4 Weitere Grundlagen der Programmiersprache C

4.1 Funktionen

Eine **Funktion** ist eine zusammengesetzte benannte Anweisung, die evtl. von Parametern abhängig ist. Dabei werden z. B.

- wiederkehrende Berechnungen und
- logisch zusammengehörige Programmschritte

zusammengefasst.

Bei der **Funktionsdefinition** wird dem Compiler mitgeteilt,

- wie die Funktion heißt,
- welche Datentypen die Argumente haben,
- welchen Datentyp das Ergebnis besitzt und
- welche Operationen beim Aufruf der Funktion auszuführen sind.

Für die Argumente (Parameter) werden dabei

- Namen festgelegt (**formale Parameter**) und
- die beim Aufruf angegebenen Ausdrücke als Werte zugewiesen (**aktuelle Parameter**).

Syntax einer Funktion

```
/* Funktionskopf */
Ergebnistyp Funktionsname (Typ1 Name1, Typ2 Name2, ...)
{
   /* Funktionsrumpf */
   lokale Anweisungen

   return Ergebnis;
}
```

Bemerkungen:

- `Ergebnistyp void` bedeutet, dass die Funktion kein Ergebnis zurückliefert; in diesem Fall kann das `return` weggelassen werden.

  ```
  void prozedur (int x) {...}
  ```

- Die Variablen im Funktionskopf sind lokal zu der Funktion.
- Für jede Variable muss der Datentyp einzeln angegeben werden.

  ```
  double x_plus_y (double x, int y) {return x+y;}
  ```

- Funktionen ohne Parameter haben `void` als Argument.

  ```
  int funktionOhneParameter (void) {return ergebnis;}
  ```

- Nach dem Funktionskopf steht *kein* Semikolon!

Deklaration von Funktionen (Prototyp)

- ermöglicht die Benutzung einer Funktion *vor* ihrer Definition.
- Dem Compiler wird nur die für die Benutzung der Funktion erforderliche Information mitgeteilt:
  - Name der Funktion,
  - Datentypen der Argumente und
  - Datentyp des Ergebnisses.
- Syntax wie bei der Funktionsdefinition außer, dass
  - am Ende des Funktionskopfes ein Semikolon steht
  - die Namen der Parameter fehlen können, d. h. eine reine Typ-Liste genügt.

  ```
  Ergebnistyp Funktionsname (Typ1 Name1, Typ2 Name2, ...);
  Ergebnistyp Funktionsname (Typ1, Typ2, ...);
  ```

- *Hinweis:* Werden bei der Deklaration Namen für Parameter angegeben, so dürfen in der Definition abweichende Namen gewählt werden.

4.2 Parameter der Hauptfunktion `main()`

Bisher: Funktionskopf der Hauptfunktion `main()`

```
int main (void)
```

d. h. `main` *ohne Parameter*.

Wollen wir einem Programm bei der Ausführung Werte übergeben, so muss der Funktionskopf der Hauptfunktion angepasst werden:

```
int main (int argc, char **argv)
```

Die Parameter sind

- `argc` (**arg**ument **c**ount); Datentyp `int`
  Anzahl der übergebenen Argumente (=von Leerzeichen getrennte Strings); der Programmname zählt dabei mit
- `argv` (**arg**ument **v**alues); Datentyp `char`-Zeiger-Array
  Werte der übergebenen Argumente

Beispiel:
Lautet der Programmaufruf eines Programms `meinProgramm` z. B.

```
./meinProgramm max 10 7.8
```

so gilt:

- `argc = 4`
- `argv[0] = "./meinProgramm"`, `argv[1] = "max"`, `argv[2] = "10"` und `argv[3] = "7.8"`

Achtung: Alle Argumente sind Zeichenketten (`strings`)! Daher ist eine **spezielle Typumwandlung** notwendig:

- Die C-Standard-Bibliothek muss eingebunden werden

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

- Zur Umwandlung in `int` („ASCII to int“): `atoi()`
  - z. B. `argv[2] = "10"` als `int`: int ganzeZahl = atoi(argv[2]);
- Zur Umwandlung in `double` („ASCII to float“): `atof()`
  - z. B. `argv[3] = "7.8"` als `double`: double zahl = atof(argv[3]);

