

# Überblick: Teil D Betriebssystemabstraktionen

15 Nebenläufigkeit

**16 Ergänzungen zur Einführung in C**

**17 Betriebssysteme**

**18 Dateisysteme**

**19 Programme und Prozesse**

**20 Speicherorganisation**

**21 Nebenläufige Prozesse**

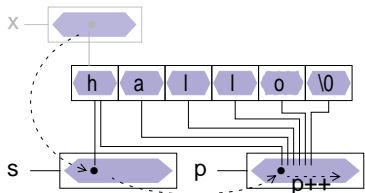
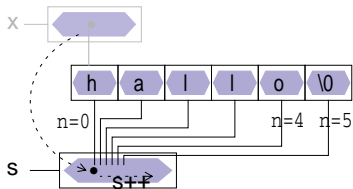


# Zeiger, Felder und Zeichenketten

- 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)**;

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

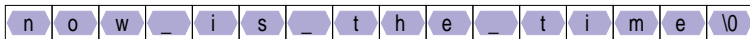
```
/* 2. Version */  
int strlen(const char *s)  
{  
    char *p = s;  
    while (*p != '\0')  
        p++;  
    return(p-s);  
}
```



## Zeiger, Felder und Zeichenketten (2)

- 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";
```



amessage  $\equiv$

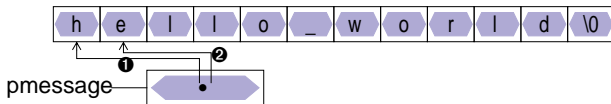
- es wird ein Speicherbereich für 16 Bytes reserviert und die Zeichen werden in diesen Speicherbereich hineinkopiert
- amessage ist ein konstanter Zeiger auf den Anfang des Speicherbereichs und kann nicht verändert werden
- der Inhalt des Speicherbereichs kann aber modifiziert werden  
`amessage[0] = 'h';`



# Zeiger, Felder und Zeichenketten (3)

- wird eine Zeichenkette zur Initialisierung eines **char**-Zeigers verwendet, ist der Zeiger eine Variable, die mit der Anfangsadresse der Zeichenkette initialisiert wird
  - ▶ die Zeichenkette selbst wird vom Compiler als konstanter Wert (String-Literal) im Speicher angelegt

```
const char *pmessage = "hello world";
```



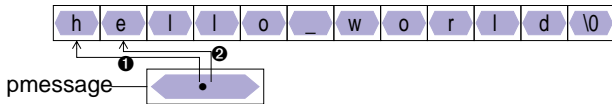
```
pmessage++; ②  
printf("%s", pmessage); /*gibt "ello world" aus*/
```

- ▶ es wird ein Speicherbereich für einen Zeiger reserviert (z. B. 4 Byte) und der Compiler legt die Zeichenkette `hello world` an irgendeiner Adresse im Speicher des Programms ab



# Zeiger, Felder und Zeichenketten (4)

```
const char *pmessage = "hello world";
```



```
pmessage++; ②  
printf("%s", 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
  - Compiler erkennt dies durch das Schlüsselwort `const` und verhindert schreibenden Zugriff über den Zeiger
  - manche Compiler legen solche Zeichenketten ausserdem in schreibgeschütztem Speicher an  
→ Speicherschutzverletzung beim Zugriff (falls der Zeiger nicht als `const`-Zeiger definiert wurde.

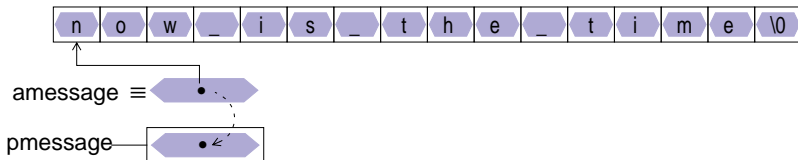


# Zeiger, Felder und Zeichenketten (5)

- 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



- 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[], t[])
{
    int i=0;
    while ( (s[i] = t[i]) != '\0' )
        i++;
}

