

Ein-/Ausgabe

Einführung in die Programmierung

Michael Felderer

Institut für Informatik, Universität Innsbruck

Standardbibliothek

- Die Standardbibliothek wurde im ANSI-Standard definiert.
- Die Standardbibliothek ist **kein Teil** der Programmiersprache C!
- Eine Umgebung, die standardisiertes C realisiert, wird auch die Funktionen, Typen und Makros dieser Bibliothek zur Verfügung stellen.
- Die Standardbibliothek umfasst bestimmte Header-Dateien, die über `#include` in das Programm eingebunden werden können.

Auswahl wichtiger Header-Dateien

- `<stdio.h>` = Ein- und Ausgabe
- `<ctype.h>` = Tests für Zeichenklassen
- `<string.h>` = Funktionen für Zeichenketten
- `<math.h>` = Mathematische Funktionen
- `<stdlib.h>` = Hilfsfunktionen zur Umwandlung von Zahlen, für Speicherverwaltung etc.
- `<time.h>` = Funktionen für Datum und Uhrzeit
- `<limits.h>` = Definiert Konstanten für den Wertumfang der ganzzahligen Typen
- `<float.h>` = Definiert Konstanten, die sich auf Gleitpunktarithmetik beziehen
-

Beispiel – stdio.h

- Diese Funktionen realisieren ein einfaches Modell für Texteingabe und Textausgabe.
 - Die Ein- und Ausgabe von Daten in C wird über sogenannte Streams (Ströme) realisiert.
- In C unterscheidet man
 - Textströme
 - Ein Textstrom ist eine Folge von Zeilen.
 - Jede Zeile endet mit einem Zeilentrenner (`\n`).
 - Alle sichtbaren ASCII-Zeichen und einige SteuerCodes werden verwendet.
 - Intern wird alles immer gleich dargestellt (Konvertierung notwendig).
 - Binäre Ströme
 - Diese Ströme werden Byte für Byte verarbeitet (keine Konvertierung).
- Weitere Unterscheidung
 - Byte-orientierter Strom mit Datentyp `char`.
 - Breitzeichenorientierter Strom mit Datentyp `wchar_t`.
 - Nachfolgend werden nur byte-orientierte Ströme betrachtet.

Pufferung

- Es ist nicht sinnvoll, immer Zeichen für Zeichen zu verarbeiten bzw. zu übertragen.
- Daher gibt es drei Arten der Pufferung bei Strömen.
 - Vollgepuffert
 - Zeichen werden erst übertragen, wenn der Puffer (z.B. 4096 Bytes) voll ist.
 - Zeilengepuffert
 - Zeichen werden erst übertragen, wenn eine neue Zeile begonnen wird oder der Puffer voll ist.
 - Ungepuffert
 - Zeichen werden sofort übertragen.

