Arrays (Felder/Vektoren)

- Zusammenfassung mehrerer Variablen des gleichen Typs unter einem Namen
- im Speicher direkt hintereinander abgelegt
- **Definition:** *Typname Arrayname* `[`*Größe*`];`

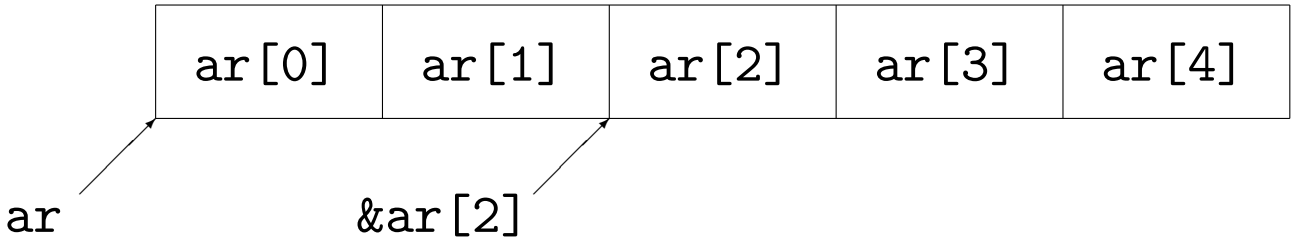
  Beispiel: `int ar [5];`
- **Zugriff** auf das i-te Array-Element: *Arrayname*`[i]`
- Indizierung beginnt bei `0`

```
for (i = 0; i < 5; i++)
  ar[i] = i + 1;
```

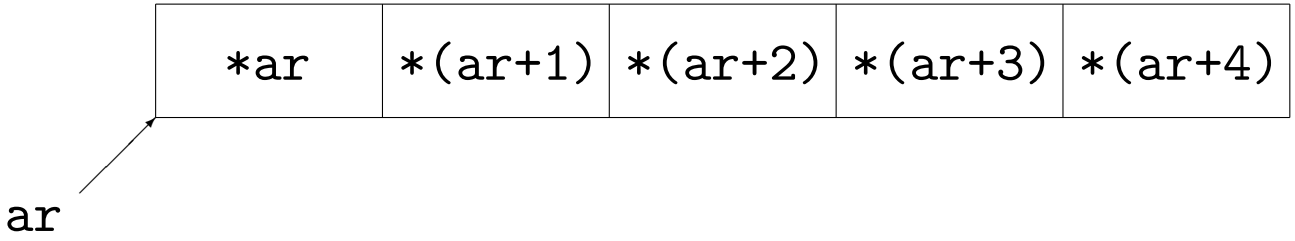
$\rightsquigarrow$ `ar[0] = 1, ar[1] = 2, ar[2] = 3, ar[3] = 4 , ar[4] = 5`

Arrays und Pointer

- Name des Arrays ist konstanter Zeiger auf das erste Array-Element



- `ar[i]` ist äquivalent zu `*(ar+i)`



- Sei `ptr` ein Pointer. Dann ist `*(ptr + i)` äquivalent zu `ptr[i]`.

Mehrdimensionale Arrays

- Verwendung mehrfacher eckiger Klammern
- Elemente sind selbst Arrays (eine Dimension niedriger)
- Beispiel: `int ar [3][4];`

`ar[0][0]`	`ar[0][1]`	`ar[0][2]`	`ar[0][3]`
`ar[1][0]`	`ar[1][1]`	`ar[1][2]`	`ar[1][3]`
`ar[2][0]`	`ar[2][1]`	`ar[2][2]`	`ar[2][3]`

Initialisierungslisten

- Elemente in geschweiften Klammern, durch Komma getrennt
- nur direkt bei der Definition

  $\rightsquigarrow$ `int ar [4] = {1, 2, 3, 4};`
- Größe in eckigen Klammern darf fehlen (*offenes Array*)

  `int ar [] = {1, 2, 3, 4};`
- fehlende Elemente werden mit `0` aufgefüllt:

  `int ar [4] = {1, 2};` $\rightsquigarrow$ `ar[0]=1; ar[1]=2; ar[2]=0; ar[3]=0;`
- *mehrdimensional*: Initialisierungsliste von Initialisierungslisten

Zeichenketten (Strings)

- sind `char`-Arrays mit **Nullzeichen** `'\0'` als Markierung des Stringendes
- Viele String-Funktionen benötigen das Nullzeichen.
  $\rightsquigarrow$ bei Definition des Arrays einplanen!

  ```
  char vorname [6] = {'N', 'a', 'd', 'j', 'a', '\0'};
  ```

- Initialisierung mit konstanter Zeichenkette:

  ```
  char vorname [6] = "Nadja";
  ```

  $\rightsquigarrow$ automatisches Anfügen des Nullzeichens
- konstante Zeichenketten sind ein *Ausdruck*:
  Rückgabewert ist Pointer auf das erste Zeichen

Übergabe und Ausgabe von Arrays und Zeichenketten

- **Übergabe** eindimensionaler Arrays an Funktionen:
  formaler Parameter als
  - offenes Array oder
  - Pointer auf den Komponententyp

- Anwendung bei Übergabe von Zeichenketten (`char`-Array)

- **Ausgabe** von Zeichenketten mit `printf()`:
  `printf()` erhält Pointer auf erstes Zeichen
  $\rightsquigarrow$ Ausgabe aller Zeichen bis Pointer auf `'\0'` zeigt

- Übergabe einer Stringvariable (Pointer auf `char`-Array) an `printf()`:
  Formatelement `%s`

