
3. void swap(int *pa, int *pb)
4. {
5.   int tmp = *pa;
6.   *pa = *pb;
7.   *pb = tmp;
8. }
9.
10. int main(void)
```

```
11. {
12.   int i1 = 3;
13.   int i2 = 7;
14.
15.   printf("i1 = %d, i2 = %d\n", i1, i2);
16.   swap(&i1, &i2);
17.   printf("i1 = %d, i2 = %d\n", i1, i2);
18. }
```

```
i1 = 3, i2 = 7
i1 = 7, i2 = 3
```

- Die Adresse der Variable **i1** wird beim Funktionsaufruf von **swap** der Zeigervariable **pa** und die Adresse der Variable **i2** der Zeigervariable **pb** zugewiesen; die Funktion **swap** kann mit Hilfe dieser Adressen und dem Inhaltsoperator * die Werte der Variablen **i1** und **i2** verändern (***pa** ist äquivalent zu **i1**, ***pb** ist äquivalent zu **i2**).

- **Beispiel:** Einlesen von Werten mit **scanf** ...

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   int i;
6.   printf("i eingeben: ");
7.   scanf("%d", &i);
8.   printf("i = %d\n", i);
```

```
9. }
```

```
i eingeben: 123
i = 123
```

- **Vorsicht! Häufige Fehlerquelle ...**
  Nicht-initialisierter (oder nicht mehr aktueller) Zeiger verwendet ... führt häufig zum Programmabsturz.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   int *p_i;
6.   printf("i eingeben: ");
7.   scanf("%d", p_i);
8.   printf("i = %d\n", *p_i);
9. }
```

```
i eingeben: 123
Speicherzugriffsfehler (Speicherabzug geschrieben)
```

- Dieses Negativbeispiel kann leicht korrigiert werden.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   int *p_i;
```

```
  int i;  // Eine int-Variable wird angelegt.
  p_i = &i;  // Dem Zeiger p_i wird die Adresse der int-Variable i zugewiesen.
  printf("i eingeben: ");
  scanf("%d", p_i);
  printf("i = %d\n", *p_i);
}
```

```
i eingeben: 123
i = 123
```

- Zeiger als Parameter von Funktionen sind hilfreich, wenn eine Funktion mehrere Größen berechnet, also es nicht ausreicht, einen einzelnen Wert zurückzugeben.
- **Beispiel:** Determinante und Spur einer 2×2-Matrix ...

```
#include<stdio.h>

// Berechnet die Determinante der 2x2-Matrix M.
double det_M(double m00, double m01, double m10, double m11)
{
  return m00*m11 - m10*m01;
}

// Berechnet die Determinante und die Spur der 2x2-Matrix M.
void det_tr_M(double m00, double m01, double m10, double m11, double *p_det, double *p_tr)
{
  *p_det = m00*m11 - m10*m01;
  *p_tr = m00 + m11;
```

```
14. }
15.
16. int main(void)
17. {
18.   // Verwende im Folgenden die Matrix
19.   //   | 1.0 2.0 |
20.   //   | 3.0 4.0 |.
21.
22.   printf("det = %f\n", det_M(1.0, 2.0, 3.0, 4.0));
23.
24.   double d1, d2;
25.   det_tr_M(1.0, 2.0, 3.0, 4.0, &d1, &d2);
26.   printf("det = %f   tr = %f\n", d1, d2);
27. }
```

```
det = -2.000000
det = -2.000000   tr = 5.000000
```

Zeiger und const

- Was bedeutet
  `const int *`?
  Ein konstanter Zeiger auf `int`? Oder ein Zeiger auf `const int`?
  → Es handelt sich um einen Zeiger auf `const int`.
- Äquivalent dazu ist
  `int const *`.
- Einen konstanter Zeiger auf `int` bezeichnet
  `int * const`.
- **Merkhilfe:** `const` bezieht sich immer auf das, was links davon steht (es sei denn es steht selbst ganz links).

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   int a = 3;
6.   int b = 5;
7.
8.   // Zeiger auf Datentyp "const int".
9.
```

```
10.   const int *p1 = &a;
11.   printf("*p1 = %d\n", *p1);
12.   p1 = &b;
13.   printf("*p1 = %d\n", *p1);
14.   // *p1 = 7;  // --> Kompiler liefert Fehlermeldung.
15.
16.   printf("********************\n");
17.
18.   int const *p2 = &a;
19.   printf("*p2 = %d\n", *p2);
20.   p2 = &b;
21.   printf("*p2 = %d\n", *p2);
22.   // *p2 = 7;  // --> Kompiler liefert Fehlermeldung.
23.
24.   printf("********************\n");
25.
26.   // Konstanter Zeiger auf Datentyp "int".
27.
28.   int * const p3 = &a;
29.   printf("*p3 = %d\n", *p3);
30.   // p3 = &b;  // --> Kompiler liefert Fehlermeldung.
31.   *p3 = 7;
32.   printf("*p3 = %d   a = %d\n", *p3, a);
33. }
```

```
*p1 = 3
*p1 = 5
********************
```

```
*p2 = 3
*p2 = 5
********************
*p3 = 3
*p3 = 7   a = 7
```

Arrays

Eindimensionale Arrays

- In Zeile 5 des folgenden Beispiels werden 10 Variablen vom Typ **`int`** definiert, ein sogenanntes **Array** (auch **Vektor**); diese Variablen (die **Elemente des Arrays**) tragen die Namen **`a[0]`**, **`a[1]`**, ..., **`a[9]`**.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   int a[10];
6.
7.   a[0] = 3;
8.   a[1] = 4;
9.   a[6] = -2;
10.
11.  printf("a[0] * a[1] + a[6] = %d\n", a[0] * a[1] + a[6]);
12. }
```

```
a[0] * a[1] + a[6] = 10
```

- Die Anzahl der Elemente eines Arrays müssen bei dessen Definition durch eine Konstante festgelegt werden.

```
1. ...
2.   // o.k.
3.   #define N 10
4.   int a[N];
5. ...
```

```
1. ...
2.   // o.k.
3.   const int n = 10;
4.   int a[n];
5. ...
```

```
1. ...
2.   // !!! falsch !!!
3.   int n = 10;
4.   int a[n];
5. ...
```

- Beispiel: Array **`double a[n];`** ...
  - Der Wert von **`a`** ist die Adresse des Anfangselements, also die Adresse von **`a[0]`**.
  - **`a`** ähnelt damit einem Zeiger auf das Anfangselement.
  - Zuweisungen an **`a`**, also **`a = ...`**, sind jedoch nicht möglich.

```
1. #include<stdio.h>
2.
```

```
3. int main(void)
4. {
5.   const int n = 10;
6.   double a[n];
7.
8.   printf("a = %p   (&a[0] = %p)\n", a, &a[0]);
9. }
```

```
a = 0x7fff77284d90   (&a[0] = 0x7fff77284d90)
```

- Einfache und übersichtliche Initialisierung mit **{***value1***, ***value2***, ...}**.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   double a[4] = {1.0, 2.0, 3.0, 4.0};
6.   printf("a[0] = %f   a[1] = %f   a[2] = %f   a[3] = %f\n", a[0], a[1], a[2], a[3]);
7. }
```

```
a[0] = 1.000000   a[1] = 2.000000   a[2] = 3.000000   a[3] = 4.000000
```

Mehrdimensionale Arrays

- Definition von mehrdimensionalen Arrays und Zugriff auf dessen Elemente durch mehrfache verwendung von **`[...]`**, z.B. **`int B[5][10];`** oder **`x[3][0][7] = ...;`**.

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

```
 2.
 3. int main(void)
 4. {
 5.   int i1, i2;
 6.
 7.   const int n = 2;
 8.   double A[n][n];  // Matrix der Groesse n x n.
 9.   A[0][0] = 1.0;
10.   A[0][1] = 2.0;
11.   A[1][0] = 3.0;
12.   A[1][1] = 4.0;
13.
14.   for(i1 = 0; i1 < n; i1++)
15.     {
16.       printf("| ");
17.
18.       for(i2 = 0; i2 < n; i2++)
19.         printf("%+6.3lf ", A[i1][i2]);
20.
21.       printf("|\n");
22.     }
23. }
```

```
| +1.000 +2.000 |
| +3.000 +4.000 |
```

- Beispiel: Zweidimensionales Array **`double A[n][n];`** ...

- Der Wert von **A** ist die Adresse des Anfangselements, also die Adresse von **A[0][0]**.
- Der Wert von **A[i]** ist die Adresse von **A[i][0]**.
- **A** und **A[i]** ähneln damit Zeigern.
- Zuweisungen an **A** oder **A[i]**, also **A = ...** oder **A[i] = ...**, sind jedoch nicht möglich.