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

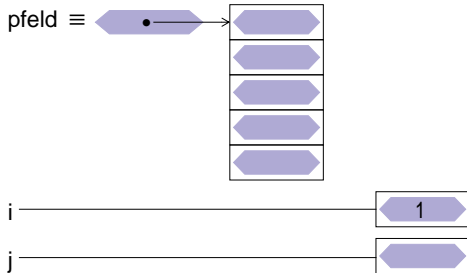
/* 3. Version */
void strcpy(char *s, *t)
{
    while ( *s++ = *t++ )
        ;
}
```



# Felder von Zeigern

- Auch von Zeigern können Felder gebildet werden
- Deklaration

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





# Felder von Zeigern

- Auch von Zeigern können Felder gebildet werden

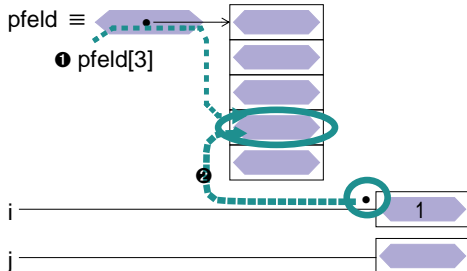
- Deklaration

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

- Zugriffe auf einen Zeiger des Feldes

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

①                      ②



# Felder von Zeigern

- Auch von Zeigern können Felder gebildet werden

- Deklaration

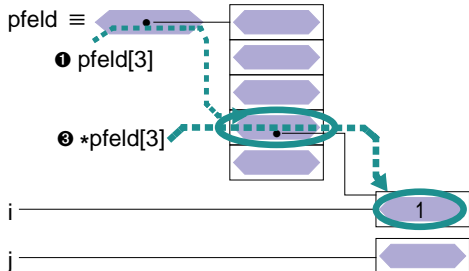
```
int *pfeld[5];  
int i = 1  
int j;
```

- Zugriffe auf einen Zeiger des Feldes

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

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

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



# Felder von Zeigern

- Auch von Zeigern können Felder gebildet werden

- Deklaration

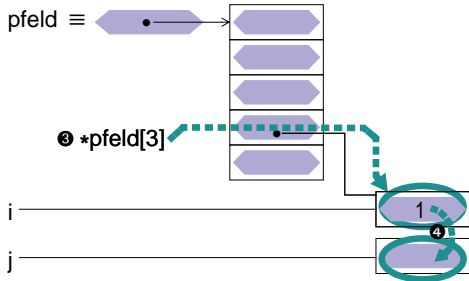
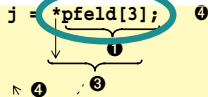
```
int *pfeld[5];  
int i = 1  
int j;
```

- Zugriffe auf einen Zeiger des Feldes

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

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

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

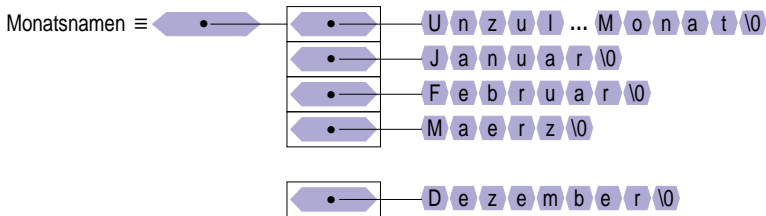


## Felder von Zeigern (2)

- Beispiel: Definition und Initialisierung eines Zeigerfeldes:

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

    return ( (n<0 || n>12) ?
            Monatsnamen[0] : Monatsnamen[n] );
}
```



# Argumente aus der Kommandozeile

- beim Aufruf eines Kommandos können normalerweise Argumente übergeben werden
- der Zugriff auf diese Argumente wird der Funktion **main( )** durch zwei Aufrufparameter ermöglicht:

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

oder

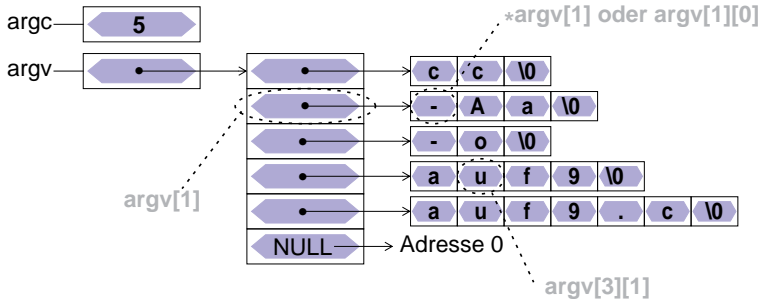
```
int  
main (int argc, char **argv)  
{  
    ...  
}
```

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



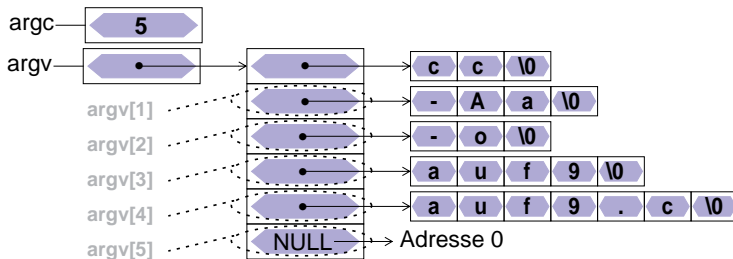
Kommando: `cc -Aa -o auf9 auf9.c`

Datei cc.c:  
...  
`main(int argc, char *argv[]) {`  
...  
`}`



- das folgende Programmstück gibt alle Argumente der Kommandozeile aus (außer dem Kommandonamen)

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



# Verbund-Datentypen / Strukturen (structs)

- Zusammenfassen mehrerer Daten zu einer Einheit (ein zusammenfassender Datentyp)
- Strukturdeklaration

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

- Definition einer Variablen vom Typ der Struktur

```
struct person p1;
```

- Zugriff auf ein Element der Struktur

```
p1.alter = 20;
```





# Zeiger auf Strukturen

- Konzept analog zu "Zeiger auf Variablen"
  - Adresse einer Struktur mit &-Operator zu bestimmen

- Beispiele

```
struct person stud1;  
struct person *pstud;  
pstud = &stud1;           /* ⇒ pstud → stud1 */
```

- Besondere Bedeutung zum Aufbau verketteter Strukturen
  - eine Struktur kann die Adresse einer weiteren Struktur desselben Typs enthalten



## Zeiger auf Strukturen (2)

- Zugriff auf Strukturkomponenten über einen Zeiger

- Bekannte Vorgehensweise

- ▶ \*-Operator liefert die Struktur
- ▶ .-Operator zum Zugriff auf Komponente
- ▶ Operatorenvorrang beachten!

➔ `(*pstud).alter = 21;`

nicht so gut leserlich!

- Syntaktische Verschönerung

➔ `->-Operator`

`pstud->alter = 21;`



- Strukturen in Strukturen sind erlaubt — aber
  - die Größe einer Struktur muss vom Compiler ausgerechnet werden können
    - Problem: eine Struktur enthält sich selbst
  - die Größe eines Zeigers ist bekannt (meist 4 Byte)
    - eine Struktur kann einen Zeiger auf eine gleichartige Struktur enthalten

```
struct liste {  
    struct student stud;  
    struct liste *rest;  
};
```

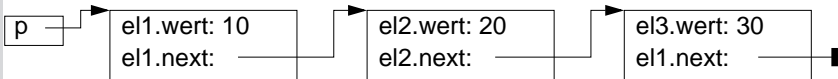
- ➔ Programmieren rekursiver Datenstrukturen  
z. B. verkettete Listen



# Verkettete Listen

- Mehrere Strukturen desselben Typs werden über Zeiger miteinander verkettet

```
struct liste { int wert; struct liste *next };  
struct liste e11, e12, e13;  
struct liste *p = &e11;  
e11.wert = 10; e12.wert = 20; e13.wert = 30;  
e11.next = &e12; e12.next = &e13; e13.next = NULL;
```



