Systemnahe Programmierung in C (SPiC)

16 Ergänzungen zur Einführung in C

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2019

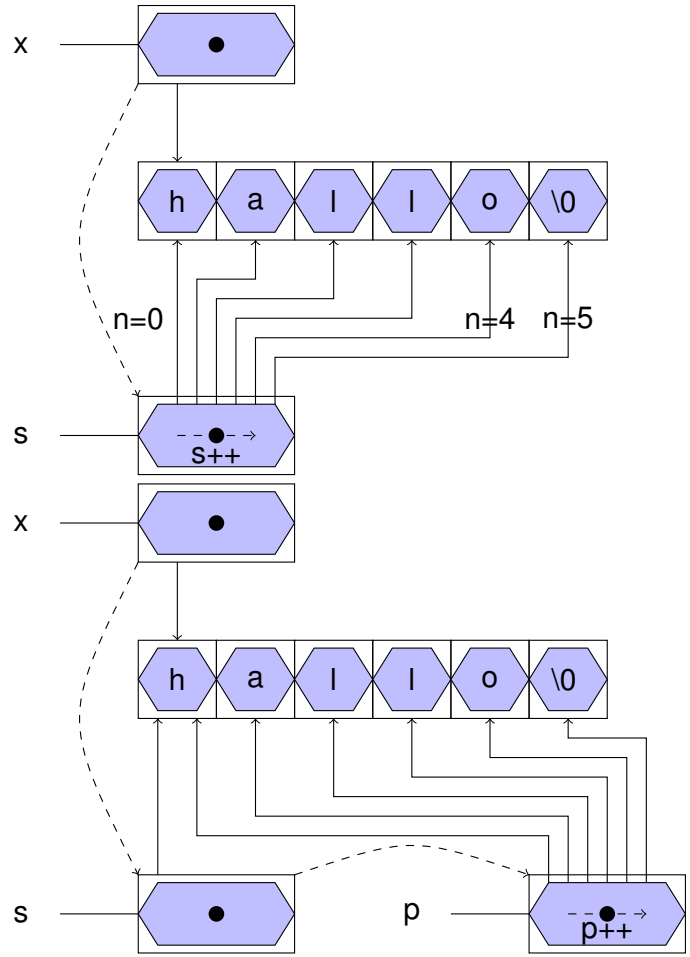
http://www4.cs.fau.de/Lehre/SS19/V_SPIC

Zeiger, Felder und Zeichenketten

- Zeichenketten sind Felder von Einzelzeichen (`char`), die in der internen Darstellung durch ein `'\0'`-Zeichen abgeschlossen sind
- Beispiel: Länge eines Strings ermitteln – Aufruf `strlen(x);`

```c
/* 1. Version */
int strlen(const char *s)
{
    int n;
    for (n = 0; *s != '\0'; n++) {
        s++;
    }
    return n;
}
```

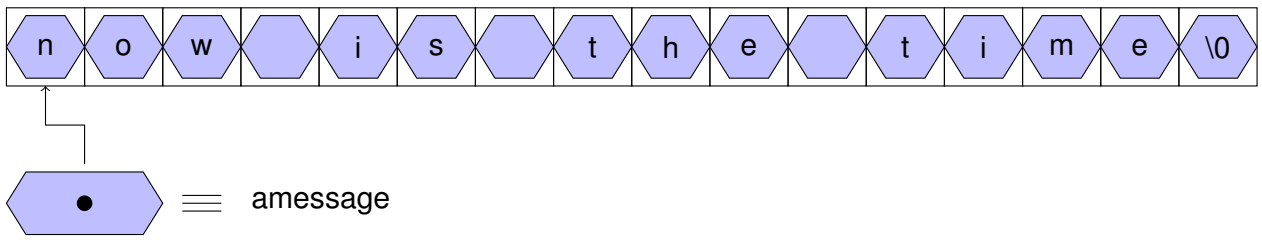
```c
/* 2. Version */
int strlen(const char *s)
{
    const char *p = s;
    while (*p != '\0') {
        p++;
    }
    return p - s;
}
```



Zeiger, Felder und Zeichenketten (Forts.)

- wird eine Zeichenkette zur Initialisierung eines `char`-Feldes verwendet, ist der Feldname ein konstanter Zeiger auf den Anfang der Zeichenkette

```
char amessage[] = "now is the time";
```



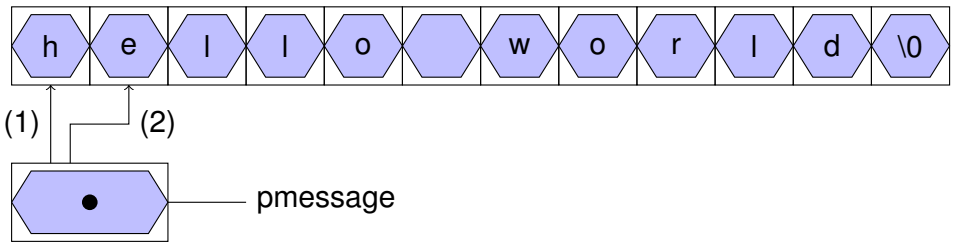
- es wird ein Speicherbereich für 16 Bytes reserviert und die Zeichen werden in diesen Speicherbereich hineinkopiert
- `amessage` ist ein konstanter Zeiger auf den Anfang des Speicherbereichs und kann nicht verändert werden
- der Inhalt des Speicherbereichs kann aber modifiziert werden

```
amessage[0] = 'h';
```


Zeiger, Felder und Zeichenketten (Forts.)