4.3 Vergleiche und logische Operationen

Operator	Bedeutung
==	gleich
!=	ungleich
<	kleiner als
>	größer als
<=	kleiner oder gleich
>=	größer oder gleich
!	logisches NICHT, Negation
||	logisches ODER
&&	logisches UND

Ergebnisse von Vergleichen und logischen Operationen sind

0 für „falsch“

$\neq 0$ für „wahr“

4.4 Die Verzweigungen `if` und `switch`

Für Fallunterscheidungen innerhalb eines Programms können `if` bzw. `if` und `else` verwendet werden.

```
if (Ausdruck)
   /* Anweisungen_if */
```

oder

```
if (Ausdruck)
   /* Anweisungen_if */
else
   /* Anweisungen_else */
```

Erläuterungen:

- Wenn `Ausdruck` einen Wert $\neq 0$ besitzt, dann werden `Anweisungen_if` ausgeführt,
- sonst werden `Anweisungen_else` ausgeführt (nur falls der `else`-Zweig existiert).
- Sollen mehrere Anweisungen ausgeführt werden, müssen diese natürlich zwischen geschweiften Klammern { und } stehen.
- Die Anweisungen dürfen wiederum Fallunterscheidungen sein.

Beispiel:

```c
/* Funktion: Ist die eingegebene Zahl negativ oder nicht? */

#include <stdio.h>

int main (void)
{
   int zahl;

   printf ("Bitte eine ganze Zahl eingeben: ");
   scanf ("%d", &zahl);

   if (zahl < 0)
      printf("Die Zahl ist negativ.\n");
   else
      printf("Die Zahl ist nicht negativ.\n");

   return 0;
}
```


if-Kaskaden

```c
if (Ausdruck_1)
   /* Anweisungen_1 */
else if (Ausdruck_2)
   /* Anweisungen_2 */
else if (Ausdruck_3)
  /* Anweisungen_3 */
...
else
  /* Anweisungen_n */
```

Erläuterungen:

- Die Ausdrücke werden der Reihe nach ausgewertet.
- Ist `Ausdruck_i` der erste wahre Ausdruck, so werden
  - die `Anweisungen_i` ausgeführt und
  - die restlichen Ausdrücke nicht mehr ausgewertet.
- Ist keiner der Ausdrücke wahr, so werden die im `else`-Zweig aufgelisteten `Anweisungen_n` ausgeführt (nur falls der `else`-Zweig existiert).

Beispiel:

```c
/* Funktion: Ist die eingegebene Zahl negativ, positiv oder 0? */

#include <stdio.h>

int main (void)
{
   int zahl;

   printf ("Bitte eine ganze Zahl eingeben: ");
   scanf ("%d", &zahl);

   if (zahl < 0)
      printf("Die Zahl ist negativ.\n");
   else if (zahl > 0)
      printf("Die Zahl ist positiv.\n");
   else
      printf("Die Zahl ist gleich Null.\n");

   return 0;
}
```


Mehrfachverzweigungen mit switch

```c
switch (GanzzahligerAusdruck)
{
  case KonstanterGanzzahligerAusdruck_1:
    /* Anweisungen_1 */
    break;
  case KonstanterGanzzahligerAusdruck_2:
    /* Anweisungen_2 */
    break;
  ...
  default:
    /* Anweisungen_n */
}
```

Erläuterungen:

- Stimmt der Wert von `GanzzahligerAusdruck` mit `KonstanterGanzzahligerAusdruck_i` überein, so
  - wird das Programm bei `Anweisungen_i` fortgesetzt und
  - alle Anweisungen vor dem nächsten `break` oder dem Ende von `switch` ausgeführt.
- Wenn kein `case` passt, wird bei `default` fortgefahren.
- Die abgefragte Konstante darf nur in einem `case` vorkommen.
- Der `default`-Block ist optional, sollte aber verwendet werden.

