# Modellierung und Programmierung

Dr. Martin Riplinger

21.11.2012



## Funktionen – elementare Merkregeln

**Deklaration und Definition**

- Die Funktions**deklaration** steht im Code vor `main`.
- Die Funktions**definition** kommt ans Ende der Datei (unauffällig „versteckt“).
- Anzahl und Datentypen der Parameter sowie Rückgabetyp müssen übereinstimmen.

  **Guter Stil**: Deklaration mit Kommentar versehen, der Parameter und Rückgabewert erläutert
- Der Name soll die **Tätigkeit** (Rückgabetyp `void`) bzw. den **zurückgegebenen Wert** (nichtleerer Rückgabetyp) widerspiegeln.
- Jede Funktion sollte durch ein `return [Wert];` beendet werden.

**Aufruf**

- Die Anzahl der Parameter muss konsistent mit der Deklaration sein.
- Es findet **Call by value** statt, d. h. die Funktion hat *nicht die Parameter selbst, sondern nur die darin gespeicherten Werte* zur Verfügung. Daher können diese auch **nicht permanent verändert** werden!
- Die übergebenen Werte werden automatisch in die Datentypen laut Deklaration umgewandelt („gecastet“). Bei inkompatiblen Typen **warnt** der Compiler lediglich.
- Funktionen „sehen“ nur **globale** Variablen, übergebene **Parameter** sowie **lokale Variablen im Funktionsrumpf**.

# Statische Felder

## Definition

Ein Feld (engl. *array*) ist die Zusammenfassung von Elementen *gleichen Typs* zu einer aufrufbaren Einheit.

- **Deklaration**: `Datentyp Feldname[Anzahl];`

  Legt ein Feld von `Anzahl` Elementen des Typs `Datentyp` an.

  **Achtung**: `Anzahl` muss ein *positiver ganzzahliger* Wert sein.

- Wie bei primitiven Datentypen ist eine Initialisierung bei der Deklaration möglich:

```
int N[4] = {1, 3, -5, 42};
double x[] = {1.9, -3.1415, 5.73e+21};
  // Die Groesse (3) wird hier vom Compiler automatisch bestimmt

int p = 23;                          // Wichtig: p initialisieren!
unsigned j[p] = {1, 0, 3};           // Rest bleibt uninitialisiert

float y[2] = {1.0, 3, -7.2};         // Fehler: zu viele Elemente!
```

# Statische Felder

- Die Deklaration `float x[5];` legt ein Feld der Länge 5 an. Die Komponenten (Feldeinträge, Feldelemente) sind dabei alle vom Typ `float`.

  Folgerung: Das Feld mit Bezeichner `x` belegt im Arbeitsspeicher $5 \cdot 4 = 20$ Byte.

- Auf die Komponenten kann mittels `x[0]`, `x[1]`, `x[2]`, `x[3]` und `x[4]` zugegriffen werden. Beispiel:

```
x[0] = 11.0;
x[1] = 12.0;
x[2] = 13.0;
x[3] = 14.0;
x[4] = 15.0;
```

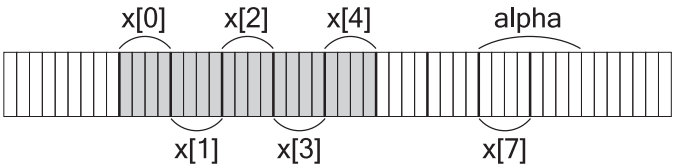
- **Merke**: Die Indizierung von Feldeinträgen beginnt in C stets mit 0!

| 1. Komp. | 2. Komp. | 3. Komp. | 4. Komp. | 5. Komp. | |
|---|---|---|---|---|---|
| 4 Byte | 4 Byte | 4 Byte | 4 Byte | 4 Byte | ← Größe |
| x[0] | x[1] | x[2] | x[3] | x[4] | ← "Name" |
| 11.0 | 12.0 | 13.0 | 14.0 | 15.0 | ← Inhalt |

- **Merke**: Alle Komponenten eines Feldes werden vom Compiler direkt **hintereinander** im Arbeitsspeicher abgelegt.

## Einschränkungen und Stolperfallen

- Einmal festgelegt, kann die Größe eines statischen Feldes nicht mehr verändert werden.
- Der maximalen Größe eines Feldes sind enge Grenzen gesetzt (Gefahr eines **stack overflow** = Stapelüberlauf).
- Die Rückgabe eines Feldes durch eine Funktion oder die Übergabe eines Feldes als Parameter einer Funktion ist **nicht möglich**, da es sich nicht um einen *primitiven*, sondern einen *zusammengesetzten* Datentyp handelt.
- Beim Zugriff auf ein Feldelement außerhalb des zulässigen Indexbereichs erfolgt im allgemeinen **keine Fehlermeldung**! Auch der Compiler warnt nicht!