- wird eine Zeichenkette zur Initialisierung eines `char`-Zeigers verwendet, ist der Zeiger eine
  Variable, die mit der Anfangsadresse der Zeichenkette initialisiert wird

```
const char *pmessage = "hello world";    /*(1)*/
```

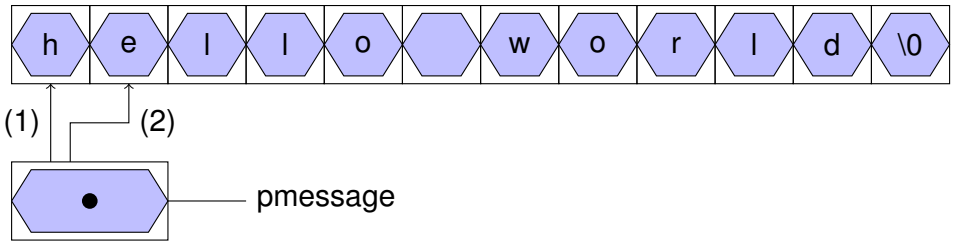



```
pmessage++;    /*(2)*/
printf("%s\n", pmessage); /* gibt "ello world" aus */
```

- die Zeichenkette selbst wird vom Compiler als konstanter Wert (String-Literal) im Speicher angelegt
- es wird ein Speicherbereich für einen Zeiger reserviert (z.B. 4 Byte) und mit der Adresse der Zeichenkette initialisiert

Zeiger, Felder und Zeichenketten (4)

```
const char *pmessage = "hello world";    /*(1)*/
```




```
pmessage++;    /*(2)*/
printf("%s\n", pmessage); /* gibt "ello world" aus */
```

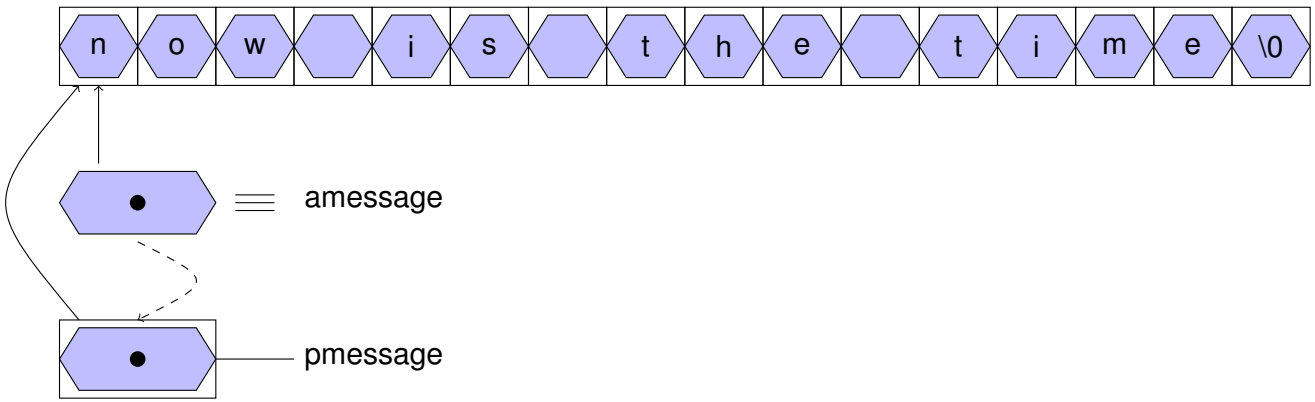
- `pmessage` ist ein variabler Zeiger, der mit dieser Adresse initialisiert wird, aber jederzeit verändert werden darf (`pmessage++;`)
- der Speicherbereich von `"hello world"` darf aber nicht verändert werden
  - der Compiler erkennt dies durch das Schlüsselwort `const` und verhindert schreibenden Zugriff über den Zeiger
  - manche Compiler legen solche Zeichenketten ausserdem im schreibgeschützten Speicher an (=> Speicherschutzverletzung beim Zugriff, falls der Zeiger nicht als `const`-Zeiger definiert wurde)

Zeiger, Felder und Zeichenketten (5)

- die Zuweisung eines `char`-Zeigers oder einer Zeichenkette an einen `char`-Zeiger bewirkt kein Kopieren von Zeichenketten!

```
pmessage = amessage;
```

weist dem Zeiger `pmessage` lediglich die Adresse der Zeichenkette `"now is the time"` zu



- wird eine Zeichenkette als aktueller Parameter an eine Funktion übergeben, erhält diese eine Kopie des Zeigers

Zeiger, Felder und Zeichenketten (6)

- Um eine ganze Zeichenkette einem anderen char-Feld zuzuweisen, muss sie kopiert werden: Funktion `strcpy` in der Standard-C-Bibliothek
- Implementierungsbeispiele:

```
/* 1. Version */
void strcpy(char s[], char t[]) {
    int i = 0;
    while ((s[i] = t[i]) != '\0') {
        i++;
    }
}
```

```
/* 2. Version */
void strcpy(char *s, char *t) {
    while ((*s = *t) != '\0') {
        s++, t++;
    }
}
```