```c
#include<stdio.h>

int main(void)
{
  const int n = 2;
  double A[n][n];  // Matrix der Groesse n x n.

  printf("A = %p   (&A[0][0] = %p)\n", A, &A[0][0]);
  printf("A[0] = %p   (&A[0][0] = %p)\n", A[0], &A[0][0]);
  printf("A[1] = %p   (&A[1][0] = %p)\n", A[1], &A[1][0]);
}
```

```
A = 0x7fff826bb730   (&A[0][0] = 0x7fff826bb730)
A[0] = 0x7fff826bb730   (&A[0][0] = 0x7fff826bb730)
A[1] = 0x7fff826bb740   (&A[1][0] = 0x7fff826bb740)
```

- Einfache und übersichtliche Initialisierung mit geschachtelter Verwendung von **{...}** analog zum eindimensionalen Fall.

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

```
 2.
 3. int main(void)
 4. {
 5.   int i1, i2;
 6.
 7.   const int n = 2;
 8.   double A[n][n] = { {1.0, 2.0},
 9.                      {3.0, 4.0} }; // Matrix der Groesse n x n.
10.
11.   for(i1 = 0; i1 < n; i1++)
12.     {
13.       printf("| ");
14.
15.       for(i2 = 0; i2 < n; i2++)
16.         printf("%+6.3lf ", A[i1][i2]);
17.
18.       printf("|\n");
19.     }
20. }
```

```
| +1.000 +2.000 |
| +3.000 +4.000 |
```

Zeigerarithmetik

- Beispiel: Array `double a[n];` und Zeiger `double *pa = a;` ...
  - Der Wert von `a` und von `pa` ist die Adresse von `a[0]`.
  - Der Wert von `a+1` und von `pa+1` ist die Adresse von `a[1]`.
  - Der Wert von `a+2` und von `pa+2` ist die Adresse von `a[2]`.
  - ...
  - Addition und Subtraktion in Kombination mit einem Array oder einem Zeiger verändern die entsprechende Adresse also um Vielfache von `sizeof(type)`, im betrachteten Beispiel um Vielfache von `sizeof(double)` (**Zeigerarithmetik**).
  - Analog kann mit `pa[0]`, `pa[1]`, `pa[2]`, ... auf die Elemente `a[0]`, `a[1]`, `a[2]`, ... zugegriffen werden.
  - Die Ausdrücke `&a[n]`, `&pa[n]`, `a+n`, `pa+n` sind äquivalent.
  - Die Ausdrücke `a[n]`, `pa[n]`, `*(a+n)`, `*(pa+n)` sind äquivalent.
  - Für Zeiger sind Ausdrücke wie `pa += 1`, `pa -= 3`, `pa++`, `--pa`, ... ebenfalls zulässig, nicht jedoch für Arraynamen, also für `a`.
- Der Name eines Arrays ähnelt einem Zeiger auf dessen Anfangselement (Unterschied:

Zuweisungen sind nur bei Zeigern erlaubt, nicht bei Arraynamen).

```c
#include<stdio.h>

int main(void)
{
  int i1;

  double a[4] = {1.0, 2.0, 3.0, 4.0};
  double *pa = a;

  for(i1 = 0; i1 < 4; i1++)
    {
      printf("&a[%d] = %p   &pa[%d] = %p   a+%d = %p   pa+%d = %p\n",
             i1, &a[i1], i1, &pa[i1], i1, a+i1, i1, pa+i1);
    }

  for(i1 = 0; i1 < 4; i1++)
    {
      printf("a[%d] = %f   pa[%d] = %f   *(a+%d) = %f   *(pa+%d) = %f\n",
             i1, a[i1], i1, pa[i1], i1, *(a+i1), i1, *(pa+i1));
    }

  pa = a+2;
  printf("pa[-2] = %f   pa[-1] = %f   pa[0] = %f   pa[1] = %f\n", pa[-2], pa[-1], pa[0], pa[1]);
}
```

```
&a[0] = 0x7fffe8a55a40   &pa[0] = 0x7fffe8a55a40   a+0 = 0x7fffe8a55a40   pa+0 = 0x7fffe8a55a40
&a[1] = 0x7fffe8a55a48   &pa[1] = 0x7fffe8a55a48   a+1 = 0x7fffe8a55a48   pa+1 = 0x7fffe8a55a48
&a[2] = 0x7fffe8a55a50   &pa[2] = 0x7fffe8a55a50   a+2 = 0x7fffe8a55a50   pa+2 = 0x7fffe8a55a50
&a[3] = 0x7fffe8a55a58   &pa[3] = 0x7fffe8a55a58   a+3 = 0x7fffe8a55a58   pa+3 = 0x7fffe8a55a58
a[0] = 1.000000   pa[0] = 1.000000   *(a+0) = 1.000000   *(pa+0) = 1.000000
a[1] = 2.000000   pa[1] = 2.000000   *(a+1) = 2.000000   *(pa+1) = 2.000000
a[2] = 3.000000   pa[2] = 3.000000   *(a+2) = 3.000000   *(pa+2) = 3.000000
a[3] = 4.000000   pa[3] = 4.000000   *(a+3) = 4.000000   *(pa+3) = 4.000000
pa[-2] = 1.000000   pa[-1] = 2.000000   pa[0] = 3.000000   pa[1] = 4.000000
```

- **Vorsicht! Häufige Fehlerquelle ...**
  Zugriff auf Elemente jenseits der Arraygrenzen ... wird nicht vom Compiler erkannt, führt häufig zu falschen Ergebnissen oder zum Programmabsturz.

```c
#include<stdio.h>

int main(void)
{
  double a[4] = {1.0, 2.0, 3.0, 4.0};
  double b[4] = {5.0, 6.0, 7.0, 8.0};

  // Zugriff auf Elemente jenseits der oberen Arraygrenze a[3].
  printf("a[4] = %f   a[5] = %f\n", a[4], a[5]);
}
```

```
a[4] = 5.000000   a[5] = 6.000000
```

Übergeben von Arrays an Funktionen

- Eindimensionale Arrays:
  - Steht die Größe des Arrays fest, z.B. 2 Elemente, kann das Array auf drei äquivalenten Wegen einer Funktion übergeben werden, z.B. als Parameter **`double v[2]`**, **`double v[]`** oder **`double *v`** (beachte erneut die Ähnlichkeit von Arraynamen und Zeigern).

```
1. #include<stdio.h>
2.
3. // ********************
4.
5. // Ausgabe eines Vektors v mit 2 Elementen.
6. void print_vector_2(double v[2])
7. // void print_vector_2(double v[])   ... ist aequivalent.
8. // void print_vector_2(double *v)   ... ist aequivalent.
9. {
10.   printf("| ");
11.
12.   for(int i = 0; i < 2; i++)
13.     printf("%+6.3lf ", v[i]);
14.
15.   printf("|\n");
16. }
```

```
17.
18. // ********************
19.
20. // Ausgabe eines Vektors v mit 4 Elementen.
21. void print_vector_4(double v[4])
22. {
23.   printf("| ");
24.
25.   for(int i = 0; i < 4; i++)
26.     printf("%+6.3lf ", v[i]);
27.
28.   printf("|\n");
29. }
30.
31. // ********************
32.
33. int main(void)
34. {
35.   double x[4] = { 1.0, 2.0, 3.0, 4.0 };
36.   print_vector_4(x);
37.
38.   double y[2] = { +1.5, -2.6 };
39.   print_vector_2(y);
40. }
```

```
| +1.000 +2.000 +3.000 +4.000 |
| +1.500 -2.600 |
```

- Häufig ist es eleganter oder sogar notwendig eine Funktion zu definieren, die Arrays beliebiger Größe verarbeiten kann; in solchen Fällen kann das Array auf zwei äquivalenten Wegen übergeben werden, z.B. als Parameter **`double v[]`** oder **`double *v`**, zusammen mit einem weiteren Integer-Parameter, der die Größe des Arrays angibt, z.B. **`int num`**.

```c
#include<stdio.h>

// ********************

// Ausgabe eines Vektors v mit num Elementen.
void print_vector(int num, double v[])
// void print_vector(int num, double *v)   ... ist aequivalent.
{
  printf("| ");

  for(int i = 0; i < num; i++)
    printf("%+6.3lf ", v[i]);

  printf("|\n");
}

// ********************

int main(void)
{
```

```
21.   double x[4] = { 1.0, 2.0, 3.0, 4.0 };
22.   print_vector(4, x);
23.
24.   double y[2] = { +1.5, -2.6 };
25.   print_vector(2, y);
26. }
```

```
| +1.000 +2.000 +3.000 +4.000 |
| +1.500 -2.600 |
```

- Mehrdimensionale Arrays:
  - Bei der Übergabe mehrdimensionaler Arrays an Funktionen darf nur die Größe der ersten Dimension offen gelassen werden.
  - Begründung an Hand des Beispiel-Arrays **`int a[3][3];`**:
    - Wird dieses einer Funktion übergeben, wird lediglich die Adresse des Anfangselements, also der Wert von **`a`**, übergeben.
    - Wie weiter oben erläutert ist der Wert von z.B. **`a[1]`** die Adresse von **`a[1][0]`**; dies muss auch nach der Übergabe, d.h. innerhalb der Funktion an die übergeben wurde, der Fall sein.
    - Da die Elemente eines Arrays hintereinander im Speicher liegen, im betrachteten Beispiel in der Reihenfolge **`a[0][0]`**, **`a[0][1]`**, **`a[0][2]`**, **`a[1][0]`**, **`a[1][1]`**, ..., wird die Größe der zweiten Dimension benötigt, um die Adresse von **`a[1][0]`** und damit den Wert von **`a[1]`** zu ermitteln.

- Ein mehrdimensionales Array kann auf drei äquivalenten Wegen übergeben werden, z.B. als Parameter **`double M[2][2]`**, **`double M[][2]`** oder **`double (*M)[2]`**, wobei die Größe der zweite Dimension jeweils 2 ist.
  (Anmerkung: **`double (*M)[2]`** ist ein Zeiger auf 2-elementige **`double`**-Arrays; **`double *M[2]` dagegen wäre ein 2-elementiges Array von `double`-Zeigern und damit falsch!**)

```c
#include<stdio.h>

// Ausgabe einer Matrix M mit 2 x 2 Elementen.
void print_matrix_2(double M[2][2])
// void print_matrix_2(double M[][2])   ... ist aequivalent.
// void print_matrix_2(double (*M)[2])   ... ist aequivalent.
{
  int i1, i2;

  for(i1 = 0; i1 < 2; i1++)
    {
      printf("| ");

      for(i2 = 0; i2 < 2; i2++)
        printf("%+6.3lf ", M[i1][i2]);

      printf("|\n");
    }
}
```

```
20.
21. int main(void)
22. {
23.   double A[2][2] = { {1.0, 2.0},
24.                      {3.0, 4.0} };
25.
26.   print_matrix_2(A);
27. }
```

```
| +1.000 +2.000 |
| +3.000 +4.000 |
```

- Will man bei der Übergabe mehrdimensionaler Arrays an Funktionen nicht nur die Größe der ersten Dimension offen lassen, verwendet man häufig hinreichend große eindimensionale Arrays und bildet die zwei, drei, ... Indizes des gewünschten mehrdimensionalen Arrays mit einer selbstgeschriebenen **Index-Funktion** auf einen einzigen **Superindex** ab.