- Beispiel:
```
char * pointer = "hello";
printf("\n%s", pointer);
```

- `char`-Array:
  - gespeicherter String kann verändert werden
  - Array-Name ist *konstanter* Pointer auf das erste Zeichen

- Stringvariable vom Typ `char *` auf *konstante Zeichenkette*:
  - Compiler speichert *konstante Zeichenkette* selbstständig
  - Zeichenkette kann nicht verändert werden
  - Pointer kann neue Adresse zugewiesen bekommen

Standardfunktionen zur Stringverarbeitung

- in Header-Datei `<string.h>` definiert

`char * strcpy (char * dest, const char * src);`	Kopieren
`char * strcat (char * dest, const char * src);`	Anhängen
`int strcmp (const char * s1, const char * s2);`	Vergleichen
`size_t strlen (const char * s);`	Länge

- Arbeiten bis zum Nullzeichen `'\0'`

- `size_t` vordefinierter Datentyp als Rückgabetyp des `sizeof`-Operator (ist meist `unsigned int` oder `unsigned long`)

- `const char *` $Name \rightsquigarrow$ Array-Elemente konstant
  `char * const` $Name \rightsquigarrow$ Pointer konstant

- `mem`-Funktionen (`memcpy()`, `memcmp()` usw.)
  - arbeiten mit Paramentern vom Typ `void *` statt `char *`
  - byteweise Bearbeitung beliebeger Speicherobjekte
  - erwarten kein Nullzeichen; Anzahl der Bytes als weiterer Parameter

  Beispiel:

```
void * memcpy (void * dest, void * src, size_t n);
```

Standardfunktionen zur Ein- und Ausgabe

- definiert in `<stdio.h>`
- `int printf (const char * format, ...);`
- `int puts (const char * s);`
  - schreibt übergebenen String `s` nach `stdout`
  - kopiert das Nullzeichen *nicht* mit
- `char * gets (char * s);`
  - liest von `stdin` (zeilengepuffert) in `char`-Array `s`
  - hängt das Nullzeichen an
  - übergibt Pointer `s`

- `int scanf (const char * format, ...);`
  - Argumente nach dem Formatstring sind **Adressen von Variablen**,
  - Speichern der Werte aus `stdin` auf diesen Adressen
  - Anzahl und Typen der Formatelemente müssen zu den adressierten Variablen passen (sonst unbestimmtes Verhalten)

- Beispiel:

```
int zahl;

printf ("\nEingabe: ");
scanf("%d", &zahl);

printf ("\nDer Wert %d wurde eingelesen.\n", zahl);
```

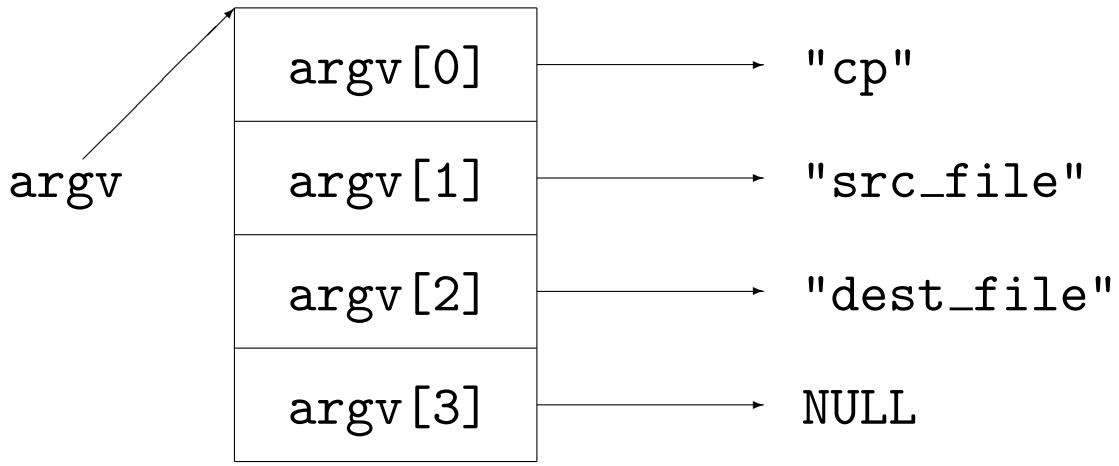
- andere Zeichen als Formatelemente im Formatstring (auch `'\n'`):
  - `scanf()` liest diese Zeichen und verwirft sie
  - liest `scanf()` ein anderes Zeichen, wird es in `stdin` zurückgestellt

```
float t;
printf("Temperatur im Format xx C: ");
scanf("%f C", &t);
t = (9. * t) / 5. + 32.;
printf("\nTemperatur in Fahrenheit: %f F", t);
```

  $\rightsquigarrow$ Kein Newline-Zeichen `'\n'` im Formatstring von `scanf()`!

Parameterübergabe beim Programmaufruf

- `cp src_file dest_file`
- zwei Varianten der `main`-Funktion:
  - `int main()` — parameterlos
  - `int main(int argc, char * argv[])` — zwei Parameter
- `argc` (**arg**ument **c**ounter): Anzahl der Argumente
- `argv` (**arg**ument **v**ector): Vektor (Array) der Argumente
  Argumente sind Strings $\rightsquigarrow$ Array von `char`-Arrays
  $\rightsquigarrow$ Array von Pointern auf `char`
- erstes Element von `argv` (`argv[0]`): Programmname



- Übergabe von Zahlen: Typumwandlung String $\longrightarrow$ Zahltyp erforderlich
- Standardfunktionen in `<stdlib.h>`

```
double atof(const char * nptr);   ascii to float
int atoi(const char * nptr);      ascii to int
int atol(const char * nptr);      ascii to long
```