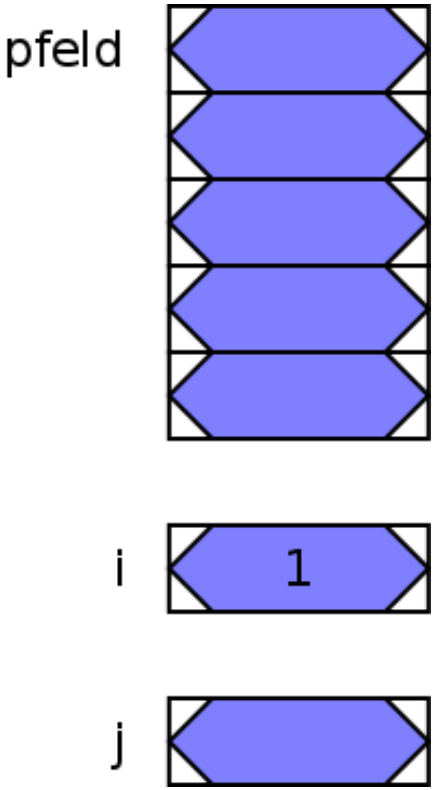
```
/* 3. Version */
void strcpy(char *s, char *t) {
    while (*s++ = *t++) {
    }
}
```

Felder von Zeigern

Auch von Zeigern können Felder gebildet werden

- Deklaration

```
int *pfeld[5];
int i = 1;
int j;
```



Felder von Zeigern (Forts.)

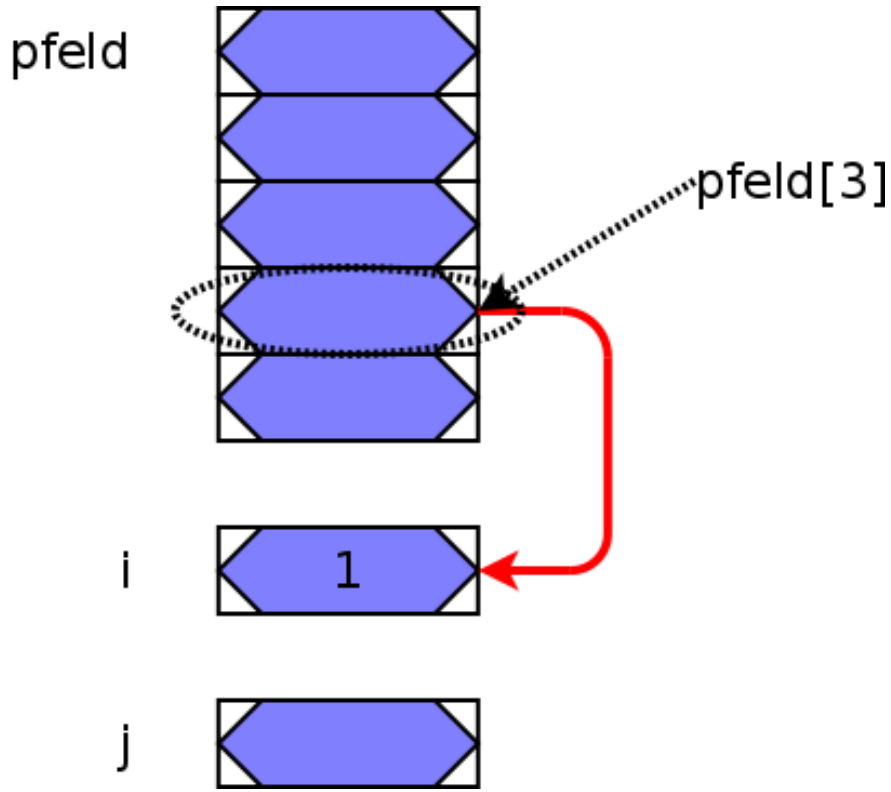
Auch von Zeigern können Felder gebildet werden

- Deklaration

```
int *pfeld[5];
int i = 1;
int j;
```

- Zugriff auf einen Zeiger des Feldes

```
pfeld[3] = &i;
```



Felder von Zeigern (Forts.)

Auch von Zeigern können Felder gebildet werden

- Deklaration

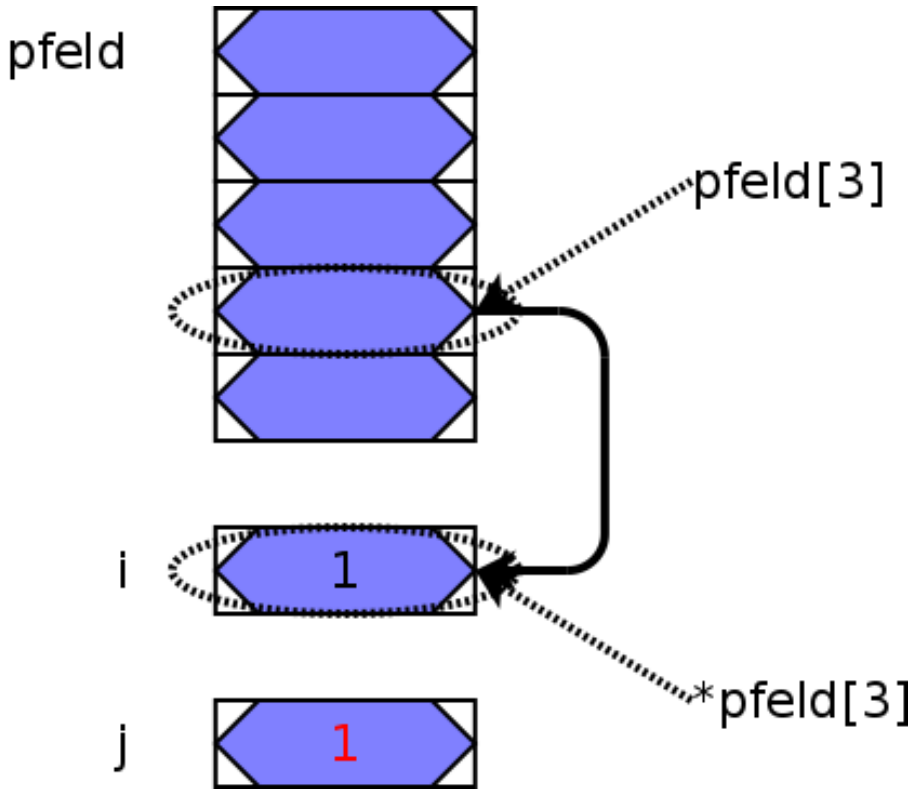
```
int *pfeld[5];
int i = 1;
int j;
```