```
1. #include<stdio.h>
2.
3. // ********************
4.
5. // Bildet zwei Indizes i1, i2 = 0 ... n-1 auf einen Superindex 0 ... n^2-1 ab.
6. // Z.B. n = 3   dann   (i1,i2) = (0,0)  -->  0
7. //                           = (0,1)  -->  1
8. //                           = (0,2)  -->  2
9. //                           = (1,0)  -->  3
10. //                          = (1,1)  -->  4
```

```
11. //                          = (1,2)  -->  5
12. //                          = (2,0)  -->  6
13. //                          = (2,1)  -->  7
14. //                          = (2,2)  -->  8
15. int index(int i1, int i2, int n)
16. {
17.   return i1*n + i2;
18. }
19.
20. // ********************
21.
22. // Ausgabe einer Matrix M mit n x n Elementen.
23. void print_matrix(int n, double M[])
24. {
25.   int i1, i2;
26.
27.   for(i1 = 0; i1 < n; i1++)
28.     {
29.       printf("| ");
30.
31.       for(i2 = 0; i2 < n; i2++)
32.         printf("%+6.3lf ", M[index(i1, i2, n)]);
33.
34.       printf("|\n");
35.     }
36. }
37.
```

```
38. // ********************
39. 
40. int main(void)
41. {
42.   const int n_A = 2;
43.   double A[n_A*n_A];
44.   A[index(0, 0, n_A)] = 1.0;
45.   A[index(0, 1, n_A)] = 2.0;
46.   A[index(1, 0, n_A)] = 3.0;
47.   A[index(1, 1, n_A)] = 4.0;
48.   print_matrix(n_A, A);
49. 
50.   const int n_B = 3;
51.   double B[n_B*n_B];
52.   B[index(0, 0, n_B)] = 0.1;
53.   B[index(0, 1, n_B)] = 0.2;
54.   B[index(0, 2, n_B)] = 0.3;
55.   B[index(1, 0, n_B)] = -0.4;
56.   B[index(1, 1, n_B)] = -0.5;
57.   B[index(1, 2, n_B)] = -0.6;
58.   B[index(2, 0, n_B)] = 0.7;
59.   B[index(2, 1, n_B)] = 0.8;
60.   B[index(2, 2, n_B)] = 0.9;
61.   print_matrix(n_B, B);
62. }
```

```
| +1.000 +2.000 |
```

```
| +3.000 +4.000 |
| +0.100 +0.200 +0.300 |
| -0.400 -0.500 -0.600 |
| +0.700 +0.800 +0.900 |
```

Anwendung: Eigene Bibliothek für lineare Algebra

- **Ziel:**
  - Ein Satz von Funktionen (eine Bibliothek), der das einfache Rechnen mit vierkomponentigen Vektoren und 4×4-Matrizen ermöglicht, z.B. im Hinblick auf Fragestellungen der speziellen Relativitätstheorie.
- **Vorgehen:**
  - Funktionen erhalten Zeiger auf eindimensionale Arrays (bei Vektoren) und zweidimensionale Arrays (bei Matrizen).
  - Selbsterklärende Funktionsnamen, z.B. **`ve_eq_ma_ti_ve`** steht für "vector equal matrix times vector", ist also eine Funktion, die eine Matrix und einen Vektor multipliziert und das Ergebnis einem anderen Vektor zuweist.
  - Die folgenden Funktionen stellen nur eine kleine Untermenge aller in diesem Zusammenhang sinnvollen bzw. häufig verwendeten Funktionen dar.
  - **Hausaufgabe:** Vervollständige die folgende Bibliothek.

```
1. #include<math.h>
2. #include<stdio.h>
3. #include<stdlib.h>
4.
```

```
 5. // ********************
 6.
 7. // Die Matrix- bzw. Vektorgroesse (durch die Definition der konstanten
 8. // Variable N koennen viele der folgenden Funktionen direkt auch fuer andere N
 9. // verwendet werden).
10. const int N = 4;
11.
12. // ********************
13.
14. // A = 0.
15. void ma_eq_zero(double A[N][N])
16. {
17.   int i1, i2;
18.
19.   for(i1 = 0; i1 < N; i1++)
20.     {
21.       for(i2 = 0; i2 < N; i2++)
22.         A[i1][i2] = 0.0;
23.     }
24. }
25.
26. // ********************
27.
28. // A = 1.
29. void ma_eq_id(double A[N][N])
30. {
31.   int i1;
```

```
32.
33.   ma_eq_zero(A);
34.
35.   for(i1 = 0; i1 < N; i1++)
36.     A[i1][i1] = 1.0;
37. }
38.
39. // ********************
40.
41. // A = Boost in x-Richtung mit beta = v/c.
42. void ma_eq_boost_x(double A[N][N], double beta)
43. {
44.   if(fabs(beta) >= 1.0)
45.     {
46.       printf("Fehler in void ma_eq_boost_x(...\n");
47.       exit(0);
48.     }
49.
50.   ma_eq_zero(A);
51.   double gamma = 1.0 / sqrt(1 - pow(beta, 2.0));
52.   A[0][0] = gamma;
53.   A[0][1] = gamma * beta;
54.   A[1][0] = gamma * beta;
55.   A[1][1] = gamma;
56.   A[2][2] = 1.0;
57.   A[3][3] = 1.0;
58. }
```

```
59.
60. // ********************
61.
62. // A = Rotation um die x-Achse mit Winkel alpha.
63. void ma_eq_rotation_x(double A[N][N], double alpha)
64. {
65.   ma_eq_zero(A);
66.   A[0][0] = 1.0;
67.   A[1][1] = 1.0;
68.   A[2][2] = cos(alpha);
69.   A[2][3] = +sin(alpha);
70.   A[3][2] = -sin(alpha);
71.   A[3][3] = cos(alpha);
72. }
73.
74. // ********************
75.
76. // A = B * C.
77. void ma_eq_ma_ti_ma(double A[N][N], const double B[N][N], const double C[N][N])
78. {
79.   int i1, i2, i3;
80.
81.   for(i1 = 0; i1 < N; i1++)
82.     {
83.       for(i2 = 0; i2 < N; i2++)
84.         {
85.           A[i1][i2] = 0.0;
```

```
86.
87.           for(i3 = 0; i3 < N; i3++)
88.             A[i1][i2] += B[i1][i3] * C[i3][i2];
89.         }
90.     }
91. }
92.
93. // ********************
94.
95. // x = A * y.
96. void ve_eq_ma_ti_ve(double x[N], const double A[N][N], const double y[N])
97. {
98.   int i1, i2;
99.
100.   for(i1 = 0; i1 < N; i1++)
101.     {
102.       x[i1] = 0.0;
103.
104.       for(i2 = 0; i2 < N; i2++)
105.         x[i1] += A[i1][i2] * y[i2];
106.     }
107. }
108.
109. // ********************
110.
111. // d = x^\mu g_{\mu \nu} y^\nu   (Skalarprodukt, Minkowski-Metrik).
112. void sc_eq_ve_g_mu_nu_ve(double *d, const double x[N], const double y[N])
```

```
113. {
114.   *d = x[0]*y[0] - x[1]*y[1] - x[2]*y[2] - x[3]*y[3];
115. }
116. 
117. // ********************
118. 
119. // Ausgabe der Matrix A.
120. void ma_print(const double A[N][N])
121. {
122.   int i1, i2;
123. 
124.   for(i1 = 0; i1 < N; i1++)
125.     {
126.       printf("| ");
127. 
128.       for(i2 = 0; i2 < N; i2++)
129.         printf("%+6.3lf ", A[i1][i2]);
130. 
131.       printf("|\n");
132.     }
133. }
134. 
135. // ********************
136. 
137. // Ausgabe des Vektors x.
138. void ve_print(const double x[N])
139. {
```

```
140.   int i1;
141.
142.   printf("| ");
143.
144.   for(i1 = 0; i1 < N; i1++)
145.     printf("%+6.3lf ", x[i1]);
146.
147.   printf("|\n");
148. }
149.
150. // ********************
151.
152. int main(void)
153. {
154.   double A[N][N];
155.
156.   ma_eq_zero(A);
157.   ma_print(A);
158.
159.   printf("********************\n");
160.
161.   ma_eq_id(A);
162.   ma_print(A);
163.
164.   printf("********************\n");
165.
166.   double boost[N][N];
```

```
167.
168.   ma_eq_boost_x(boost, 0.5);  // Boost mit beta = 0.5 in x-Richtung.
169.   ma_print(boost);
170.
171.   printf("********************\n");
172.
173.   double rotation[N][N];
174.
175.   ma_eq_rotation_x(rotation, 30.0 * M_PI / 180.0);  // 30-Grad-Rotation um die x-Achse.
176.   ma_print(rotation);
177.
178.   printf("********************\n");
179.
180.   double Lambda[N][N];
181.
182.   ma_eq_ma_ti_ma(Lambda, rotation, boost);  // Lorentz-Transformation Lambda = rotation * boost.
183.   ma_print(Lambda);
184.
185.   printf("********************\n");
186.
187.   double x[4] = { 1.0, 0.0, 3.0, 0.0 };  // Ein Vierervektor.
188.   ve_print(x);
189.
190.   double x_scalar_x;
191.   sc_eq_ve_g_mu_nu_ve(&x_scalar_x, x, x);
192.   printf("x_scalar_x = %f\n", x_scalar_x);
193.
```