4.5 Die Schleifen `for`, `while` und `do while`

Zählschleifen: `for`

```
for (Initialisierung; Bedingung; Aenderung)
{
   /* Anweisungen */
}
```

Erläuterungen:

- Zu Beginn (vor dem 1. Durchlauf) wird eine `Initialisierung` durchgeführt.
- Dann wird, solange `Bedingung` erfüllt ist,
  - der Schleifenrumpf `Anweisungen` ausgeführt,
  - dann der Ausdruck `Aktualisierung` ausgewertet
  - und schließlich die `Bedingung` neu ausgewertet.
- Jeder der drei Teile `Initialisierung`, `Bedingung` und `Aktualisierung` kann fehlen.

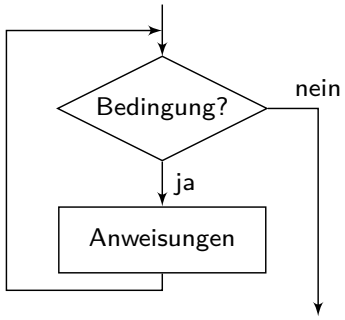
Beispiel:

```
1  /* Funktion: Berechnung und Ausgabe der ersten 20
2               Fibonacci-Zahlen */
3
4  #include <stdio.h>
5
6  int main (void)
7  {
8     int fib1=1, fib2=1, fib_tmp;
9
10    /* Ausgabe der ersten beiden Fibonacci-Zahlen */
11    printf("\n%4d\n", fib1);
12    printf("%4d\n", fib2);
13
14    /* Berechnung und Ausgabe der 3. bis 20. Fibonacci-Zahl */
15    for (int i=3; i <= 20; i++)
16    {
17       fib_tmp = fib1 + fib2;
18       fib1    = fib2;
19       fib2    = fib_tmp;
20       printf("%4d\n",fib_tmp);
21    }
22
23    return 0;
24 }
```

Abweisende Schleifen: `while`

```
while (Bedingung)
{
   /* Anweisungen */
}
```

Erläuterungen:



- Zu Beginn wird die `Bedingung` ausgewertet.
- Wenn sie erfüllt ist, dann wird, solange `Bedingung` erfüllt ist,
  - der Schleifenrumpf `Anweisungen` ausgeführt
  - und die `Bedingung` neu ausgewertet.
- Der Abbruch-Test erfolgt *vor* jedem Schleifen-Durchlauf
  $\Rightarrow$ die Schleife wird evtl. gar nicht durchlaufen

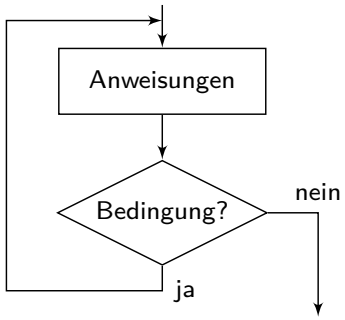
Beispiel:

```
1  /* Funktion: Berechnung und Ausgabe aller Fibonacci-
2               Zahlen kleiner 1000 */
3
4  #include <stdio.h>
5
6  int main (void)
7  {
8     int fib1=1, fib2=1, fib_tmp;
9
10    /* Ausgabe der ersten beiden Fibonacci-Zahlen */
11    printf("\n%4d\n", fib1);
12    printf("%4d\n", fib2);
13
14    /* Berechnung und Ausgabe aller Fibonacci-Zahlen < 1000 */
15    while ( (fib1 + fib2) < 1000 )
16    {
17       fib_tmp = fib1 + fib2;
18       fib1    = fib2;
19       fib2    = fib_tmp;
20       printf("%4d\n",fib_tmp);
21    }
22
23    return 0;
24 }
```

Nicht-abweisende Schleifen: `do ... while`

```
do
{
    /* Anweisungen */
} while (Bedingung);
```

Erläuterungen:



- Zu Beginn werden die `Anweisungen` ausgeführt.
- Dann wird die `Bedingung` ausgewertet.
- Wenn sie erfüllt ist, dann wird, solange `Bedingung` erfüllt ist,
  - der Schleifenrumpf `Anweisungen` ausgeführt
  - und die `Bedingung` neu ausgewertet.
- Der Abbruch-Test erfolgt *nach* jedem Schleifen-Durchlauf
  $\Rightarrow$ die Schleife wird mindestens einmal durchlaufen

4.6 Zeiger (Pointer)

Zeigervariablen (**Pointer**) sind Variablen, deren Wert eine *Adresse* im Speicher ist.