- Zugriff auf einen Zeiger des Feldes

```
pfeld[3] = &i;
```

- Zugriff auf das Objekt, auf das ein Zeiger des Feldes verweist

```
j = *pfeld[3];
```

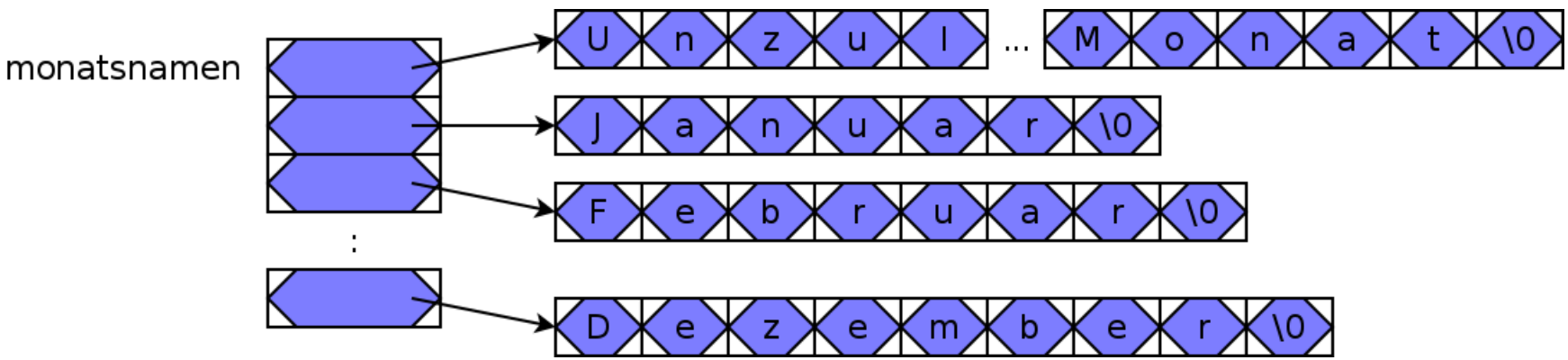


Felder von Zeigern (Forts.)

Beispiel: Definition und Initialisierung eines Zeigerfeldes:

```
const char *
month_name(int n)
{
    static const char *monatsname[] = {
        "Unzulaessiger Monat",
        "Januar",
        ...
        "Dezember"
    };

    return (n < 1 || 12 < n) ? monatsname[0] : monatsname[n];
}
```



Argumente aus der Kommandozeile

- beim Aufruf eines Programms können normalerweise Argumente übergeben werden
- der Zugriff auf diese Argumente wird der Funktion `main()` durch zwei Aufrufparameter ermöglicht (beide Varianten gleichwertig):

```
int
main(int argc, char *argv[])
{
    ...
}
```

```
int
main(int argc, char **argv)
{
    ...
}
```

- der Parameter `argc` enthält die Anzahl der Argumente, mit denen das Programm aufgerufen wurde
- der Parameter `argv` ist ein Feld von Zeigern auf die einzelnen Argumente (Zeichenketten)
- der Programmname wird als erstes Argument übergeben (`argv[0]`)