Die Zuweisung `x[7] = 241.98` schreibt in den Bereich, der (zufällig) von `alpha` belegt wird.

**Mögliche Folge**: Das Programm wird völlig unberechenbar!

Häufigste Fehlerquelle: `int x[N]; ... x[N] = 1;`

Tritt zumeist dann auf, wenn `x[1]` statt `x[0]` als erster Eintrag interpretiert wird.

## Beispiel: Euklidische Norm eines Vektors im $\mathbb{R}^3$

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

int main(void)
{
  double x[3], norm2 = 0.0, norm;
  int i;

  // Vektor einlesen
  for (i=0; i<3; i++)       // Vorsicht: Indices beginnen bei 0!
  {
    printf("Geben Sie die %d-te Komponente ein: ", i+1);
    scanf("%lf", &x[i]);
  }

  // Vektor ausgeben
  printf("\nDer Vektor hat folgende Eintraege:\n");
  for (i=0; i<3; i++)
    printf("x[%d] = % 7.4lf\n", i, x[i]);

  // Berechnung der Summe der Komponentenquadrate
  for (i=0; i<3; i++)
  norm2 += x[i]*x[i];

  norm = sqrt(norm2);       // euklidische Norm

  printf("\nDie Norm des Vektors ist % .4lf.\n\n", norm);
  return 0;
}
```

## Beispiel: Euklidische Norm eines Vektors im $\mathbb{R}^3$

**Ausgabe**:

```
Geben Sie die 1-te Komponente ein: 1
Geben Sie die 2-te Komponente ein: -2
Geben Sie die 3-te Komponente ein: 2

Der Vektor hat folgende Eintraege:
x[0] =  1.0000
x[1] = -2.0000
x[2] =  2.0000

Die Norm des Vektors ist  3.0000.
```

## Mehrdimensionale Felder

**Deklaration**: `Datentyp Feldname[dim1][dim2]...[dimN];`

Beachte: `A[4][3]` wird (unabhängig vom Typ) im Speicher „zeilenweise“ abgelegt:

`A[0][0]`, `A[0][1]`, `A[0][2]`, `A[1][0]`, ..., `A[1][2]`, `A[2][0]`, ..., `A[3][2]`

Mathematische Interpretation: $\mathbf{A} = (a_{ij})_{\substack{1 \le i \le 4 \\ 1 \le j \le 3}} \in \mathbb{R}^{4 \times 3}$

$$\begin{bmatrix} \texttt{A[0][0]} & \texttt{A[0][1]} & \texttt{A[0][2]} \\ \texttt{A[1][0]} & \texttt{A[1][1]} & \texttt{A[1][2]} \\ \texttt{A[2][0]} & \texttt{A[2][1]} & \texttt{A[2][2]} \\ \texttt{A[3][0]} & \texttt{A[3][1]} & \texttt{A[3][2]} \end{bmatrix} \longleftrightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{bmatrix}$$

## 2D-Felder als Matrizen

**Beispiel**: Zeilenweises Einlesen der Komponenten einer $2 \times 3$-Matrix vom Typ `int`

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int i, j;
6    int A[2][3];
7
8    for(i=0; i<2; i++)
9    {
10     for(j=0; j<3; j++)
11     {
12       printf("A[%d][%d] = ", i, j);
13       scanf("%d", &A[i][j]);
14     } // for j
15   } // for i
16
17   return 0;
18 }
```

## 2D-Felder als Matrizen

Auch bei mehrdimensionalen Feldern ist eine direkte Initialisierung möglich, bspw.

```
int A[3][2] = {{11,12},{21,22},{31,32}};
```

Da die Komponenten im Speicher in einer Reihe angeordnet sind, ist obige Zeile äquivalent zu

```
int A[3][2] = {11,12,21,22,31,32};
```

Erfolgt bei der Deklaration eine partielle Initialisierung, so wird mit Nullen aufgefüllt:

```
int A[3][3] = {{1,2},{3},{4,5}};
```

generiert die Matrix

$$\mathtt{A} = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 0 & 0 \\ 4 & 5 & 0 \end{bmatrix}.$$

## Zeichenketten

- Zeichenketten (engl. *strings*) sind formal nichts anderes als Felder vom Typ `char`.
- Eine Zeichenkette (`"Hallo Welt!\n"`) wird vom Compiler automatisch als Feld von Zeichen dargestellt. Dabei wird am Schluss automatisch ein zusätzliches Zeichen `'\0'` (*Nullzeichen*) angehängt, um das Stringende zu markieren.
- Stringverarbeitungsfunktionen benötigen unbedingt das Nullzeichen, damit sie das Ende eines Strings erkennen. Somit muss bei der Deklaration ein zusätzlicher Speicherplatz für `'\0'` eingeplant werden.