```
194.   printf("********************\n");
195.   double y[4];
196.   ve_eq_ma_ti_ve(y, Lambda, x);  // y = Lambda * x   (d.h. der LT Vierervektor x).
197.   ve_print(y);
198.
199.   double y_scalar_y;
200.   sc_eq_ve_g_mu_nu_ve(&y_scalar_y, y, y);
201.   printf("y_scalar_y = %f\n", y_scalar_y);
202. }
```

```
| +0.000 +0.000 +0.000 +0.000 |
| +0.000 +0.000 +0.000 +0.000 |
| +0.000 +0.000 +0.000 +0.000 |
| +0.000 +0.000 +0.000 +0.000 |
**********
| +1.000 +0.000 +0.000 +0.000 |
| +0.000 +1.000 +0.000 +0.000 |
| +0.000 +0.000 +1.000 +0.000 |
| +0.000 +0.000 +0.000 +1.000 |
**********
| +1.155 +0.577 +0.000 +0.000 |
| +0.577 +1.155 +0.000 +0.000 |
| +0.000 +0.000 +1.000 +0.000 |
| +0.000 +0.000 +0.000 +1.000 |
**********
| +1.000 +0.000 +0.000 +0.000 |
| +0.000 +1.000 +0.000 +0.000 |
| +0.000 +0.000 +0.866 +0.500 |
| +0.000 +0.000 -0.500 +0.866 |
**********
| +1.155 +0.577 +0.000 +0.000 |
| +0.577 +1.155 +0.000 +0.000 |
| +0.000 +0.000 +0.866 +0.500 |
| +0.000 +0.000 -0.500 +0.866 |
**********
| +1.000 +0.000 +3.000 +0.000 |
x_scalar_x = -8.000000
**********
```

```
| +1.155 +0.577 +2.598 -1.500 |
y_scalar_y = -8.000000
```

Strings

- Ein **char**-Array dient dem Speichern von Zeichenketten; man bezeichnet ein solches Array auch als **String**.
- Das Ende eines Strings wird durch das Zeichen **'\0'** markiert; seine Länge im Speicher ist damit um ein Zeichen größer, als die Anzahl der "sichtbaren Zeichen" (Buchstaben, Zahlen, SPACE, etc.).
- **String-Konstanten** werden in der Form **"*string*"** geschrieben.
- Strings und String-Konstanten können mit **printf** und **%s** im Formatstring ausgegeben werden.

```
1.  #include<stdio.h>
2.
3.  int main(void)
4.  {
5.    // Definition eines Strings, sehr umstaendliche Initialisierung mit der
6.    // Zeichenkette "abc".
7.    char string1[4];
8.    string1[0] = 'a';
9.    string1[1] = 'b';
10.   string1[2] = 'c';
11.   string1[3] = '\0';
```

```
12.
13.   // string2 ist aequivalent zu string1, die Initialisierung ist offensichtlich
14.   // komfortabler.
15.   char string2[4] = "abc";
16.
17.   // string3 enthaelt ebenfalls die Zeichenkette "abc"; es wurde aber
18.   // Speicherplatz fuer 1000 Zeichen reserviert; string3 koennte also spaeter
19.   // veraendert werden und dann eine deutlich laengere Zeichenkette aufnehmen.
20.   char string3[1000] = "abc";
21.
22.   // Die Arraygroesse kann bei der Initialisierung auch weggelassen werden;
23.   // dann wird automatisch die minimale Laenge gewaehlt, also im vorliegenden
24.   // Fall Laenge 4.
25.   char string4[] = "abc";
26.
27.   // Ausgabe von strings mit %s im Formatstring.
28.   printf("%s %s %s %s %s\n", string1, string2, string3, string4, "abc");
29. }
```

```
abc abc abc abc abc
```

- Einlesen von Zeichenketten in Strings mit **`scanf`** und ebenfalls **`%s`** im Formatstring; SPACE separiert dabei Zeichenketten.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
```

```
5.   char string[1000];
6.   printf("Zeichenkette eingeben: ");
7.   scanf("%s", string);
8.
9.   printf("\"%s\" in string gespeichert.\n", string);
10. }
```

```
Zeichenkette eingeben: Sheldon
"Sheldon" in string gespeichert.
```

```
Zeichenkette eingeben: Sheldon Cooper
"Sheldon" in string gespeichert.
```

```
1.  #include<stdio.h>
2.
3.  int main(void)
4.  {
5.    char string1[1000], string2[1000];
6.    printf("Zeichenkette eingeben: ");
7.    scanf("%s %s", string1, string2);
8.
9.    printf("\"%s\" in string1 gespeichert, \"%s\" in string2 gespeichert.\n", string1, string2);
10. }
```

```
Zeichenkette eingeben: Sheldon Cooper
"Sheldon" in string1 gespeichert, "Cooper" in string2 gespeichert.
```

- Eine ganze Zeile (die beliebig viele SPACE enthalten darf) kann z.B. mit den in **stdio.h** enthaltenen Funktionen **gets** und **fgets** eingelesen werden.

```
mwagner@laptop-tigger:~$ man gets
```

```
GETS(3)                   Linux Programmer's Manual                  GETS(3)

NAME
       fgetc, fgets, getc, getchar, gets, ungetc - input of characters and
       strings

SYNOPSIS
       #include <stdio.h>
...
       char *fgets(char *s, int size, FILE *stream);
...
       char *gets(char *s);
...

DESCRIPTION
...
       gets() reads a line from stdin into the buffer pointed to by s until
       either a terminating newline or EOF, which it replaces with a null byte
       ('\0'). No check for buffer overrun is performed (see BUGS below).

       fgets() reads in at most one less than size characters from stream and
       stores them into the buffer pointed to by s. Reading stops after an
       EOF or a newline. If a newline is read, it is stored into the buffer.
       A terminating null byte ('\0') is stored after the last character in
       the buffer.
...

RETURN VALUE
...
       gets() and fgets() return s on success, and NULL on error or when end
       of file occurs while no characters have been read.
...

BUGS
       Never use gets(). Because it is impossible to tell without knowing the
       data in advance how many characters gets() will read, and because
       gets() will continue to store characters past the end of the buffer, it
       is extremely dangerous to use. It has been used to break computer
       security. Use fgets() instead.
```

```c
#include<stdio.h>

int main(void)
{
  char string1[10];
  gets(string1);  // Liest beliebig viele Zeichen ein --> kann gefaehrlich sein.
  printf("Zeile 1: %s\n", string1);  // gets ersetzt '\n' durch '\0'.

  char string2[10];
  fgets(string2, 10, stdin);  // Liest maximal 9 Zeichen ein --> sicher.
  printf("Zeile 2: %s", string2);  // fgets speichert '\n', haengt dann '\0' an.
}
```

```
abc def
Zeile 1: abc def
uvw xyz
Zeile 2: uvw xyz
```

```
abc def abc def
Zeile 1: abc def abc def
abc def abc def
Zeile 2: abc def amwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$
```

- Mit
  `int sscanf(const char *str, const char *format, ...);`
  und
  `int sprintf(char *str, const char *format, ...);`
  (beide in **`stdio.h`** enthalten) kann aus Strings gelesen und in Strings geschrieben werden (sehr ähnlich zu **`scanf`** und **`printf`**).

- Mit
  `int atoi(const char *nptr);`
  und
  `double atof(const char *nptr);`
  (beide in **`stdlib.h`** enthalten) können Strings in **`int`**- beziehungsweise **`double`**-Werte umgewandelt werden.

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

int main(void)
{
  char string[1000], string1[1000], string2[1000], string3[1000];
  gets(string);

  sscanf(string, "%s %s %s", string1, string2, string3);
  printf("1. Wort: %s\n", string1);
  printf("2. Wort: %s\n", string2);
  printf("3. Wort: %s\n", string3);

  // 2. Wort in int umwandeln.
  int i = atoi(string2);
  printf("i = %d\n", i);

  // 3. Wort in double umwandeln.
  double d = atof(string3);
```

```
20.   printf("d = %f\n", d);
21. }
```

```
abcdef 123 0.456
1. Wort: abcdef
2. Wort: 123
3. Wort: 0.456
i = 123
d = 0.456000
```

- Weitere hilfreiche Funktionen zur Bearbeitung von Strings (in **`string.h`** enthalten):
  - **`size_t strlen(const char *s);`**
    Bestimmt die Länge des Strings **`s`**.
  - **`char *strcpy(char *dest, const char *src);`**
    Kopiert den String **`src`** in den String **`dest`**.
  - **`char *strcat(char *dest, const char *src);`**
    Hängt den String **`src`** ans Ende des Strings **`dest`** an.
  - **`int strcmp(const char *s1, const char *s2);`**
    Vergleicht die zwei Strings **`s1`** und **`s2`** alphabetisch (liefert "0" bei Gleichheit).