Argumente aus der Kommandozeile

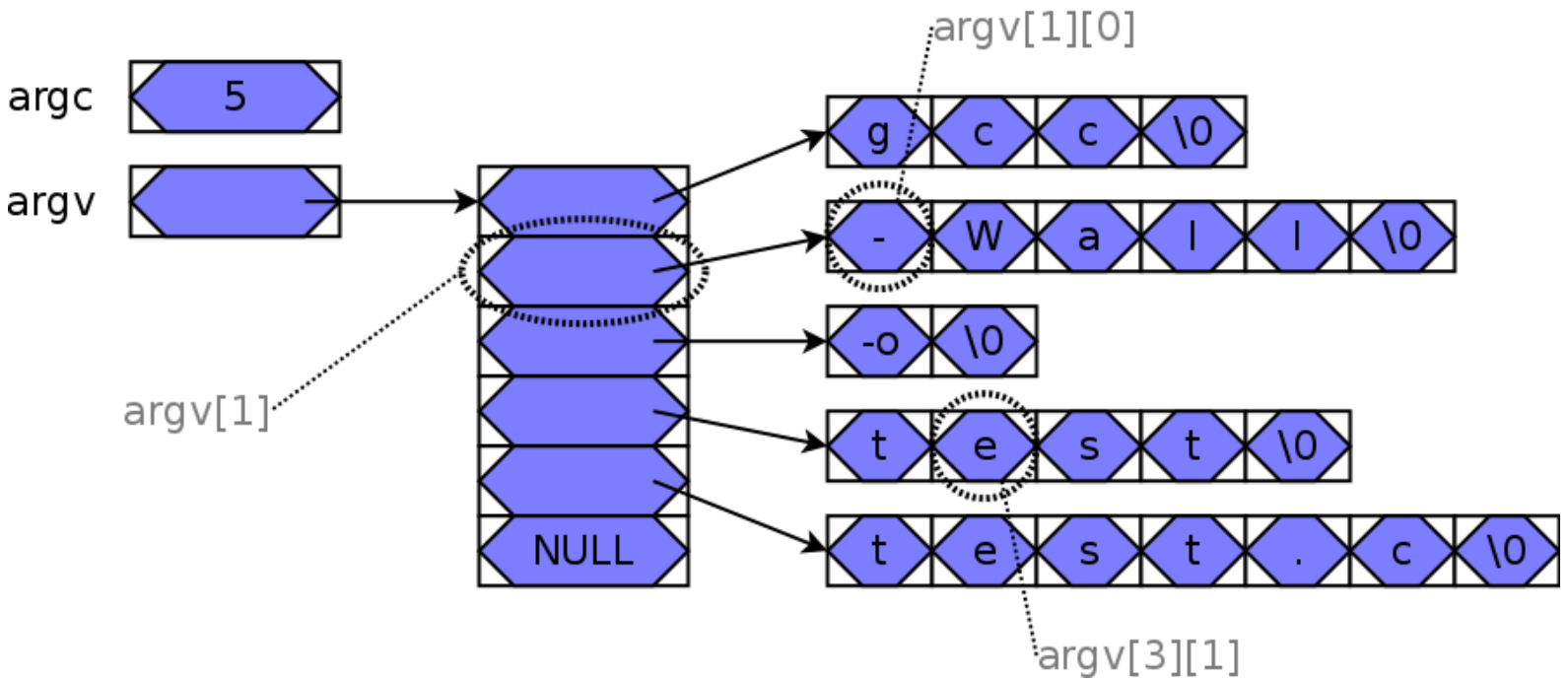
- Kommando:

```
gcc -Wall -o test test.c
```

- C-Datei:

```
...
int main(int argc, char *argv[])
...
```

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



Argumente – Beispiel

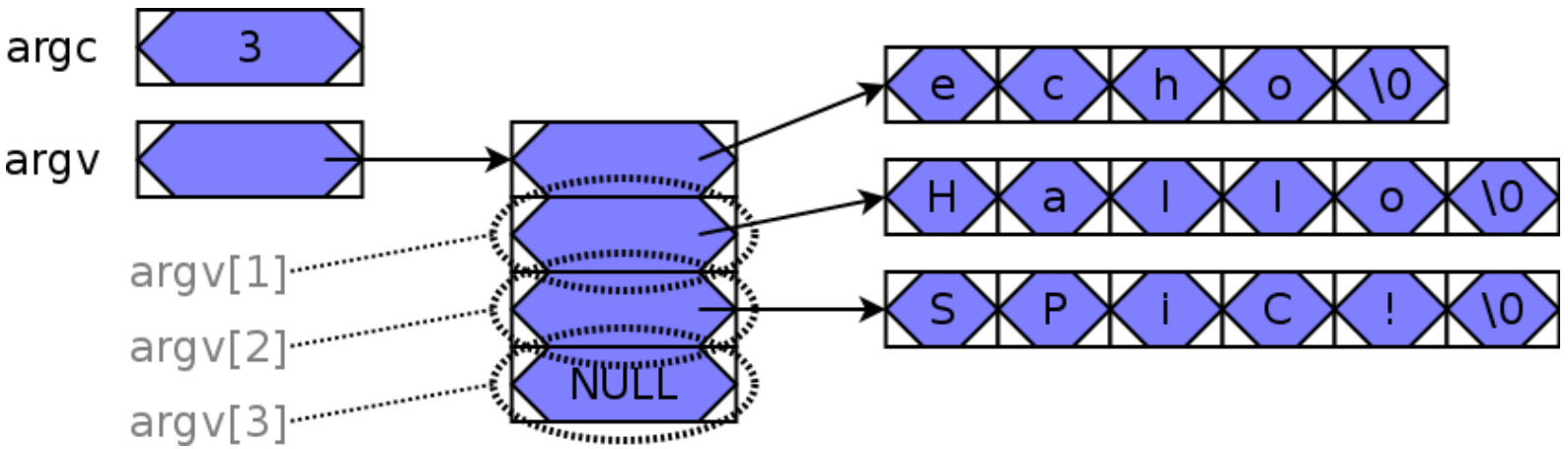
Beispiel: **echo**-Programm

```
~> echo Hallo SPiC!
Hallo SPiC!
~>
```

```c
#include <stdio.h>

int
main(int argc, char *argv[])
{
    for (int i = 1; i < argc; i++) {
        printf("%s ", argv[i]);
    }
    printf("\n");

    return 0;
}
```



Verbund-Datentypen / Strukturen

- Zusammenfassen mehrerer Daten zu einer Einheit
- Struktur-Deklaration

```
struct person {
    char name[20];
    int age;
};
```

- Definition einer Variablen vom Typ der Struktur

```
struct person p1;
```

- Zugriff auf ein Element der Struktur

```
strcpy(p1.name, "Peter Pan");
p1.age = 12;
```


Zeiger auf Strukturen

- Konzept analog zu „Zeiger auf Variable”
  - Adresse einer Struktur mit &-Operator zu bestimmen
- Beispiel

```
struct person stud1;
struct person *pstud;
pstud = &stud1;
```

- Besondere Bedeutung beim Aufbau verketteter Strukturen (Listen, Bäume, ...)
  - eine Struktur kann Adressen weiterer Strukturen desselben Typs enthalten

Zeiger auf Strukturen (Forts.)

