# Programmierung 1

## Arrays

Dr. Monika Schak

*Woche 7*

03. Dezember 2025

# Wiederholung

- Modularisierung über Header-Dateien $\rightarrow$ Trennung von Schnittstelle und Implementierung
- Header-Dateien: Funktionsprototypen, Konstanten, Typdefinitionen, Schnittstellendokumentation (Include-Guards)
- c-Dateien: Funktionsdefinitionen, interne Hilfsfunktionen, einbinden der Header-Datei
- Rekursion = Funktion ruft sich selbst wieder auf
- Rekursive Funktion besteht aus:
  - Basisfall: Abbruchbedingung
  - Rekursionsschritt: Erneuter Aufruf mit „einfacherem“ Problem

# Arrays

- Oft müssen viele Werte desselben Datentyps gespeichert werden
- Dann ist es praktisch, diese Werte konsekutiv im Speicher abzulegen
- Dabei wird über einen Index auf die einzelnen Werte zugegriffen

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Inhalt | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 |

- Arrays enthalten immer nur Elemente des **gleichen** Datentyps
- Die Anzahl der Elemente ist unveränderbar (Arrays haben eine feste Länge!)
- Die Elemente an den einzelnen Indizes (Positionen) sind aber änderbar
- Indexzugriff erlaubt wahlfreien Zugriff auf einzelne Array-Elemente

# Erzeugen von Arrays

- Arraylänge wird bei Variablendeklaration in eckigen Klammern Klammern angegeben und an den Bezeichner angehängt
  - Beispiel 1 (Integer-Array mit zehn Elementen): `int werte[10];`
  - Beispiel 2 (Char-Array): `char str[] = {'p', 'r', 'o', 'g', '\0'};`
    Hier wird das Array explizit mit Mengenschreibweise unter Angabe aller Elemente deklariert und initialisiert. Die Arraylänge ergibt sich automatisch aus der Anzahl der Elemente
- Achtung: Ein Array kennt seine eigene Länge nicht! Die Arraylänge muss also zusätzlich abgespeichert werden.

# Elementzugriff

- Indizes gehen bei einer Arraylänge von $n$ von 0 bis $n - 1$.
- Auf einzelne Arraypositionen greift man zu, indem man einen Integer-Ausdruck in eckige Klammern hinter den Arraynamen schreibt.
- Das Überschreiten der Arraygrenzen ist ein Fehler, wird in C aber **nicht** geprüft!

```
const int N = 10;
// Deklaration eines Array der Länge N
int feld[N];
// Initialisierung aller Arraywerte mit 0
for (int i = 0; i < N; i++) {
  feld[i] = 0;
}
// Zugriff auf 5. Element (Startindex 0)
printf("feld[%d] ist %d.\n", 4, feld[4]);
```

# Arrays im Speicher

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

- Arrays sind Adresskonstanten
  - Arrayname (ohne Index dahinter) enthält als Wert die konstante Anfangsadresse des Array (Adresse des ersten Elements)
  - Arrays kann man deshalb in C einander **nicht** zuweisen
  - Arrays kann man **nicht** mit == vergleichen, da man so nur die Anfangsadressen vergleicht
- Arrays sind zusammenhängende Speicherbereiche
  - Ein Array zu indizieren bedeutet, intern einen Offset auf die Adresse des ersten Feldelements zu addieren
  - Die Größe des Offsets ist abhängig vom Typ, bei `int` i.d.R. 4 Byte.

# Übung

- Geben Sie an, wie ein Integer-Array der Länge 100 deklariert wird.
- Deklarieren Sie ein Float-Array und initialisieren Sie es direkt mit Beispielwerten.
- Warum sollte die Länge eines Arrays (z.B. `arr[N]`) mit Hilfe einer zuvor definierten Konstante/Variable deklariert werden?
- Was wird hier ausgegeben?

```
int k = 0, arr1[] = {3, 4, 5}, arr2[3];
arr2[0] = 3;
arr2[1] = 4;
arr2[2] = 5;
printf("%s\n", arr1 == arr2 ? "true" : "false");
arr1[++k] = 7;
++arr2[k++];
printf("%d %d\n", arr1[1], arr2[1]);
```

# Arrays als Funktionsparameter

```
int sumOfArray(int values[], int numValues);
```

- Die aufgerufene Funktion kennt die Länge des übergebenen Arrays nicht, da Arrays ihre Länge ja selbst nicht kennen. Die Arraylänge muss daher separat als weiterer Parameter übergeben werden.
- Arrayinhalte werden beim Funktionsaufruf **nicht** kopiert!
- Lediglich der Wert der Arrayvariable (also die Adresse des ersten Arrayelements!) wird beim Aufruf an die Funktion übergeben.
- Verändert man also ein Array innerhalb einer Funktion, dann ist auch das übergebene Array in der aufrufenden Funktion verändert, weil die Arrayvariable jeweils auf die gleiche Start-Speicherstelle zeigt.

# Beispiel: Tic Tac Toe

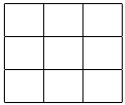
| | | |
|---|---|---|
| | o | |
| o | x | x |

- Spielfeld der Größe 3x3
- Zwei Spieler, die abwechselnd Steine setzen
- Wer zuerst drei Steine in einer Reihe hat, gewinnt
- Jede Reihe könnte man als Array mit Länge 3 anlegen
- Spielfeld könnte man auch als Array betrachten, das wiederum 3 Arrays enthält

# Mehrdimensionale Arrays

- Für jede weitere Array-Dimension gibt es ein eigenes Klammerpaar.
- Realisierung des Spielfelds mit zweidimensionalem Array:
  - `char spielfeld[3][3];`
  - Wird das Array direkt mit Mengenschreibweise initialisiert, dann muss die äußere Dimension nicht angegeben werden, da sie vom C-Compiler berechenbar ist.