```
1. #include<stdio.h>
2. #include<string.h>
3.
4. int main(void)
5. {
6.   char string1[1000] = "Sheldon";
7.   char string2[1000] = "Cooper";
```

```
 8.
 9.   printf("strlen(\"%s\") = %zd\n", string1, strlen(string1));
10.
11.   printf("********************\n");
12.
13.   printf("string1 = %s\n", string1);
14.   strcat(string1, " ");
15.   strcat(string1, string2);
16.   printf("string1 = %s\n", string1);
17.
18.   printf("********************\n");
19.
20.   printf("string1 = %s\n", string1);
21.   printf("string2 = %s\n", string2);
22.   strcpy(string1, string2);
23.   printf("string1 = %s\n", string1);
24.   printf("string2 = %s\n", string2);
25.
26.   printf("********************\n");
27.
28.   printf("strcmp(\"%s\", \"AAA\") = %d\n", string1, strcmp(string1, "AAA"));
29.   printf("strcmp(\"%s\", \"Cooper\") = %d\n", string1, strcmp(string1, "Cooper"));
30.   printf("strcmp(\"%s\", \"EEE\") = %d\n", string1, strcmp(string1, "FFF"));
31. }
```

```
strlen("Sheldon") = 7
********************
string1 = Sheldon
```

```
string1 = Sheldon Cooper
**********
string1 = Sheldon Cooper
string2 = Cooper
string1 = Cooper
string2 = Cooper
**********
strcmp("Cooper", "AAA") = 2
strcmp("Cooper", "Cooper") = 0
strcmp("Cooper", "EEE") = -3
```

Anwendung: Auswertung arithmetischer Ausdrücke in Textform

- **Ziel:**
  - Ein Programm, das einen in einem String gespeicherten arithmetischen Ausdruck auswertet.
  - Der String darf Ziffern, "+", "-", "(", ")" und `SPACE` enthalten (es sind also nur ganze Zahlen erlaubt).
  - "+" und "-" sollen linksassoziativ ausgewertet werden, also "10-2-3" soll 5 ergeben und nicht 11.
- **Algorithmus:**
  1. Entferne alle `SPACE` aus dem String.
  2. Suche von rechts kommend den ersten Operator "+" oder "-" im String, der nicht innerhalb von Klammern "(" und ")" steht.
  3. Falls ein solcher Operator gefunden wird, rufe den Algorithmus beginnend mit Schritt 2 **rekursiv** mit dem Teilstring links des Operators und mit dem Teilstring rechts des Operators auf und addiere/subtrahiere die beiden Ergebnisse; liefere das Resultat zurück und beende damit den Algorithmus.

4. Falls kein solcher Operator gefunden wird, prüfe, ob ganz links im String "(" und ganz rechts im String ")" steht; falls ja, entferne die beiden Klammern und rufe damit den Algorithmus beginnend mit Schritt 2 **rekursiv** auf; liefere das Ergebnis zurück und beende damit den Algorithmus.
5. Prüfe, ob der String ausschließlich Ziffern enthält; falls ja, wandle die Ziffernfolge in einen **`int`**-Wert um; liefere das Ergebnis zurück und beende damit den Algorithmus.
6. Die Syntax des im String gespeicherten arithmetischen Ausdrucks ist nicht korrekt; beende das Programm mit einer Fehlermeldung.

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

// ********************

// Entfernt alle Leerzeichen aus dem string str.
void remove_spaces(char str[])
{
  int i1, i2;

  for(i1 = 0; i1 < strlen(str); i1++)
    {
      if(str[i1] == ' ')
        {
          // Die folgende Schleife entfernt das Leerzeichen bei Position i1.
```

```
17.           for(i2 = i1; i2 < strlen(str); i2++)
18.             str[i2] = str[i2+1];
19.
20.           i1--;  // Schleifenzaehler um 1 zuruecksetzen, sonst wird ein Zeichen uebersprungen.
21.         }
22.     }
23. }
24.
25. // ********************
26.
27. // Wertet den in str gespeicherten arithmetischen Ausdruck rekursiv aus;
28. // erlaubt sind ganze Zahlen, Leerzeichen, binaeres '+' und '-' und Klammern
29. // '(' und ')'.
30. int eval(char str[])
31. {
32.   int pos;
33.   int num_brackets_open = 0;
34.
35.   for(pos = strlen(str)-1; pos >= 0; pos--)
36.     // Sucht nach dem am weitesten rechts (wegen Linksassoziativitaet)
37.     // stehenden Operator '+' oder '-', der sich nicht innerhalb von Klammern
38.     // befindet; Schleife vom letzten bis zum ersten Zeichen.
39.     {
40.       if(str[pos] == '(')
41.         num_brackets_open--;
42.
43.       if(str[pos] == ')')
```

```
44.         num_brackets_open++;
45.
46.       if(num_brackets_open != 0)
47.         // Wenn innerhalb von '(' und ')', nicht nach Operatoren '+' oder '-'
48.         // suchen.
49.         continue;
50.
51.       if(str[pos] == '+')
52.         // Operator '+' gefunden.
53.         {
54.           str[pos] = '\0';
55.           return eval(str) + eval(&str[pos+1]);  // Rekursiver Aufruf von eval.
56.         }
57.
58.       if(str[pos] == '-')
59.         // Operator '-' gefunden.
60.         {
61.           str[pos] = '\0';
62.           return eval(str) - eval(&str[pos+1]);  // Rekursiver Aufruf von eval.
63.         }
64.     }
65.
66.   // str enthaelt entweder "(...)" oder eine Zahl ...
67.
68.   if(str[0] == '(' && str[strlen(str)-1] == ')')
69.     {
70.       for(pos = 1; pos < strlen(str)-1; pos++)
```

```
71.         str[pos-1] = str[pos];
72.
73.       str[strlen(str)-2] = '\0';
74.
75.       return eval(str);  // Rekursiver Aufruf von eval.
76.     }
77.
78.   // str enthaelt eine Zahl ...
79.
80.   for(pos = 0; pos < strlen(str); pos++)
81.     {
82.       if(str[pos] < '0' || str[pos] > '9')
83.         // Keine Ziffer.
84.         {
85.           printf("Fehlerhafter arithmetischer Ausdruck!\n");
86.           exit(0);
87.         }
88.     }
89.
90.   // Arithmetischer Ausdruck ist eine Zahl.
91.   return atoi(str);
92. }
93.
94. // ********************
95.
96. int main(void)
97. {
```

```
 98.   char string1[1000];
 99.   printf("Arithmetischen Ausdruck eingeben: ");
100.   gets(string1);
101.
102.   remove_spaces(string1);
103.   // printf("string1 = %s\n", string1);
104.
105.   printf("Der Wert dieses Ausdrucks ist %d.\n", eval(string1));
106. }
```

```
Arithmetischen Ausdruck eingeben: 55
Der Wert dieses Ausdrucks ist 55.
```

```
Arithmetischen Ausdruck eingeben: 3 + 44
Der Wert dieses Ausdrucks ist 47.
```

```
Arithmetischen Ausdruck eingeben: (  (7   -1) + 5    -((3)) )
Der Wert dieses Ausdrucks ist 8.
```

```
Arithmetischen Ausdruck eingeben: 10 - 2 - 3
Der Wert dieses Ausdrucks ist 5.
```

```
Arithmetischen Ausdruck eingeben: 10 - (2 - 3)
Der Wert dieses Ausdrucks ist 11.
```

```
Arithmetischen Ausdruck eingeben: ((3) + 4))
Fehlerhafter arithmetischer Ausdruck!
```

Argumente aus der Kommandozeile

- Sollen beim Programmaufruf Argumente aus der Kommandozeile an das Programm übergeben werden, ist folgende Version von **`main`** zu verwenden:
  `int main(int argc, char *argv[]) {...}`.
  - **`argc`** enthält die Anzahl der Argumente (der in der Kommandozeile zum Aufruf verwendete Programmname zählt selbst als Argument, d.h. **`argc`** ist "1" oder größer).
  - **`argv`** ist ein Array von Zeigern auf **`char`** und damit ein Array von Strings; die Strings entsprechen den Argumenten aus der Kommandozeile.

```
1.  #include<stdio.h>
2.  #include<stdlib.h>
3.
4.  int main(int argc, char *argv[])
5.  {
6.    if(argc != 3)
7.      {
8.        printf("%s <summand1> <summand2>\n", argv[0]);
9.        exit(0);
10.     }
11.
12.   printf("Summand 1 = %s\n", argv[1]);
```

```
13.   printf("Summand 2 = %s\n", argv[2]);
14.
15.   double d1 = atof(argv[1]);
16.   double d2 = atof(argv[2]);
17.
18.   printf("Summe = %f\n", d1+d2);
19. }
```

```
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ ./add
./add <summand1> <summand2>
```

```
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ ./add 3.0 2.5
Summand 1 = 3.0
Summand 2 = 2.5
Summe = 5.500000
```

Zeiger auf Funktionen

- Neben Zeigern auf Variablen existieren auch Zeiger auf Funktionen; damit können z.B. Funktionen selbst als Parameter an andere Funktionen übergeben werden.
- Ein Zeiger auf eine Funktion
  type function_name`(`*type1 para1*`, `*type2 para2*`, ...){...}`
  ist vom Typ
  type `(*)(`*type1*`, `*type2*`, ...)`;
  die Klammern um * sind notwendig, da die Klammern um die Argumente *type1*`, `*type2*`, ...` (d.h. der Operator für Funktionsaufrufe) höheren Vorrang besitzen als *.
- Die Adresse einer Funktion erhält man über den Funktionsnamen (ohne `(...)`; analog zu Arrays).