- Zugriff auf Strukturkomponenten über Zeiger
- bekannte Vorgehensweise
  - „*”-Operator liefert die Struktur
  - „.”-Operator liefert ein Element der Struktur
  - **Aber:** Operatorenvorrang beachten!

```
(*pstud).age = 21;
```

- syntaktische Verschönerung
  - „->”-Operator

```
pstud->age = 21;
```


Verschachtelte/verkettete Strukturen

- Strukturen in Strukturen sind erlaubt – aber:
  - die Größe einer Struktur muss vom Compiler ausgerechnet werden können
    => Struktur kann sich nicht selbst enthalten
  - die Größe eines Zeigers ist bekannt
    => Struktur kann Zeiger auf gleiche Struktur enthalten
  - Beispiele:

Verkettete Liste:

```
struct list {
    struct list *next;
    struct person stud;
};

struct list *head;
```

Baum:

```
struct tree {
    struct tree *left;
    struct tree *right;
    struct person stud;
};

struct tree *root;
```

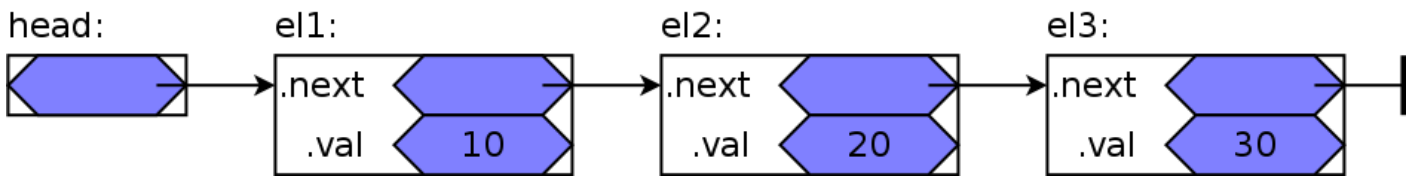

Verkettete Listen

- Mehrere Strukturen desselben Typs werden über Zeiger miteinander verkettet

```
struct list { struct list *next; int val; };

struct list el1, el2, el3;
struct list *head;

head = &el1;
el1.next = &el2; el2.next = &el3; el3.next = NULL;
el1.val = 10;    el2.val = 20;    el3.val = 30;
```



- Laufen über eine verkettete Liste

```
int sum = 0;
for (struct list *curr = head; curr != NULL; curr = curr->next) {
    sum += curr->val;
}
```


Ein-/Ausgabe

- E/A-Funktionalität nicht Teil der Programmiersprache
- Realisierung durch „normale” Funktionen
  - Bestandteil der Standard-Bibliothek
  - einfache Programmierschnittstelle
  - effizient
  - portabel
  - betriebssystem-nah
- Funktionsumfang
  - Öffnen/Schließen von Dateien
  - Lesen/Schreiben von Zeichen, Zeilen oder beliebigen Datenblöcken
  - formatierte Ein-/Ausgabe

Standard-Ein-/Ausgabe

Jedes C-Programm erhält beim Start automatisch 3 E/A-Kanäle:

`stdin`: Standard-Eingabe

- normalerweise mit der Tastatur verbunden
- „Dateiende" (`EOF`) wird durch Eingabe von `CTRL-D` am Zeilenanfang signalisiert
- bei Programmaufruf in der Shell auf Datei umlenkbar

```
~> prog < eingabedatei
```

`stdout`: Standard-Ausgabe

- normalerweise mit Bildschirm (bzw. dem Fenster in dem das Programm gestartet wurde) verbunden
- bei Programmaufruf in der Shell auf Datei umlenkbar

```
~> prog > ausgabedatei
```

`stderr`: Ausgabekanal für Fehlermeldungen

- normalerweise ebenfalls mit Bildschirm verbunden

Standard-Ein-/Ausgabe (Forts.)

- Pipes
  - Die Standardausgabe eines Programmes kann mit der Standardeingabe eines anderen Programms verbunden werden:

    ```
    ~> prog1 | prog2
    ```

    Die Umlenkung von Standard-E/A-Kanälen ist für die aufgerufenen Programme weitgehend unsichtbar.
- automatische Pufferung
  - Eingaben von der Tastatur werden normalerweise vom Betriebssystem zeilenweise zwischengespeichert und erst bei einem `NEWLINE`-Zeichen (`'\n'`) an das Programm übergeben!
  - Ausgaben an den Bildschirm werden vom Programm normalerweise zeilenweise zwischengespeichert und erst beim `NEWLINE`-Zeichen wirklich auf den Bildschirm geschrieben!