- Laufen über eine verkettete Liste

```
int summe = 0;  
while (p != NULL) {  
    summe += p-> wert;  
    p = p->next;  
}
```



- E-/A-Funktionalität nicht Teil der Programmiersprache
- Realisierung durch "normale" Funktionen
  - ▶ Bestandteil der Standard-Funktionsbibliothek
  - ▶ einfache Programmierschnittstelle
  - ▶ effizient
  - ▶ portabel
  - ▶ betriebssystemnah
- Funktionsumfang
  - ▶ Öffnen/Schließen von Dateien
  - ▶ Lesen/Schreiben von Zeichen, Zeilen oder beliebigen Datenblöcken
  - ▶ Formatierte Ein-/Ausgabe



- Jedes C-Programm erhält beim Start automatisch 3 E-/A-Kanäle:
  - **stdin** Standardeingabe
    - 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**  
( bei Erreichen des Dateiendes wird **EOF** signalisiert )
  - **stdout** Standardausgabe
    - normalerweise mit dem 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 (2)

## ■ Pipes

- die Standardausgabe eines Programms kann mit der Standardeingabe eines anderen Programms verbunden werden

- ▶ Aufruf

```
prog1 | prog2
```

- ! Die Umlenkung von Standard-E/A-Kanäle ist für die aufgerufenen Programme völlig unsichtbar

## ■ automatische Pufferung

- Eingabe von der Tastatur wird normalerweise vom Betriebssystem zeilenweise zwischengespeichert und erst bei einem **NEWLINE**-Zeichen (`' \n '`) an das Programm übergeben!



# 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`:

```
#include <stdio.h>
FILE *fopen(char *name, char *mode);
```

<b>name</b>	Pfadname der zu öffnenden Datei
<b>mode</b>	Art, wie die Datei geöffnet werden soll
"r"	zum Lesen
"w"	zum Schreiben
"a"	append: Öffnen zum Schreiben am Dateiende
"rw"	zum Lesen und Schreiben

- ▶ Ergebnis von `fopen`:  
Zeiger auf einen Datentyp **FILE**, der einen Dateikanal beschreibt  
im Fehlerfall wird ein **NULL**-Zeiger geliefert





# Öffnen und Schließen von Dateien (2)

## ■ Beispiel:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    FILE *eingabe;

    if (argv[1] == NULL) {
        fprintf(stderr, "keine Eingabedatei angegeben\n");
        exit(1);          /* Programm abbrechen */
    }

    if ((eingabe = fopen(argv[1], "r")) == NULL) {
        /* eingabe konnte nicht geöffnet werden */
        perror(argv[1]); /* Fehlermeldung ausgeben */
        exit(1);        /* Programm abbrechen */
    }

    ... /* Programm kann jetzt von eingabe lesen */
}
```

## ■ Schließen eines E/A-Kanals

```
int fclose(FILE *fp)
```

► schließt E/A-Kanal **fp**



## ■ Lesen eines einzelnen Zeichens

- von der Standardeingabe

```
int getchar( )
```

- lesen das nächste Zeichen
- geben das gelesene Zeichen als **int**-Wert zurück
- geben bei Eingabe von **CTRL-D** bzw. am Ende der Datei **EOF** als Ergebnis zurück

- von einem Dateikanal

```
int getc(FILE *fp )
```

## ■ Schreiben eines einzelnen Zeichens

- auf die Standardausgabe

```
int putchar(int c)
```

- schreiben das im Parameter **c** übergeben Zeichen
- geben gleichzeitig das geschriebene Zeichen als Ergebnis zurück

- auf einen Dateikanal

```
int putc(int c, FILE *fp )
```



## Zeichenweise Lesen und Schreiben (2)

- Beispiel: copy-Programm, Aufruf: `copy Quelldatei Zieldatei`

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

Teil 1: Aufrufargumente  
auswerten