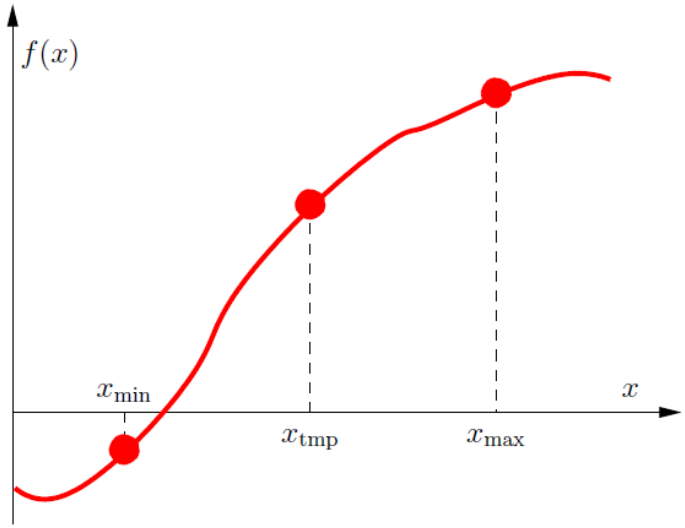
```
1. #include<stdio.h>
2.
3. double f(double x)
4. {
5.   return x + 1.0;
6. }
7.
8. double g(double x)
9. {
```

```
10.   return x*x;
11. }
12. 
13. // Die Klammersetzung bei
14. //   double (*func)(double)
15. // ist essentiell; es handelt sich um einen Zeiger mit Namen func auf eine
16. // Funktion mit double-Rueckgabewert und einem double-Parameter;
17. //   double *func(double)
18. // wuerde dagegen eine Funktion mit Namen func mit double *-Rueckgabewert und
19. // einem double-Parameter beschreiben, da der Operator "()" fuer
20. // Funktionsaufrufe hoeheren Vorrang als der Operator "*" besitzt.
21. void apply_function_and_print_result(double (*func)(double), double x)
22. {
23.   double func_x = (*func)(x);
24.   printf("x = %f   func(x) = %f\n", x, func_x);
25. }
26. 
27. int main(int argc, char *argv[])
28. {
29.   apply_function_and_print_result(f, 3.0);
30.   apply_function_and_print_result(g, 3.0);
31. }
```

```
x = 3.000000   func(x) = 4.000000
x = 3.000000   func(x) = 9.000000
```

- **Beispiel:** Elegante Version der Nullstellensuche mit Bijektion ...

- Startpunkt: $x_{\min}$ und $x_{\max}$ mit $f(x_{\min})f(x_{\max}) \leq 0$ (damit ist mindestens eine Nullstelle im Intervall $[x_{\min}, x_{\max}]$ garantiert).
- **Algorithmus**:
  1. Falls $x_{\max} - x_{\min} < \epsilon$, beende den Algorithmus; die Nullstelle ist $x_0 = (x_{\min} + x_{\max})/2$ mit einer numerischen Genauigkeit von mindestens $\epsilon/2$.
  2. Teile das Intervall in der Mitte, d.h. bei $x_{\text{tmp}} = (x_{\min} + x_{\max})/2$.
  3. Falls eine Nullstelle im linken Intervall garantiert ist (falls $f(x_{\min})f(x_{\text{tmp}}) \leq 0$), ersetze $x_{\max} = x_{\text{tmp}}$; gehe zu 1.
  4. Eine Nullstelle ist im rechten Intervall garantiert; ersetze $x_{\min} = x_{\text{tmp}}$; gehe zu 1.




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

// ********************

// Nullstellensuche mit Bisektion.
//   f: Funktion, deren Nullstelle bestimmt wird.
//   x_min, x_max: Intervall, in dem eine Nullstelle garantiert sein muss, d.h.
//     f(x_min) * f(x_max) <= 0.
//   eps: Numerische Genauigkeit der Nullstellensuche.
double bisection(double (*f)(double), double x_min, double x_max, double eps)
{
```

```
14.  // Fehlerabfrage der Input-Daten.
15.
16.  if(eps <= 0.0)
17.    {
18.      printf("Fehler in double bisection(...: eps <= 0.0.\n");
19.      exit(0);
20.    }
21.
22.  if(x_max < x_min)
23.    {
24.      printf("Fehler in double bisection(...: x_max < x_min.\n");
25.      exit(0);
26.    }
27.
28.  double f_min = (*f)(x_min);
29.  double f_max = (*f)(x_max);
30.
31.  if((f_min <= 0.0 && f_max <= 0.0) ||
32.     (f_min >= 0.0 && f_max >= 0.0))
33.    {
34.      printf("Fehler in double bisection(...: Nullstelle im Intervall [x_min,x_max] ist nicht garantiert.\n");
35.      exit(0);
36.    }
37.
38.  // **********
39.
40.  printf("x_min = %+f f(x_min) = %+f x_max = %+f f(x_max) = %+f\n", x_min, f_min, x_max, f_max);
```

```
41.
42.   // Nullstellensuche mit Bisektion.
43.
44.   while(x_max-x_min > eps)
45.     {
46.       double x_tmp = 0.5 * (x_min+x_max);
47.       double f_tmp = (*f)(x_tmp);
48.
49.       if((f_min <= 0.0 && f_tmp >= 0.0) ||
50.          (f_min >= 0.0 && f_tmp <= 0.0))
51.         // Nullstelle im linken Teilintervall [x_min,x_tmp].
52.         {
53.           x_max = x_tmp;
54.           f_max = f_tmp;
55.         }
56.       else
57.         // Nullstelle im rechten Teilintervall [x_tmp,x_max].
58.         {
59.           x_min = x_tmp;
60.           f_min = f_tmp;
61.         }
62.
63.       printf("x_min = %+f f(x_min) = %+f x_max = %+f f(x_max) = %+f\n", x_min, f_min, x_max, f_max);
64.     }
65.
66.   return 0.5 * (x_min+x_max);  // Die gesuchte Nullstelle.
67. }
```

```
68.
69. // ********************
70.
71. // Eine Funktion, von der eine Nullstelle gesucht wird (sin(x) + 0.01*x).
72. double f(double x)
73. {
74. return sin(x) + 0.01*x;
75. }
76.
77. // ********************
78.
79. int main(void)
80. {
81.   double x_0;
82.
83.   x_0 = bisection(sin, 3.0, 3.5, 1.0e-6);
84.   printf("Nullstelle von sin(x), Startintervall [3.0,3.5]: sin(%+f) = %+e\n", x_0, sin(x_0));
85.
86.   printf("\n");
87.
88.   x_0 = bisection(f, 3.0, 3.5, 1.0e-6);
89.   printf("Nullstelle von f(x) = sin(x) + 0.01*x, Startintervall [3.0,3.5]: f(%+f) = %+e\n", x_0, f(x_0));
90. }
```

```
x_min = +3.000000 f(x_min) = +0.141120 x_max = +3.500000 f(x_max) = -0.350783
x_min = +3.000000 f(x_min) = +0.141120 x_max = +3.250000 f(x_max) = -0.108195
x_min = +3.125000 f(x_min) = +0.016592 x_max = +3.250000 f(x_max) = -0.108195
x_min = +3.125000 f(x_min) = +0.016592 x_max = +3.187500 f(x_max) = -0.045891
```

```
x_min = +3.125000 f(x_min) = +0.016592 x_max = +3.156250 f(x_max) = -0.014657
x_min = +3.140625 f(x_min) = +0.000968 x_max = +3.156250 f(x_max) = -0.014657
x_min = +3.140625 f(x_min) = +0.000968 x_max = +3.148438 f(x_max) = -0.006845
x_min = +3.140625 f(x_min) = +0.000968 x_max = +3.144531 f(x_max) = -0.002939
x_min = +3.140625 f(x_min) = +0.000968 x_max = +3.142578 f(x_max) = -0.000985
x_min = +3.140625 f(x_min) = +0.000968 x_max = +3.141602 f(x_max) = -0.000009
x_min = +3.141113 f(x_min) = +0.000479 x_max = +3.141602 f(x_max) = -0.000009
x_min = +3.141357 f(x_min) = +0.000235 x_max = +3.141602 f(x_max) = -0.000009
x_min = +3.141479 f(x_min) = +0.000113 x_max = +3.141602 f(x_max) = -0.000009
x_min = +3.141541 f(x_min) = +0.000052 x_max = +3.141602 f(x_max) = -0.000009
x_min = +3.141571 f(x_min) = +0.000022 x_max = +3.141602 f(x_max) = -0.000009
x_min = +3.141586 f(x_min) = +0.000006 x_max = +3.141602 f(x_max) = -0.000009
x_min = +3.141586 f(x_min) = +0.000006 x_max = +3.141594 f(x_max) = -0.000001
x_min = +3.141590 f(x_min) = +0.000003 x_max = +3.141594 f(x_max) = -0.000001
x_min = +3.141592 f(x_min) = +0.000001 x_max = +3.141594 f(x_max) = -0.000001
x_min = +3.141592 f(x_min) = +0.000001 x_max = +3.141593 f(x_max) = -0.000000
Nullstelle von sin(x), Startintervall [3.0,3.5]: sin(+3.141593) = +1.509958e-07