Öffnen und Schließen von Dateien

- Neben den Standard-E/A-Kanälen kann ein Programm selbst weitere E/A-Kanäle öffnen
  - Zugriff auf Dateien
- Öffnen eines E/A-Kanals
  - Funktion `fopen` (File Open)
- Schließen eines E/A-Kanals
  - Funktion `fclose` (File Close)

Öffnen und Schließen von Dateien (Forts.)

- Schnittstelle `fopen`

```
#include <stdio.h>

FILE *fopen(const char *name, const char *mode);
```

`name`: Pfadname der zu öffnenden Datei

`mode`: Art, wie Datei zu öffnen ist

- `"r"`: zum Lesen (read)
- `"w"`: zum Schreiben (write)
- `"a"`: zum Schreiben am Dateiende (append)
- `"rw"`: zum Lesen und Schreiben (read/write)

- öffnet Datei `name`
- Ergebnis von `fopen`: Zeiger auf einen Datentyp `FILE`, der einen Dateikanal beschreibt; im Fehlerfall `NULL`

Öffnen und Schließen von Dateien (Forts.)

- Schnittstelle `fclose`

```
#include <stdio.h>

int fclose(FILE *fp);
```

  - schließt E/A-Kanal `fp`
  - Ergebnis ist entweder `0` (kein Fehler aufgetreten) oder `EOF` im Falle eines Fehlers

Öffnen und Schließen von Dateien – Beispiel

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

int main(void)
{
    FILE *fp; int ret;

    fp = fopen("test.dat", "w"); /* Open "test.dat" for writing. */
    if (fp == NULL) {
        /* Error */
        perror("test.dat"); /* Print error message. */
        exit(EXIT_FAILURE); /* Terminate program. */
    }

    ... /* Program can now write to file "test.dat". */

    ret = fclose(fp); /* Close file. */
    if (ret == EOF) {
        /* Error */
        perror("test.dat"); /* Print error message. */
        exit(EXIT_FAILURE); /* Terminate program. */
    }

    return EXIT_SUCCESS;
}
```

Zeichenweises Lesen und Schreiben

- Lesen eines einzelnen Zeichens
  - von der Standardeingabe

```
#include <stdio.h>
int getchar(void);
```

  - aus einer Datei

```
#include <stdio.h>
int fgetc(FILE *fp);
```

  - lesen das nächste Zeichen
  - geben das Zeichen als `int`-Wert zurück
  - geben bei Eingabe von CTRL-D bzw. am Ende der Datei `EOF` als Ergebnis zurück
- Schreiben eines einzelnen Zeichens
  - auf die Standardausgabe

```
#include <stdio.h>
int putchar(int c);
```

  - in eine Datei

```
#include <stdio.h>
int fputc(int c, FILE *fp);
```

  - schreiben das Zeichen `c`
  - geben im Fehlerfall `EOF` als Ergebnis zurück

Zeichenweises Lesen und Schreiben – Beispiel

Kopierprogramm:

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

int main(int argc, char *argv[]) {
    FILE *src, *dst;
    int c;

    if (argc != 3) { ... }

    if ((src = fopen(argv[1], "r")) == NULL) { ... }
    if ((dst = fopen(argv[2], "w")) == NULL) { ... }

    while ((c = fgetc(src)) != EOF) {
        if (fputc(c, dst) == EOF) { ... }
    }

    if (fclose(dst) == EOF) { ... }
    if (fclose(src) == EOF) { ... }

    return EXIT_SUCCESS;
}
```


Zeilenweises Lesen und Schreiben

- Lesen einer Zeile

```
#include <stdio.h>
char *fgets(char *buf, int bufsize, FILE *fp);
```

  - liest Zeichen aus Dateikanal `fp` in das `char`-Feld `s` bis entweder `bufsize-1` Zeichen gelesen wurden oder `'\n'` oder `EOF` gelesen wurde
  - `s` wird mit `'\0'` abgeschlossen (`'\n'` wird nicht entfernt)
  - gibt bei `EOF` oder Fehler `NULL` zurück
  - für `fp` kann `stdin` eingesetzt werden, um von der Standardeingabe zu lesen

- Schreiben einer Zeile

```
#include <stdio.h>
int fputs(char *buf, FILE *fp);
```

  - schreibt die Zeichen im Feld `s` auf Dateikanal `fp`
  - gibt im Fehlerfall `EOF` zurück
  - für `fp` kann auch `stdout` oder `stderr` eingesetzt werden

Formatierte Ausgabe

- Schnittstelle

```
#include <stdio.h>
int printf(char *format, ...);
int fprintf(FILE *fp, char *format, ...);
int sprintf(char *buf, char *format, ...);
int snprintf(char *buf, int bufsize, char *format, ...);
```

- Die statt `...` angegebenen Parameter werden entsprechend der Angaben im `format`-String ausgegeben
  - bei `printf` auf der Standardausgabe
  - bei `fprintf` auf dem Dateikanal `fp`
    (für `fp` kann auch `stdout` oder `stderr` eingesetzt werden)
  - `sprintf` schreibt die Ausgabe in das `char`-Feld `buf`
    (achtet dabei aber nicht auf das Feldende => Pufferüberlauf möglich!)
  - `snprintf` arbeitet analog, schreibt aber nur maximal `bufsize` Zeichen
    (`bufsize` sollte natürlich nicht größer als die Feldgröße sein)

Formatierte Ausgabe (Forts.)