- Idee: Zeiger „zeigen“ auf eine Stelle im Speicher, an der eine „normale“ Variable abgespeichert ist.
- Zeigervariablen werden in C deklariert, indem wir einen Stern (`*`) *vor den Variablennamen* schreiben, z. B.:

  ```
  int *pi;
  ```

  deklariert den Zeiger `pi`, der auf eine Variable vom Typ `int` „zeigen“ kann, d. h. dessen Inhalt als Adresse einer `int`-Variablen zu interpretieren ist.
- Die Adresse einer Variablen kann man mittels des **Adressoperators** `&` ermitteln und einem Zeiger zuweisen, z. B.:

```
1 int  a = 13;
2 int *pa = &a;
```

- Umgekehrt liefert **Dereferenzieren** mit dem **Dereferenzierungs-operator** `*` den Inhalt einer Zeigervariablen, z. B.

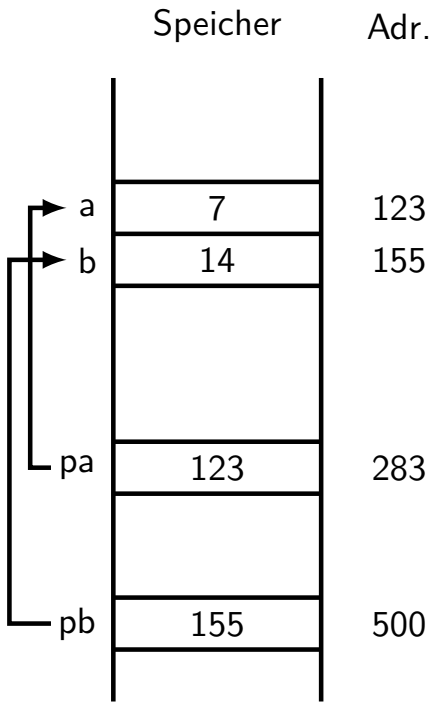
```
3 int  b = *pa;  /* entspricht int b = a; */
```

Anschauliche Interpretation von Zeigern

```
int a = 7, b = 14;
int *pa, *pb;

pa = &a;
pb = &b;
```

- pa „zeigt“ auf a
- pb „zeigt“ auf b



Bemerkungen:

- Werden mehrere gleichartige Zeiger zusammen deklariert, so muss * jedesmal angegeben werden, z. B.:

```
int *a, *b; /* entspricht int *a; int *b; */
int *a, b;  /* entspricht int *a; int b;  */
```

- Bei

```
int    *a;
double *b;
```

  gilt:
  - Der Typ von a ist „Zeiger auf eine int-Variable“
  - Der Typ von b ist „Zeiger auf eine double-Variable“
  - Die Zuweisung

    ```
    b = a;
    ```

    ist nicht zulässig (verschiedene Typen!)
- Am Typ eines Pointers p kann der Compiler erkennen,
  - wie groß der Speicherbereich ist, auf den p zeigt und
  - wie die dort abgelegte Bitfolge zu interpretieren ist.

Warum sind Zeiger wichtig?

```
1  /* Funktion: Vertauschung zweier ganzer Zahlen mit Hilfe
2               einer Funktion (1. Versuch) */
3
4  #include <stdio.h>
5
6  /* Vertauschung zweier ganzer Zahlen */
7  void tauschen (int a, int b)
8  {
9      int tmp=a;
10     a = b;
11     b = tmp;
12     printf("tauschen: a = %d, b = %d\n", a, b);
13 }
14
15 int main (void)
16 {
17     int a=7, b=14;
18     tauschen(a,b);
19     printf("main: a = %d, b = %d\n", a, b);
20     return 0;
21 }
```

Was wird in den Zeilen 12 und 19 ausgegeben?