x_min = +3.000000 f(x_min) = +0.171120 x_max = +3.500000 f(x_max) = -0.315783
x_min = +3.000000 f(x_min) = +0.171120 x_max = +3.250000 f(x_max) = -0.075695
x_min = +3.125000 f(x_min) = +0.047842 x_max = +3.250000 f(x_max) = -0.075695
x_min = +3.125000 f(x_min) = +0.047842 x_max = +3.187500 f(x_max) = -0.014016
x_min = +3.156250 f(x_min) = +0.016906 x_max = +3.187500 f(x_max) = -0.014016
x_min = +3.171875 f(x_min) = +0.001441 x_max = +3.187500 f(x_max) = -0.014016
x_min = +3.171875 f(x_min) = +0.001441 x_max = +3.179688 f(x_max) = -0.006289
x_min = +3.171875 f(x_min) = +0.001441 x_max = +3.175781 f(x_max) = -0.002424
x_min = +3.171875 f(x_min) = +0.001441 x_max = +3.173828 f(x_max) = -0.000492
x_min = +3.172852 f(x_min) = +0.000475 x_max = +3.173828 f(x_max) = -0.000492
x_min = +3.172852 f(x_min) = +0.000475 x_max = +3.173340 f(x_max) = -0.000008
x_min = +3.173096 f(x_min) = +0.000233 x_max = +3.173340 f(x_max) = -0.000008
x_min = +3.173218 f(x_min) = +0.000112 x_max = +3.173340 f(x_max) = -0.000008
x_min = +3.173279 f(x_min) = +0.000052 x_max = +3.173340 f(x_max) = -0.000008
x_min = +3.173309 f(x_min) = +0.000022 x_max = +3.173340 f(x_max) = -0.000008
x_min = +3.173325 f(x_min) = +0.000007 x_max = +3.173340 f(x_max) = -0.000008
x_min = +3.173325 f(x_min) = +0.000007 x_max = +3.173332 f(x_max) = -0.000001
x_min = +3.173328 f(x_min) = +0.000003 x_max = +3.173332 f(x_max) = -0.000001
x_min = +3.173330 f(x_min) = +0.000001 x_max = +3.173332 f(x_max) = -0.000001
x_min = +3.173331 f(x_min) = +0.000000 x_max = +3.173332 f(x_max) = -0.000001
Nullstelle von f(x) = sin(x) + 0.01*x, Startintervall [3.0,3.5]: f(+3.173332) = -4.379912e-07
```

Komplizierte Definitionen

- Definitionen von Variablen und Funktionen können teilweise kompliziert sein, insbesondere wenn Zeiger beteiligt sind.
- Beispiele:
  - `char **a`
    Ein Zeiger auf einen Zeiger auf `char`.
  - `int *b[10]`
    Ein Array von 10 Zeigern auf `int`.
  - `int (*b)[10]`
    Ein Zeiger auf ein `int`-Array mit 10 Elementen.
  - `double *c()`
    Eine Funktion mit `double *`-Rückgabewert.
  - `double (*c)()`
    Ein Zeiger auf eine Funktion mit `double`-Rückgabewert.
  - `char (*(*d())[])()`
    Eine Funktion die einen Zeiger auf ein Array von Zeigern auf Funktionen mit `char`-Rückgabewert zurückgibt.

- `char (*(*e[5])())[10]`
  Ein 5-elementiges Array von Funktionszeigern; die Funktionen, auf die gezeigt wird, geben jeweils einen Zeiger auf ein 10-elementiges `char`-Array zurück.

- **In eignem Programmcode sollten derartig komplizierte Definitionen, wie in den letzten beiden Zeilen, falls möglich, vermieden werden.**

Dynamische Speicherverwaltung

- Häufig ist die benötigte Größe eines Arrays erst zur Laufzeit eines Programms bekannt, z.B.
  - der Experimentator gibt an, wie viele Messwerte eingelesen und gespeichert werden sollen,
  - der Theoretiker legt fest, mit welcher Matrixgröße er rechnen will,
  - ...
- In solchen Fällen empfiehlt sich die Verwendung **dynamischer Speicherverwaltung**:
  - Während der Laufzeit des Programms wird an geeigneter Stelle ein hinreichend großer Speicherbereich angefordert (z.B. für ein Array der gewünschten Größe).
  - Sobald der angeforderte Speicherbereich nicht mehr benötigt wird, wird er, ebenfalls zur Laufzeit des Programms, wieder freigegeben und steht damit für eventuelle weitere Speicheranforderungen zur Verfügung.
- **Speicherbereich anfordern** mit
  `void *malloc(size_t size);`
  (in `stdlib.h` enthalten); `size`: Anzahl der benötigten Bytes; Rückgabewert: die Adresse des reservierten Speicherbereichs, bzw. `NULL` bei einem Fehler, z.B. nicht genügend

Speicherplatz verfügbar.

- **Speicherbereich freigeben** mit
  `void free(void *ptr);`
  (in **`stdlib.h`** enthalten); **`ptr`**: Zeiger auf den freizugebenden Speicherbereich.
- **Angeforderten Speicherbereich vergrößern/verkleinern** mit
  `void *realloc(void *ptr, size_t size);`
  (in **`stdlib.h`** enthalten); **`size`**: Anzahl der benötigten Bytes; **`ptr`**: Zeiger auf den zu vergrößernden/verkleinernden Speicherbereich; Rückgabewert: die Adresse des vergrößerten/verkleinerten Speicherbereichs, bzw. **`NULL`** bei einem Fehler.
- **Beispiel:** dynamisches Anfordern von Speicherplatz für eine **`double`**-Variable (ein rein akademisches Beispiel) ...

```
1. #include<stdio.h>
2. #include<stdlib.h>
3.
4. int main(void)
5. {
6.   double *pd = NULL;
7.
8.   // Speicherplatz fuer eine "double-Variable" anfordern.
9.   pd = (double *)malloc(sizeof(double));
10.
11.  // Pruefe, ob Speicherplatz zur Verfuegung gestellt werden konnte.
12.  if(pd == NULL)
```

```
13.     {
14.       printf("Fehler: malloc fehlgeschlagen!\n");
15.       exit(0);
16.     }
17.
18.   *pd = 1.23;
19.   printf("*pd = %f\n", *pd);
20.
21.   // Angeforderten Speicherplatz freigeben.
22.   free(pd);
23.   pd = NULL;  // Zeigt an, dass pd keine brauchbare Adresse mehr zugeordnet ist.
24.
25.   // ********************
26.
27.   // Das Gleiche nochmal etwas kompakter ...
28.
29.   if((pd = (double *)malloc(sizeof(double))) == NULL)
30.     {
31.       printf("Fehler: malloc fehlgeschlagen!\n");
32.       exit(0);
33.     }
34.
35.   // ...
36.
37.   free(pd);
38.   pd = NULL;
39. }
```

```
*pd = 1.230000
```

- **Beispiel:** dynamisches Anfordern von Speicherplatz für ein **double**-Array, anschließendes Vergrößern dieses Arrays ...

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

int main(void)
{
  int i1;
  char string1[1000];

  int n;  // Anzahl der zu speichernden double-Werte.

  // ********************

  // Daten einlesen, dafuer dynamisch Speicher anfordern.

  printf("Wie viele Messwerte willst Du eingeben? ");
  scanf("%d", &n);

  if(n <= 0)
    {
      printf("Fehler: n muss groesser als 0 sein.\n");
      exit(0);
```

```
23.     }
24.
25.   double *data;
26.
27.   if((data = (double *)malloc(n*sizeof(double))) == NULL)
28.     {
29.       printf("Fehler: malloc fehlgeschlagen!\n");
30.       exit(0);
31.     }
32.
33.   for(i1 = 0; i1 < n; i1++)
34.     {
35.       printf("Messwert %d: ", i1);
36.       scanf("%lf", data+i1);  // &data[i1] waere aequivalent zu data+i1
37.     }
38.
39.   // ********************
40.
41.   for(i1 = 0; i1 < n; i1++)
42.     printf("data[%d] = %f\n", i1, data[i1]);
43.
44.   // ********************
45.
46.   while(1)
47.     {
48.       printf("Willst Du weitere Messwerte eingeben (yes/no)? ");
49.       scanf("%s", string1);
```

```
50.
51.       if(strcmp(string1, "no") == 0)
52.         break;
53.
54.       if(strcmp(string1, "yes") != 0)
55.         continue;
56.
57.       // **********
58.
59.       // Weitere Daten einlesen, dafuer dynamisch Speicher erweitern.
60.
61.       int delta_n;
62.       printf("Wie viele Messwerte willst Du eingeben? ");
63.       scanf("%d", &delta_n);
64.
65.       if(delta_n <= 0)
66.         {
67.           printf("Fehler: delta_n muss groesser als 0 sein.\n");
68.           exit(0);
69.         }
70.
71.       if((data = (double *)realloc((void *)data, (n+delta_n)*sizeof(double))) == NULL)
72.         {
73.           printf("Fehler: realloc fehlgeschlagen!\n");
74.           exit(0);
75.         }
76.
```

```
77.       for(i1 = n; i1 < n+delta_n; i1++)
78.         {
79.           printf("Messwert %d: ", i1);
80.           scanf("%lf", data+i1);  // &data[i1] waere aequivalent zu data+i1
81.         }
82.
83.       n += delta_n;
84.
85.       // **********
86.
87.       for(i1 = 0; i1 < n; i1++)
88.         printf("data[%d] = %f\n", i1, data[i1]);
89.     }
90.
91.   // ********************
92.
93.   // Angeforderten Speicher freigeben.
94.   free(data);
95.   data = NULL;
96. }
```

```
Wie viele Messwerte willst Du eingeben? 3
Messwert 0: 2.1
Messwert 1: 3.2
Messwert 2: 4.3
data[0] = 2.100000
data[1] = 3.200000
data[2] = 4.300000
Willst Du weitere Messwerte eingeben (yes/no)? yes
Wie viele Messwerte willst Du eingeben? 1
```

```
Messwert 3: 7.0
data[0] = 2.100000
data[1] = 3.200000
data[2] = 4.300000
data[3] = 7.000000
Willst Du weitere Messwerte eingeben (yes/no)? yes
Wie viele Messwerte willst Du eingeben? 2
Messwert 4: 8.1
Messwert 5: 8.2
data[0] = 2.100000
data[1] = 3.200000
data[2] = 4.300000
data[3] = 7.000000
data[4] = 8.100000
data[5] = 8.200000
Willst Du weitere Messwerte eingeben (yes/no)? no
```

- **Beispiel:** dynamisches Anfordern von Speicherplatz für ein zweidimensionales **double**-Array (für eine Matrix) ...
  - Vorgehen in zwei Schritten:
    1. Für jede Zeile einen **double**-Zeiger anfordern, also insgesamt ein **double** *-Array.
    2. Für jeden dieser Zeiger ein separates **double**-Array anfordern.
  - Ein auf diese Weise dynamisch erzeugtes zweidimensionales Array variabler Größe kann einer Funktion übergeben werden, wobei in der Funktion problemlos mit **[***index1***][***index2***]** auf dessen Elemente zugegriffen werden kann; bei z.B. einem über **double A[5][5];** definierten Array ist dies nicht möglich (siehe oben, Umweg über **index**-Funktion, etc.).