- Zeichen im `format`-String können verschiedene Bedeutung haben
  - normale Zeichen:
    werden einfach in die Ausgabe kopiert
  - Escape-Zeichen:
    z.B. `\n` oder `\t` werden durch die entsprechenden Zeichen (hier Zeilenvorschub bzw. Tabulator) bei der Ausgabe ersetzt
  - Format-Anweisungen:
    beginnen mit `%`-Zeichen und beschreiben, wie der dazugehörige Parameter in der Liste nach dem `format`-String aufbereitet werden soll
- für genauere Informationen siehe Manuals (`man 3 printf`, ...)

Formatierte Ausgabe (Forts.)

- Format-Anweisungen

%d, %i: `int`-Parameter als Dezimalzahl ausgeben

%ld, %li: entsprechend für `long int`

%f: `float`-Parameter als Fließkommazahl ausgeben (z.B. `13.153534`)

%lf: entsprechend für `double`

%e: `float`-Parameter als Fließkommazahl in 10er-Potenz-Schreibweise ausgeben (z.B. `2.71456e+02`)

%le: entsprechend für `double`

%c: `char`-Parameter als einzelnes Zeichen ausgeben

%s: `char`-Feld wird ausgegeben, bis `'\0'` erreicht ist

%%: ein `%`-Zeichen wird ausgegeben

...: ...

Formatierte Ausgabe – Beispiel

```
int tag = 25;
int monat = 6;
int jahr = 2009;
char *name = "Michael Jackson";
printf("Am %d.%d.%d starb\n%s.\n",
       tag, monat, jahr, name);

printf("\n");

double pi = asin(1.0) * 2.0;
double e = exp(1.0);
fprintf(stdout,
        "Wichtige Werte sind:\n");
fprintf(stdout,
        "pi=%lf und e=%lf\n", pi, e);
```

```
~> ./test
Am 25.6.2009 starb
Michael Jackson.

Wichtige Werte sind:
pi=3.141593 und e=2.718282
~>
```

Formatierte Eingabe

- Schnittstelle

```
#include <stdio.h>

int scanf(char *format, ...);
int fscanf(FILE *fp, char *format, ...);
int sscanf(char *buf, char *format, ...);
```

Format-String analog zur formatierten Ausgabe.
Für genauere Informationen siehe Manuals (`man 3 scanf`, ...).

Aber: da Werte gelesen werden sollen, müssen Zeiger auf die zu beschreibenden Variablen übergeben werden!

Formatierte Eingabe – Beispiel

```c
double pi, e;
int ret;

ret = scanf("pi=%lf, e=%lf\n", &pi, &e);
if (ret != 2) {
    fprintf(stderr, "Bad input!\n");
    exit(EXIT_FAILURE);
}
printf("I got\n\tpi=%lf\n\te=%lf\n", pi, e);
```

```
~> ./test
3.14 2.718
Bad input!
~>
```

```
~> ./test
pi=3.14, e=2.718
I got
        pi=3.140000
        e=2.718000
~>
```


Fehlerbehandlung

- Fast jeder Systemaufruf/Bibliotheksaufruf kann fehlschlagen

  $=>$ **Fehlerbehandlung unumgänglich!**

- Ziel:

  Es darf kein Programm ohne Fehlermeldung abstürzen!

Fehlerbehandlung

- Vorgehensweise:
  - Rückgabewert von Systemaufruf/Bibliotheksaufruf abfragen
  - Im Fehlerfall (häufig durch Rückgabewert -1 oder `NULL` angezeigt): Fehlercode steht in globaler Variablen `errno`
- Fehlermeldung kann mit der Funktion `perror` auf die Fehlerausgabe ausgegeben werden:

```
#include <errno.h>
void perror(const char *s);
```

- Zwischenergebnisse auf Plausibilität überprüfen

```
#include <assert.h>
void assert(int condition);
```

Wenn Bedingung `condition` nicht „wahr” ist, wird das Programm mit Fehlermeldung abgebrochen.

Fehlerbehandlung (Forts.)

- Fehlerbehandlung dem Kontext anpassen; Beispiele
  - Fehler aufgrund von Benutzer-Fehlern
    (z.B. Benutzer gibt falschen Dateinamen oder falsche URL ein)
    - Benutzer auf Fehler hinweisen
    - Benutzer neue Eingabe ermöglichen
    - fehlgeschlagenen Programmteil widerholen
  - Fehler aufgrund fehlender Ressourcen
    (z.B. Speicher oder Platte voll)
    - Benutzer auf Fehler hinweisen
    - Benutzer Möglichkeit geben „aufzuräumen”
    - fehlgeschlagenen Programmteil widerholen
  - Programmierfehler
    (z.B. Zwischenergebnisse falsch)
    - Fehlermeldung ausgeben
    - Programm abbrechen
  - ...

Fehlerbehandlung – Beispiel

```
...

assert(argv[1] != NULL);

/* Open file for writing. */
FILE *fp = fopen(argv[1], "w");
if (fp == NULL) {
    perror(argv[1]);
    exit(EXIT_FAILURE);
}

/* Write to file. */
...

/* Close file. */
int ret = fclose(fp);
if (ret == EOF) {
    perror("fclose");
    exit(EXIT_FAILURE);
}

...
```

```
~> ./test
test.c:9: main: Assertion
          'argv[1] != NULL' failed.
~>
```

```
~> ./test /etc/shadow
/etc/shadow: Permission denied
~>
```

```
~> ./test hallo.txt
fclose: Quota exceeded
~>
```