Was läuft im 1. Versuch schief?

```
/* ... */
void tauschen (int a, int b)
{
    int tmp=a;
    a = b;
    b = tmp;
}

int main (void)
{
    int a=7, b=14;
    tauschen(a,b);
/* ... */
```

- In den Funktionen `tausche` und `main` gibt es jeweils die *lokalen Variablen* `a` und `b`
  - → in `tausche` werden die Werte von `a` und `b` getauscht
  - → in `main` bleiben die Werte von `a` und `b` unverändert

Zeiger lösen das Problem

```c
/* Funktion: Vertauschung zweier ganzer Zahlen mit Hilfe
             einer Funktion (2. Versuch) */

#include <stdio.h>

/* Vertauschung zweier ganzer Zahlen.
   Die Parameter von tauschen sind jetzt Zeiger auf int.
   tauschen(&a, &b): pa zeigt auf a und pb zeigt auf b. */
void tauschen (int *pa, int *pb)
{
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
    printf("tauschen: a = %d, b = %d\n", *pa, *pb);
}

int main (void)
{
    int a=7, b=14;
    /* Uebergabe der Adressen von a und b an tauschen. */
    tauschen(&a, &b);
    printf("main: a = %d, b = %d\n", a, b);
    return 0;
}
```


Vorteile von Zeigern

- Mit Hilfe von Zeigern können Variablen, die in einer Funktion deklariert wurden, in einer anderen Funktion geändert werden.
  → Verbesserung der Übersichtlichkeit bei langen Quellcodes
- Speicherbedarf von Zeigern: 64 Bit (64 Bit System) *unabhängig vom Datentyp*
  → effiziente Speichernutzung möglich

4.7 Felder (Arrays)

Ein **Feld** (**array**) kann eine *feste Anzahl* von Komponenten *desselben Datentyps* speichern.

- Felder liegen als *zusammenhängender Block* im Speicher.
- Unterschied zu „normalen“ Variablen: Felder besitzen einen Index. *Achtung:* In C beginnt der Index bei 0! Zu einem Feld der Länge 3 gehören also die Indizes 0, 1 und 2.
- Zur Deklaration schreiben wir die Größe eines Feldes (d. h. die Anzahl Einträge) in *eckigen Klammern hinter den Variablennamen*:

```
Datentyp Feld_Name[Laenge];
```

- Felder können ein-, zwei- oder auch mehrdimensional sein, z. B.

```
int    x[3];
double A[10][5];
```

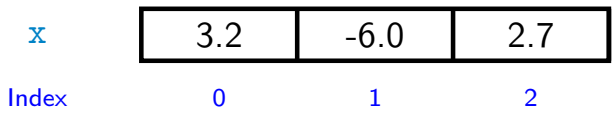
  - `x` ist ein eindimensionales Feld der Länge 3 vom Typ `int` (z. B. ein Vektor mit 3 Einträgen)
  - `A` ist ein zweidimensionales Feld mit 10∗5 Elementen vom Typ `double` (z. B. eine 10×5-Matrix)

Initialisierung von eindimensionalen Feldern

- Felder können durch eine **Initialisierliste** mit *konstanten Ausdrücken* hinter der Variablenvereinbarung initialisiert werden, z. B.

```
double x[3] = {3.2, -1.5*4.0, 2.7};
```

  → reserviert einen zusammenhängenden Speicherblock für 3 `double`-Variablen
  → Initialisiert die Einträge 0 bis 2 des Feldes `x` mit den Werten 3.2, -6.0 und 2.7



- *Hinweis:* Ist die Länge der Initialisierliste kleiner als die Feldgröße, so werden die restlichen Werte auf 0 gesetzt, z. B.

```
double x[1000] = {0};
```

initialisiert alle 1000 Einträge des Feldes `x` mit 0.

Felder und Zeiger

- Zum **Zugriff** auf die Werte eines Feldes verwenden wir *eckige Klammern*, z. B.:

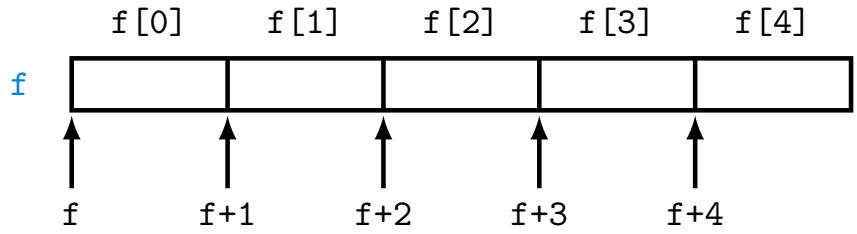
```
double x[3] = {3.2, -1.5*4.0, 2.7};
printf ("x_1 = %.2f\n", x[0]);
```

liefert bei Ausführung des Programms die Ausgabe

```
x_1 = 3.20
```

- Intern werden Zugriffe mit Hilfe von Zeigern umgesetzt:
Der Feldname ist intern ein Zeiger auf den 1. Eintrag des Feldes und es gilt:

`f[i]` ist äquivalent zu `*(f + i)`



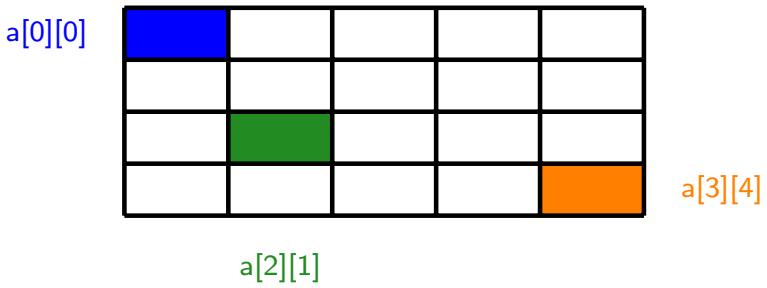
Zweidimensionale Felder

- Bei der Deklaration werden 2 Feldgrößen angegeben, z. B. mit

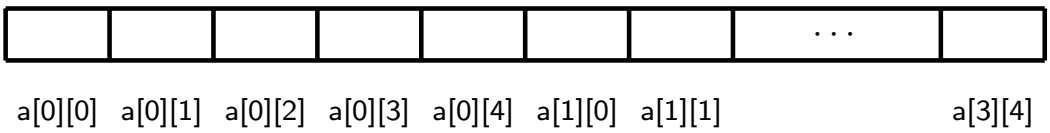
```
double a[4][5];
```

deklariert man ein 4×5-Feld a vom Typ `double`

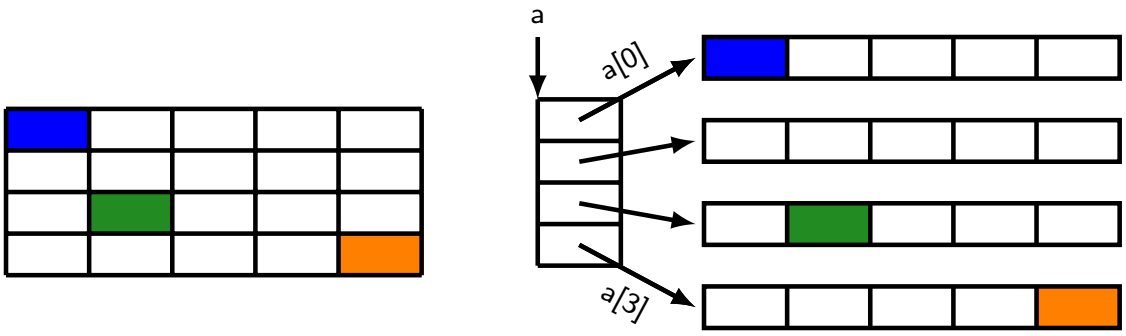
- Der Zugriff auf die Komponenten erfolgt über den **Indexoperator** „[...]“, wobei ein Klammerpaar [...] je Dimension verwendet wird.



- intern lineare Ablage der Komponenten im Speicher „*zeilenweise*“
  - „letzter“ Index läuft „am schnellsten“
  - „erster“ Index läuft „am langsamsten“



Anschauliche Interpretation von zweidimensionalen Feldern



- Intern ist `a[ i ]` ein Zeiger auf die `i`-te Zeile des Feldes
- Die `i`-te Zeile des Feldes ist wieder ein Feld – also ein Zeiger

→ Zweidimensionale Felder sind intern also **Zeiger auf Zeiger**

Was macht C intern aus dem Zugriff `a[ i ][ j ]`?

```
a[ i ][ j ] = ( a[ i ] )[ j ]
            = *( ( a[ i ] ) + j )
            = *( *( a + i ) + j )
```