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

// ********************

// Ausgabe einer Matrix M mit N x N Elementen.
void matrix_print(int N, double **M)
{
  int i1, i2;

  for(i1 = 0; i1 < N; i1++)
    {
      printf("| ");

      for(i2 = 0; i2 < N; i2++)
        printf("%+6.3lf ", M[i1][i2]);

      printf("|\n");
    }
}

// ********************

int main(void)
{
  int i1, i2;

```

```
28.  // **********
29.
30.  // Matrixgroesse eingeben.
31.
32.  int N;
33.  printf("Matrixgroesse N = ");
34.  scanf("%d", &N);
35.
36.  if(N <= 0)
37.    {
38.      printf("Fehler: N muss groesser als 0 sein.\n");
39.      exit(0);
40.    }
41.
42.  // **********
43.
44.  // Zweidimensionales Array dynamisch anfordern.
45.
46.  double **A;
47.
48.  // Zunaechst ein N-elementiges eindimsionales Array von double *-Werten ...
49.
50.  if((A = (double **)malloc(N*sizeof(double *))) == NULL)
51.    {
52.      printf("Fehler: malloc fehlgeschlagen!\n");
53.      exit(0);
54.    }
```

```
55.
56.   // ... dann fuer jeden dieser Zeiger ein weiteres N-elementiges
57.   // eindimsionales Array von double-Werten.
58.
59.   for(i1 = 0; i1 < N; i1++)
60.     {
61.       if((A[i1] = (double *)malloc(N*sizeof(double))) == NULL)
62.         {
63.           printf("Fehler: malloc fehlgeschlagen!\n");
64.           exit(0);
65.         }
66.     }
67.
68.   // ********************
69.
70.   // Irgendwelche Matrixoperationen (Tridiagonalmatrix anlegen und ausgeben)
71.   // ...
72.
73.   for(i1 = 0; i1 < N; i1++)
74.     {
75.       for(i2 = 0; i2 < N; i2++)
76.         {
77.           A[i1][i2] = 0.0;
78.
79.           if(i1 == i2)
80.             A[i1][i2] = -2.0;
81.
```

```
82.           if(abs(i1-i2) == 1)
83.             A[i1][i2] = +1.0;
84.         }
85.     }
86.
87.   matrix_print(N, A);
88.
89.   // ********************
90.
91.   // Angeforderten Speicher freigeben.
92.
93.   for(i1 = 0; i1 < N; i1++)
94.     free(A[i1]);
95.
96.   free(A);
97.   A = NULL;
98. }
```

```
Matrixgroesse N = 2
| -2.000 +1.000 |
| +1.000 -2.000 |
```

```
Matrixgroesse N = 5
| -2.000 +1.000 +0.000 +0.000 +0.000 |
| +1.000 -2.000 +1.000 +0.000 +0.000 |
| +0.000 +1.000 -2.000 +1.000 +0.000 |
| +0.000 +0.000 +1.000 -2.000 +1.000 |
| +0.000 +0.000 +0.000 +1.000 -2.000 |
```

```
Matrixgroesse N = 8
| -2.000 +1.000 +0.000 +0.000 +0.000 +0.000 +0.000 +0.000 |
| +1.000 -2.000 +1.000 +0.000 +0.000 +0.000 +0.000 +0.000 |
```

```
| +0.000 +1.000 -2.000 +1.000 +0.000 +0.000 +0.000 +0.000 |
| +0.000 +0.000 +1.000 -2.000 +1.000 +0.000 +0.000 +0.000 |
| +0.000 +0.000 +0.000 +1.000 -2.000 +1.000 +0.000 +0.000 |
| +0.000 +0.000 +0.000 +0.000 +1.000 -2.000 +1.000 +0.000 |
| +0.000 +0.000 +0.000 +0.000 +0.000 +1.000 -2.000 +1.000 |
| +0.000 +0.000 +0.000 +0.000 +0.000 +0.000 +1.000 -2.000 |
```

- **Beispiel:** erneut dynamisches Anfordern von Speicherplatz für ein zweidimensionales **double**-Array (für eine Matrix), diesmal eleganter, mit speziellen Funktionen zur Speicherverwaltung ...

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

// ********************

// Speicher fuer eine Matrix mit N x N Elementen anfordern.
void matrix_alloc(int N, double ***pM)
{
  int i1;

  // Zunaechst ein N-elementiges eindimsionales Array von double *-Werten ...

  if((*pM = (double **)malloc(N*sizeof(double *))) == NULL)
    {
      printf("Fehler: malloc fehlgeschlagen!\n");
      exit(0);
    }

```

```
19.   // ... dann fuer jeden dieser Zeiger ein weiteres N-elementiges
20.   // eindimsionales Array von double-Werten.
21.
22.   for(i1 = 0; i1 < N; i1++)
23.     {
24.       // !!!!! hier sind die Klammern um *pM essentiell !!!!!
25.       if(((*pM)[i1] = (double *)malloc(N*sizeof(double))) == NULL)
26.         {
27.           printf("Fehler: malloc fehlgeschlagen!\n");
28.           exit(0);
29.         }
30.     }
31. }
32.
33. // ********************
34.
35. // Speicher fuer eine Matrix mit N x N Elementen freigeben.
36. void matrix_free(int N, double ***pM)
37. {
38.   int i1;
39.
40.   for(i1 = 0; i1 < N; i1++)
41.     // !!!!! hier sind die Klammern um *pM essentiell !!!!!
42.     free((*pM)[i1]);
43.
44.   free(*pM);
45.   *pM = NULL;
```

```
46. }
47.
48. // ********************
49.
50. // Ausgabe einer Matrix M mit N x N Elementen.
51. void matrix_print(int N, double **M)
52. {
53.   int i1, i2;
54.
55.   for(i1 = 0; i1 < N; i1++)
56.     {
57.       printf("| ");
58.
59.       for(i2 = 0; i2 < N; i2++)
60.         printf("%+6.3lf ", M[i1][i2]);
61.
62.       printf("|\n");
63.     }
64. }
65.
66. // ********************
67.
68. int main(void)
69. {
70.   int i1, i2;
71.
72.   // Matrixgroesse eingeben.
```

```
73.
74.   int N;
75.   printf("Matrixgroesse N = ");
76.   scanf("%d", &N);
77.
78.   if(N <= 0)
79.     {
80.       printf("Fehler: N muss groesser als 0 sein.\n");
81.       exit(0);
82.     }
83.
84.   // **********
85.
86.   // Speicher fuer die NxN-Matrix A anfordern.
87.   double **A;
88.   matrix_alloc(N, &A);
89.
90.   // ********************
91.
92.   // Irgendwelche Matrixoperationen (Tridiagonalmatrix anlegen und ausgeben)
93.   // ...
94.
95.   for(i1 = 0; i1 < N; i1++)
96.     {
97.       for(i2 = 0; i2 < N; i2++)
98.         {
99.           A[i1][i2] = 0.0;
```

```
100.
101.          if(i1 == i2)
102.            A[i1][i2] = -2.0;
103.
104.          if(abs(i1-i2) == 1)
105.            A[i1][i2] = +1.0;
106.        }
107.    }
108.
109.  matrix_print(N, A);
110.
111.  // ********************
112.
113.  // Speicher fuer die NxN-Matrix A freigeben.
114.  matrix_free(N, &A);
115. }
```

```
Matrixgroesse N = 4
| -2.000 +1.000 +0.000 +0.000 |
| +1.000 -2.000 +1.000 +0.000 |
| +0.000 +1.000 -2.000 +1.000 |
| +0.000 +0.000 +1.000 -2.000 |
```

- **Beispiel:** erneut dynamisches Anfordern von Speicherplatz für ein zweidimensionales **`double`**-Array (für eine Matrix), **diesmal mit Absicht falsch** ...

```
4. ...
5.
6. // Speicher fuer eine Matrix mit N x N Elementen anfordern.
```

```
 7. void matrix_alloc(int N, double ***pM)
 8. {
 9.   int i1;
10.
11.   // Zunaechst ein N-elementiges eindimsionales Array von double *-Werten ...
12.
13.   if((*pM = (double **)malloc(N*sizeof(double *))) == NULL)
14.     {
15.       printf("Fehler: malloc fehlgeschlagen!\n");
16.       exit(0);
17.     }
18.
19.   // ... dann fuer jeden dieser Zeiger ein weiteres N-elementiges
20.   // eindimsionales Array von double-Werten.
21.
22.   for(i1 = 0; i1 < N; i1++)
23.     {
24.       // !!!!! jetzt mit Absicht falsch ... *pM[i1] statt (*pM)[i1] !!!!!
25.       if((*pM[i1] = (double *)malloc(N*sizeof(double))) == NULL)
26.         {
27.           printf("Fehler: malloc fehlgeschlagen!\n");
28.           exit(0);
29.         }
30.     }
31. }
32.
33. ...
```

```
Matrixgroesse N = 4
Speicherzugriffsfehler (Speicherabzug geschrieben)
```