

## H.1 Überblick

- 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

## H.2 Standard Ein-/Ausgabe

- Jedes C-Programm erhält beim Start automatisch 3 E/A-Kanäle:
  - ◆ **stdin** Standardeingabe
    - normalerweise mit der Tastatur verbunden, Umlenkung durch <
    - Dateiende (**EOF**) wird durch Eingabe von **CTRL-D** am Zeilenanfang signalisiert
  - ◆ **stdout** Standardausgabe
    - normalerweise mit dem Bildschirm (bzw. dem Fenster, in dem das Programm gestartet wurde) verbunden, Umlenkung durch >
  - ◆ **stderr** Ausgabekanal für Fehlermeldungen
    - normalerweise ebenfalls mit Bildschirm verbunden
- automatische Pufferung
  - ◆ Eingabe von der Tastatur wird normalerweise vom Betriebssystem zeilenweise zwischengespeichert und erst bei einem **NEWLINE**-Zeichen ('\n') an das Programm übergeben!

## H.3 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**
    - Prototyp:

```
FILE *fopen(char *name, char *mode);
```
- name Pfadname der zu öffnenden Datei  
mode 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

## H.3 Öffnen und Schließen von Dateien (2)

- Beispiel:

```
#include <stdio.h>

main(void) {
    FILE *eingabe;
    char dateiname[256];

    printf("Dateiname: ");
    scanf("%s\n", dateiname);

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

    ... /* Programm kann jetzt von eingabe lesen */
    ... /* z. B. mit c = getc(eingabe) */
}
```

- Schließen eines E/A-Kanals

```
int fclose(FILE *fp)
```
- schließt E/A-Kanal **fp**

## H.4 Zeichenweise Lesen und Schreiben

### ■ Lesen eines einzelnen Zeichens

- ◆ von der Standardeingabe
- ◆ von einem Dateikanal

```
int getchar( )
```

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

- 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

### ■ Schreiben eines einzelnen Zeichens

- ◆ auf die Standardausgabe
- ◆ auf einen Dateikanal

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

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

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

### ■ Beispiel: copy-Programm

```
#include <stdio.h>
main(int argc, char *argv[]) {
    FILE *quelle;
    FILE *ziel;
    char quelldatei[256], zieldatei[256];
    int c; /* gerade kopiertes Zeichen */

    printf("Quelldatei und Zielfile eingeben: ");
    scanf("%s %s\n", quelldatei, zieldatei);

    if ((quelle = fopen(quelldatei, "r")) == NULL) {
        perror(quelldatei); /* Fehlermeldung ausgeben */
        exit(1); /* Programm abbrechen */
    }

    if ((ziel = fopen(zieldatei, "w")) == NULL) {
        perror(zieldatei); /* Fehlermeldung ausgeben */
        exit(1); /* Programm abbrechen */
    }

    /* ... */
}
```

#### Teil 1: Dateien öffnen

## H.4 Zeichenweise Lesen und Schreiben (3)

- ... Beispiel: copy-Programm
- Fortsetzung

```
/* ... */
while ( (c = getc(quelle)) != EOF ) {
    putc(c, ziel);
}

fclose(quelle);
fclose(ziel);
}
```

Teil 2: kopieren

## H.5 Formatierte Ausgabe

### 1 Bibliotheksfunktionen — Prototypen (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  
-> potentielle Sicherheitsprobleme!)
- snprintf arbeitet analog, schreibt aber maximal nur n Zeichen  
(n sollte natürlich nicht größer als die Feldgröße sein)

## 2 Formatangaben

- 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 oder double Parameter wird als Fließkommazahl (z. B. 271.456789) ausgegeben
- %e** float oder double 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

## H.6 Formatierte Eingabe

### 1 Bibliotheksfunktionen — Prototypen (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, hier nur Kurzüberblick für Details siehe Manual-Seiten

## 2 Bearbeitung der Eingabe-Daten

- White space (Space, Tabulator oder Newline `\n`) bildet jeweils die Grenze zwischen Daten, die interpretiert werden
  - white space wird in beliebiger Menge einfach überlesen
  - Ausnahme: bei Format-Anweisung `%c` wird auch white space eingelesen
- Alle anderen Daten in der Eingabe müssen zum **format**-String passen oder die Interpretation der Eingabe wird abgebrochen
  - wenn im **format**-String normale Zeichen angegeben sind, müssen diese exakt so in der Eingabe auftauchen
  - wenn im **format**-String eine Format-Anweisung (...) angegeben ist, muß in der Eingabe etwas hierauf passendes auftauchen
    - ➔ diese Daten werden dann in den entsprechenden Typ konvertiert und über den zugehörigen Zeiger-Parameter der Variablen zugewiesen
- Die `scanf`-Funktionen liefern als Ergebnis die Zahl der erfolgreich an die Parameter zugewiesenen Werte

## 3 Format-Anweisungen

<b>%d</b>	int	■ nach % kann eine Zahl folgen, die die maximale Feldbreite angibt
<b>%hd</b>	short	<code>%3d</code> = 3 Ziffern lesen
<b>%ld</b>	long int	<code>%5c</code> = 5 char lesen (Parameter muß dann Zeiger auf char-Feld sein)
<b>%lld</b>	long long int	➤ <code>%5c</code> überträgt exakt 5 char (hängt aber kein ' <code>\0</code> ' an!)
<b>%f</b>	float	➤ <code>%5s</code> liest max. 5 char (bis white space) und hängt ' <code>\0</code> ' an
<b>%lf</b>	double	
<b>%Lf</b>	long double	
		analog auch <code>%e</code> oder <code>%g</code>
<b>%c</b>	char	■ Beispiele:
<b>%s</b>	String, wird automatisch mit ' <code>\0</code> ' abgeschl.	<pre>int a, b, c, d, n; char s1[20] = "XXXXXX", s2[20]; n = scanf("%d %2d %3d %5c %s %d",           &amp;a, &amp;b, &amp;c, &amp;s1, s2, &amp;d); Eingabe: 12 1234567 sowas hmm Ergebnis: n=5, a=12, b=12, c=345           s1="67 sox", s2="was"</pre>