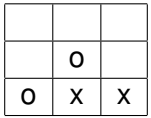
```
char spielfeld[][3] = {
      {' ', ' ', ' '},
      {' ', ' ', ' '},
      {' ', ' ', ' '}
    }; // Init. mit Leerzeichen
```



# Zweidimensionales Array

- Angenommen wir haben den aktuellen Spielstand rechts: Wie könnte Spieler o gewinnen?

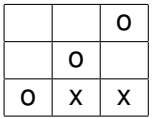
```
spielfeld[0][2] = 'o';
```




- Zuerst wird die Zeile angegeben und dann die Spalte, d.h. das Array wird zeilenweise linear im Speicher abgespeichert.

- Ausgabe des Spielfeldes mit verschachtelter Schleife:

```
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 3; j++) {
    printf(" %c ", spielfeld[i][j]);
  }
  printf("\n");
}
```



# Grobentwurf Tic Tac Toe

- Spielfeld: Zweidimensionales Array, drei mögliche Werte pro Element: `' '`, `'x'`, `'o'`
- Eingabe des Zugs: Setzen des Steins
- Prüfen, ob dies ein gültiger Zug ist, falls nein: neue Eingabe
- Prüfen, ob das Spiel gewonnen wurde: Falls ja, Spiel zu Ende, falls nein, Spielerwechsel!

# Zeichenkette (String)

- In C ist ein String kein eigener Datentyp, sondern ein Character-Array
- Lässt sich aber bequem als sog. String-Literal initialisieren:
  Beispiel: `char str[] = "Hello World";`
  Terminiert mit einem String-Endzeichen `'\0'`, das wird bei einem String-Literal aber automatisch angefügt.
- Bei Vereinbarung wird Speicherplatz passender Länge reserviert.
- Belegt (wie das Array auch) einen Speicherblock, d.h. eine Sequenz von Adressen.
  Der Bezeichner (im Beispiel `str`) ist von Natur aus ein (konstanter) Zeiger.
- Kann mit `printf()`/`scanf()` ausgegeben bzw. eingelesen werden (mittels: `%s`).
  Der Variable wird beim Einlesen kein `&` vorangestellt, da der Variablenwert schon eine Adresse ist!

# C-Strings sind Arrays

- String ist in C ein Array von Charactern
  - Beispiel: `char str[] = "Hello";`
  - Bzw: `char str[] = {'H', 'e', 'l', 'l', 'o', '\0'};`
  - Beides gleichbedeutend, Strings werden immer mit `'\0'` terminiert!
  - Das Feld `str[]` besteht immer aus 6 Zeichen, die Länge des Strings selbst ist aber nur 5 Zeichen.

- Bearbeitung und Ausgabe von Strings

```
str[0] = 'H';
str[1] = 'i';
printf("%s\n", str);

str[2] = '\0';
printf("%s", str);
```

# Strings modifizieren

- Einlesen von Zeichenketten

```
int tag, jahr;
char monat[10];
scanf("%d %s %d", &tag, monat, &jahr);
```

- Zuweisung von String-Literal nur bei Initialisierung einer Variablen möglich.
- Wie sieht hier der Speicherinhalt aus?

```
char msg[10] = "Bonjour";
```

# Strings kopieren

- Seien `s` und `t` Stringvariablen:
  - Dann kann man Zeichenkette `t` nicht zuweisen, bzw. kopieren mit `s = t;`
  - Grund: `s` und `t` sind Felder, welche in C unveränderbar die Anfangsadresse des Feldes beinhalten, d.h. Felder sind konstante Zeiger auf den reservierten Speicherbereich
- Implementierungsvorschlag:

```
int i = 0;
// Klammerung bei Zuweisung wichtig!
while ((s[i] = t[i]) != '\0') {
  i++;
}
```

- Achtung: Hier wird leider nirgends überprüft, ob die Feldlänge überschritten wird.

# String-Operationen

- C kennt keine speziellen Operationen auf ganzen Zeichenketten. Es gibt nur Operationen auf einzelnen Zeichen bzw. auf ein Array von Zeichen.
- Strings werden meistens mit Hilfe von Bibliotheksfunktionen bearbeitet:
  `#include <string.h>`
  - `strcpy(s, t)`: String `t` wird in String `s` kopiert, inklusive `'\0'`. Der neue String `s` wird zurückgegeben.
  - `strncpy(s, t, n)`: `n` Zeichen aus `t` werden in String `s` kopiert. Wenn `t` weniger Zeichen hat als `n`, dann wird mit `'\0'` aufgefüllt.

# String-Operationen

- Funktion `strcmp(s, t)` vergleicht die Zeichenkette `s` mit der Zeichenkette `t`
  - Liefert < 0, wenn s kleiner ist, liefert 0 wenn beide Strings gleich sind, sonst > 0
  - `strcmp("A", "A")` ist 0
  - `strcmp("A", "B")` ist -1
  - `strcmp("B", "A")` ist 1
  - `strcmp("Z", "a")` ist -1
- Funktion `strncmp(s, t, n)` vergleicht die ersten `n` Zeichen von String `s` mit den ersten `n` Zeichen von String `t`. Ergebnis analog zu `strcmp`
- Stringlänge ermitteln mit `strlen(str)`
  - Liefert die Länge von `str` (ohne `'\0'`)
  - `char str[8] = "Hello";` → Welchen Wert liefert `strlen(str)` hier?