  **Beispiel**:

```
char wort[6] = "Hallo";
```

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| H | a | l | l | o | \0 |

```
printf("Der 2. Buchstabe von wort ist ein '%c'.\n", wort[1]);
```

  **Ausgabe**: `Der 2. Buchstabe von wort ist ein 'a'.`

## Zeichenketten

- Direkte Initialisierung: als String

```
char wort[] = "Hallo";
```

  oder als Feld von Zeichen

```
char wort[] = {'H', 'a', 'l', 'l', 'o', '\0'};
```

- Einlesen von Zeichenketten mittels `fgets`:

```
fgets(Zielstring, Anzahl + 1, stdin);
```

  liest Zeichen von der Tastatur ein, bis ein 'Return' eingegeben wird, und speichert die ersten `Anzahl` Zeichen und ein abschließendes `'\0'` im `Zielstring`.

  **Bemerkung**: Die „einfachere“ Funktion `gets` prüft nicht die Länge des Zielstrings. Daraus ergeben sich u. U. *gravierende* Sicherheitsmängel (Gefahr eines Überlaufs), weshalb die Funktion unter Programmierern „geächtet“ ist.

- Ausgabe via `printf` wie üblich. Der Platzhalter ist `%s`.

```
printf("%s\n", wort);
```

# Operationen mit Strings (kleine Auswahl)

```
#include <string.h>
```

| | |
|---|---|
| `strlen(s)` | liefert Länge von `s`, abschließendes `'\0'` nicht mitgezählt |
| `strncpy(s, t, n)` | kopiert höchstens `n` Zeichen von `t` nach `s` (*) |
| `strncat(s, t, n)` | hängt `n` Zeichen von `t` an das Ende von `s` an (*) |
| `strcmp(s, t)` | $\begin{cases} > 0 & , \text{ falls } s \text{ lexikographisch kleiner als } t \text{ ist} \\ < 0 & , \text{ falls } s \text{ lexikographisch größer als } t \text{ ist} \\ 0 & , \text{ falls } s \text{ und } t \text{ identisch sind} \end{cases}$ |
| | Beachte: `if (strcmp(s, t))` testet auf Ungleichheit! |
| `strncmp(s, t, n)` | wie `strcmp`, aber nur für die ersten `n` Zeichen |

(*) **Vorsicht**: Der Programmierer ist dafür verantwortlich, dass in `s` genügend Platz vorhanden ist, um `n` Zeichen zu speichern. Das sollte immer geprüft werden!

# Operationen mit Strings: Beispiel

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

int main(void)
{
  char str1[] = "Modellierung", vl_eigenschaft[8], zu_kurz[3], vl_name[51];

  strncpy(vl_name, str1, strlen(str1));            // okay

  // simpler Test auf ausreichende Laenge
  if (50 >= strlen(vl_name) + strlen(" und ") + strlen("Programmierung"))
  {
    strncat(vl_name, " und ", 5);
    strncat(vl_name, "Programmierung", strlen("Programmierung"));
  }

  printf("Wie findest du %s? ", vl_name);
  fgets(vl_eigenschaft, 8, stdin);

  printf("\n%s ist deiner Meinung nach %s\n", vl_name, vl_eigenschaft);

  strncpy(zu_kurz, str1, strlen(str1));            // Ueberlauf!
  printf("Inhalt von zu_kurz: %s\n", zu_kurz);
  printf("Inhalt von vl_eigenschaft: %s\n", vl_eigenschaft);

  return 0;
}
```

## Operationen mit Strings: Beispiel

**Ausgabe**:

```
Wie findest du Modellierung und Programmierung? okay

Modellierung und Programmierung ist deiner Meinung nach okay

Inhalt von zu_kurz: Modellierungodellierung
Inhalt von vl_eigenschaft: ellierungodellierung
```

(Die Ausgabe der letzten beiden Zeilen hängt vom Speicherlayout ab.)

**Vorsicht**:

- Offenbar warnt der Compiler nicht vor dem Überlauf in Zeile 22, und auch zur Laufzeit tritt kein Fehler auf.
- Die Funktion `strncpy` schreibt über die Feldgrenzen von `zu_kurz` hinaus in einen Bereich, der (möglicherweise) von einer anderen Variable belegt wird.
- Da ein String erst mit dem Nullzeichen als beendet gilt, wird von `printf` bei der Ausgabe des fehlerhaften Strings *aus einem fremden Bereich gelesen*!

**Ergebnis**: Ein unberechenbares Programm mit einem äußerst schwierig zu lokalisierenden Fehler.

## Umwandlung von Strings

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

**String** $\longrightarrow$ **Ganzzahl**: `int atoi(Zeichenkette)`

```
int n;
char s[11];

printf("Es ist atoi(\"101\") = %d \n", atoi("101"));

n = atoi("3218");
printf("Es ist atoi(\"3218\") = %d \n", n);

strncpy(s, "-157", 10);
printf("Es ist atoi(s) = %d \n", atoi(s));
```

**Ausgabe**:

```
Es ist atoi("101") = 101
Es ist atoi("3218") = 3218
Es ist atoi(s) = -157
```

## Umwandlung von Strings

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

**String** $\longrightarrow$ **Gleitkommazahl**: `double atof(Zeichenkette)`

```
double x;
char s[11];

printf("Es ist atof(\"101.32\") = %lf \n", atof("101.32"));

x = atof("3218.927");
printf("Es ist atof(\"3218.927\") = %lf \n", x);

strncpy(s, "-157.58", 10);
printf("Es ist atof(s) = %lf \n", atof(s));
```

**Ausgabe**:

```
Es ist atof("101.32") = 101.320000
Es ist atof("3218.927") = 3218.927000
Es ist atof(s) = -157.580000
```

## Größe von Datentypen und Speicherobjekten: `sizeof`

**Allgemein**

- `#include <stdlib.h>`
- Rückgabetyp von `sizeof` ist `size_t`.
  Dabei gilt: `size_t` ist ganzzahlig und vorzeichenlos, entspricht `unsigned (int)` oder `unsigned long`.

`sizeof` gibt den Speicherbedarf eines Datentyps in Byte aus:

```
size_t sizeof(Datentyp);
```

```
printf("sizeof(char) = %u, ",sizeof(char));
printf("sizeof(int) = %u\n",sizeof(int));
printf("sizeof(float) = %u, ",sizeof(float));
printf("sizeof(double) = %u\n",sizeof(double));
```

**Ausgabe**:

```
sizeof(char) = 1, sizeof(int) = 4
sizeof(float) = 4, sizeof(double) = 8
```

## Größe von Datentypen und Speicherobjekten: `sizeof`

`sizeof` liefert den Speicherbedarf eines deklarierten Speicherobjekts in Byte:

```
size_t sizeof Speicherobjekt;
```
oder
```
size_t sizeof(Speicherobjekt);
```

**Beispiel**

```
int i;                                  sizeof i ⇝ 4
int j=42;                               sizeof j ⇝ 4
float x;                                sizeof x ⇝ 4
float y=3.1415;                         sizeof y ⇝ 4
double z=M_PI;                          sizeof z ⇝ 8
char s[10];                             sizeof s ⇝ 10
char t[]= "Ay, caramba!";               sizeof t ⇝ 13
int a[6];                               sizeof a ⇝ 24
int b[]={3,2,1,8};                      sizeof b ⇝ 16
float c[]={1.1,2.2,3.3};                sizeof c ⇝ 12
double A[3][2];                         sizeof A ⇝ 48
int B[][] = {{11,12,13},{21,22,23}};    sizeof B ⇝ error
```

## Speicheradressen

**Adress-Operator `&`**

Erinnerung:

```
int a;
scanf("%d",&a);
```

**Beispiel**: Adressen von Skalaren:

```
int i=1, j=9;
double x=2.7, y=M_PI;
printf("Adresse von i = %p\t",&i);
printf("Wert von i = %d\n", i);
printf("Adresse von j = %p\t",&j);
printf("Wert von j = %d\n", j);
printf("Adresse von x = %p\t",&x);
printf("Wert von x = %lf\n", x);
printf("Adresse von y = %p\t",&y);
printf("Wert von y = %lf\n", y);
```

**Ausgabe**:

```
Adresse von i = 0x28abf4     Wert von i = 1
Adresse von j = 0x28abf0     Wert von j = 9
Adresse von x = 0x28abe8     Wert von x = 2.700000
Adresse von y = 0x28abe0     Wert von y = 3.141593
```

## Speicheradressen

**Beispiel**: Adressen von eindimensionalen Feldkomponenten