```
int main(int argc, char *argv[]) {  
    FILE *quelle, *ziel;  
    int c;                /* gerade kopiertes Zeichen */  
  
    if (argc < 3) { /* Fehlermeldung, Abbruch */ }  
  
    if ((quelle = fopen(argv[1], "r")) == NULL) {  
        perror(argv[1]); /* Fehlermeldung ausgeben */  
        exit(EXIT_FAILURE); /* Programm abbrechen */  
    }  
  
    if ((ziel = fopen(argv[2], "w")) == NULL) {  
        /* Fehlermeldung, Abbruch */  
    }  
  
    while ( (c = getc(quelle)) != EOF ) {  
        putc(c, ziel);  
    }  
  
    fclose(quelle);  
    fclose(ziel);  
}
```



## ■ Lesen einer Zeile von der Standardeingabe

```
char *fgets(char *s, int n, FILE *fp)
```

- ▶ liest Zeichen von Dateikanal **fp** in das Feld **s** bis entweder **n-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, sonst **s**
- ▶ für **fp** kann **stdin** eingesetzt werden, um von der Standardeingabe zu lesen

## ■ Schreiben einer Zeile

```
int fputs(char *s, FILE *fp)
```

- ▶ schreibt die Zeichen im Feld **s** auf Dateikanal **fp**
- ▶ für **fp** kann auch **stdout** oder **stderr** eingesetzt werden
- ▶ als Ergebnis wird die Anzahl der geschriebenen Zeichen geliefert



## ■ Bibliotheksfunktionen — Schnittstelle

```
int printf(char *format, /* Parameter */ ... );  
int fprintf(FILE *fp, char *format, /* Parameter */ ... );  
int sprintf(char *s, char *format, /* Parameter */ ... );  
int snprintf(char *s, int n, char *format, /* Parameter */ ...);
```

## ■ 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 **s**  
(achtet dabei aber nicht auf das Feldende -> Pufferüberlauf möglich!)
- ▶ **snprintf** arbeitet analog, schreibt aber maximal nur **n** Zeichen  
(**n** sollte natürlich nicht größer als die Feldgröße sein)



- Zeichen im **format**-String können verschiedene Bedeutung haben
  - ▶ normale Zeichen: werden einfach auf 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

### ■ Format-Anweisungen

- %d, %i** **int** Parameter als Dezimalzahl ausgeben
- %f** **float** Parameter wird als Fließkommazahl (z. B. 271.456789) ausgegeben
- %e** **float** Parameter wird als Fließkommazahl in 10er-Potenz-Schreibweise (z. B. 2.714567e+02) ausgegeben
- %c** **char**-Parameter wird als einzelnes Zeichen ausgegeben
- %s** **char**-Feld wird ausgegeben, bis '`\0`' erreicht ist



## ■ Bibliotheksfunktionen — Schnittstelle

```
int scanf(char *format, /* Parameter */ ...);  
int fscanf(FILE *fp, char *format, /* Parameter */ ...);  
int sscanf(char *s, const char *format, /* Parameter */ ...);
```

- Die Funktionen lesen Zeichen von **stdin** (**scanf**), **fp** (**fscanf**) bzw. aus dem **char**-Feld **s**.
- **format** gibt an, welche Daten hiervon extrahiert und in welchen Datentyp konvertiert werden sollen
- Die folgenden Parameter sind Zeiger auf Variablen der passenden Datentypen (bzw. **char**-Felder bei Format **%s**), in die die Resultate eingetragen werden
- relativ komplexe Funktionalität, für Details siehe Manual-Seiten



- Fast jeder Systemcall/Bibliotheksaufruf kann fehlschlagen
  - Fehlerbehandlung unumgänglich!
- Vorgehensweise:
  - Rückgabewerte von Systemcalls/Bibliotheksaufrufen abfragen
  - Im Fehlerfall (meist durch Rückgabewert -1 angezeigt): Fehlercode steht in der globalen Variable **errno**
- Fehlermeldung kann mit der Funktion **perror** auf die Fehlerausgabe ausgegeben werden:

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