```
char c[] ="Wetterwachs";
int a[] = {1,2,3};
printf("Adresse von c[0] = %p\n",&c[0]);
printf("Adresse von c[1] = %p\n",&c[1]);
printf("Adresse von c[2] = %p\n",&c[2]);
printf("Adresse von c[3] = %p\n",&c[3]);
printf("Adresse von a[0] = %p\n",&a[0]);
printf("Adresse von a[1] = %p\n",&a[1]);
printf("Adresse von a[2] = %p\n",&a[2]);
```

**Ausgabe**:

```
Adresse von c[0] = 0x22ccb0
Adresse von c[1] = 0x22ccb1
Adresse von c[2] = 0x22ccb2
Adresse von c[3] = 0x22ccb3

Adresse von a[0] = 0x22cca0
Adresse von a[1] = 0x22cca4
Adresse von a[2] = 0x22cca8
```

## Speicheradressen

**Beispiel**: Adressen von mehrdimensionalen Feldkomponenten

```
double A[3][2] = {{11,12},{21,22},{31,32}};
printf("Adresse von A[0][0] = %u\n",&A[0][0]);
printf("Adresse von A[0][1] = %u\n",&A[0][1]);
printf("Adresse von A[1][0] = %u\n",&A[1][0]);
printf("Adresse von A[1][1] = %u\n",&A[1][1]);
printf("Adresse von A[2][0] = %u\n",&A[2][0]);
printf("Adresse von A[2][1] = %u\n",&A[2][1]);
```

```
Adresse von A[0][0] = 2280560
Adresse von A[0][1] = 2280568
Adresse von A[1][0] = 2280576
Adresse von A[1][1] = 2280584
Adresse von A[2][0] = 2280592
Adresse von A[2][1] = 2280600
```

**Überlegungen:**

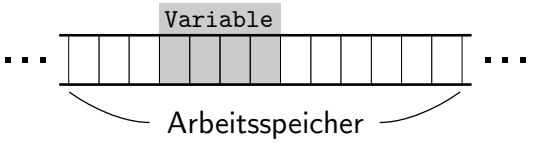
- Wenn man die Adresse einer Variable kennt, wird man auch ihren Inhalt, d.h. ihren Wert manipulieren können.
- Da man direkt auf den Arbeitsspeicher zugreift, sollte dies auch von jeder Stelle im Programm aus möglich sein - vorausgesetzt die Adresse im Speicher ist bekannt.

# Zeiger: Definition und Prinzip

## Definition

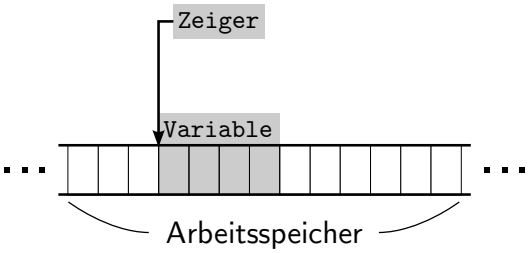
Ein Zeiger (engl. *pointer*) ist eine Variable, deren Inhalt eine Speicheradresse und der zugehörige Datentyp ist.

**Funktionsprinzip**:




**Zugriff über Variablennamen**

- Variablenname steht stellvertretend für einen Speicherbereich
- Verknüpfung ist mit Deklaration festgelegt und unveränderbar
- Bei Zugriff wird die Speicher-Referenz **intern** aufgelöst und der Wert anschließend gelesen bzw. geschrieben.




**Zugriff über Zeiger**

- Speicherbereich ist **Inhalt** der Zeigervariablen
- Dieser Inhalt ist selbstverständlich veränderbar → Zeiger kann „umgebogen“ werden
- Auflösung der Speicherreferenz geschieht **explizit**, d. h. durch den Programmierer

# Zeiger

**Wozu überhaupt Zeiger**?

- Indem man die Speicheradresse einer Variablen an eine Funktion übergibt, ermöglicht man es, den Inhalt der Variablen innerhalb der Funktion *permanent* zu verändern (**Call by reference**).
- Mit Zeigern lässt sich wie mit Feldern umgehen, mit dem Unterschied, dass sie als Parameter und als Rückgabewert von Funktionen verwendet werden können (**dynamische Felder**).
- Die gesamte Verwaltung von Speicherbereichen zur Laufzeit geschieht mit Hilfe von Zeigern (**dynamische Speicherverwaltung**).
- Zeiger ermöglichen es, aus einer Menge von Funktionen zur Ausführung einer bestimmten Aufgabe *zur Laufzeit* eine Variante auszuwählen (**Callback-Prinzip**).