Standard-Streams

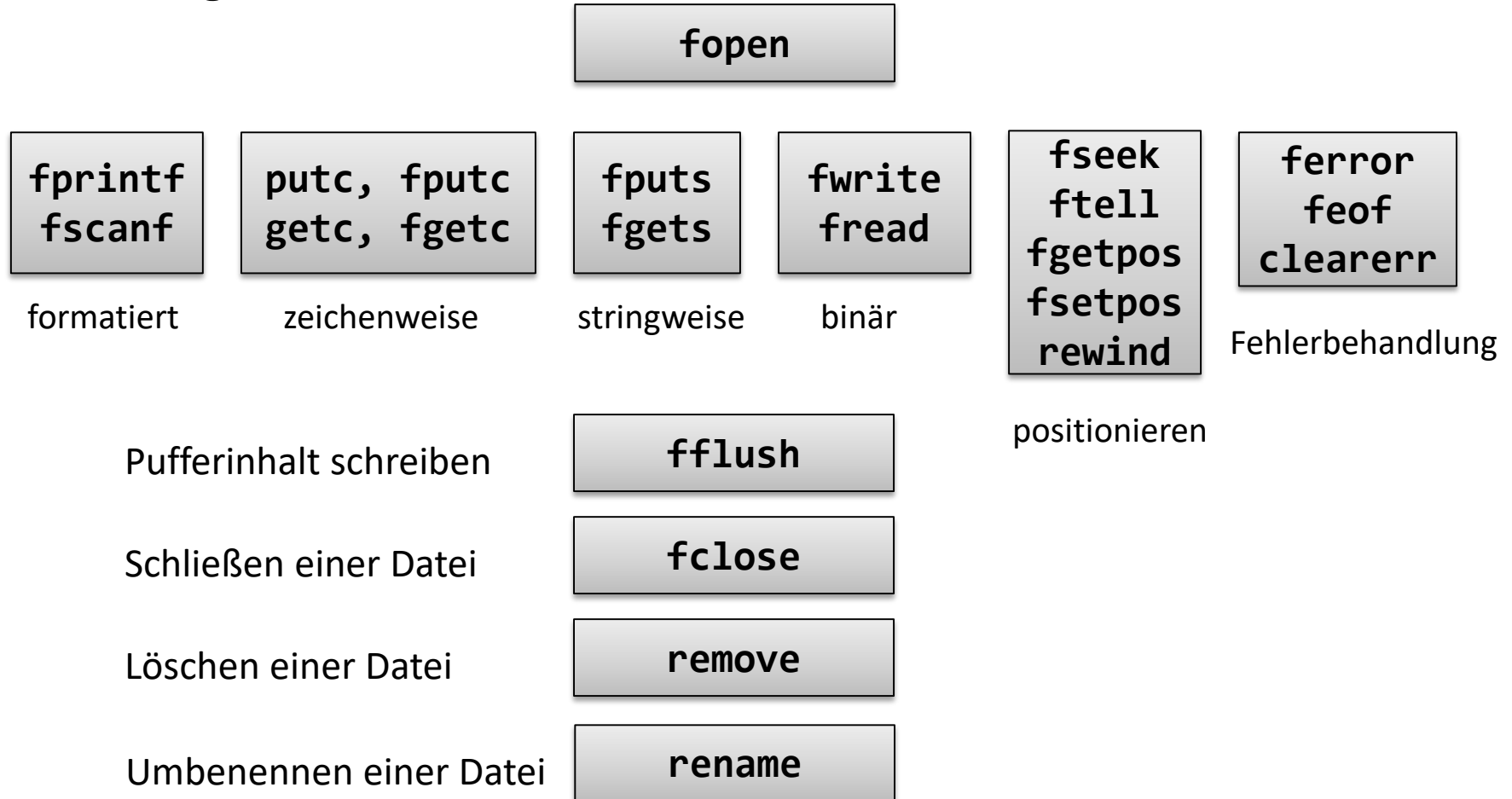
- Drei Standard-Ströme (`stdin`, `stdout`, `stderr`) sind bei jedem C-Programm von Anfang an vorhanden.
 - Es handelt sich dabei um Zeiger auf ein `FILE`-Objekt.
- Die Standard-(Text)-Ströme
 - **`stdin`**
 - Das ist die Standardeingabe (standard input), die gewöhnlich mit der Tastatur verbunden ist.
 - Die Standardeingabe wird zeilenweise gepuffert.
 - **`stdout`**
 - Die Standardausgabe (standard output) ist gewöhnlich mit dem Bildschirm zur Ausgabe verbunden.
 - Die Standardausgabe wird zeilenweise gepuffert.
 - **`stderr`**
 - Die Standardfehlerausgabe (standard error output) ist gewöhnlich auch mit dem Bildschirm verbunden, aber die Ausgabe erfolgt ungepuffert.

Dateien

- Oft möchte man Daten nicht nur am Bildschirm ausgeben, sondern auch in eine Datei schreiben und diese Daten später wieder aus der Datei lesen.
- In C gibt es Standardfunktionen für das Öffnen einer Datei.
- Beim Öffnen wird ein Speicherobjekt vom Typ FILE angelegt und initialisiert.
- Eine erfolgreich geöffnete Datei liefert immer einen Zeiger auf ein FILE-Speicherobjekt zurück, das mit dem Strom verbunden ist.
- Das FILE-Objekt ist eine Struktur, die alle nötigen Informationen für die Ein- und Ausgabefunktionen enthält.
- Die folgenden Beschreibungen beziehen sich auf den zid-gpl!
 - Manual-Seiten lesen!
 - Der C-Standard ist manchmal etwas allgemeiner!

Überblick über wichtige Funktionen

- Wichtige Funktionen und Makros



- Der Zugriff erfolgt meist über einen File-Zeiger, der auf eine Struktur vom Typ `FILE` zeigt.

FILE-Zeiger

- Deklaration
 - `FILE *fp;`
- `fp` ist ein Zeiger auf eine `FILE`-Struktur (struct in `stdio.h`).
- Wird gebraucht, um mit Strömen zu arbeiten.
 - Beinhaltet alle benötigten Informationen.
 - Ein `FILE`-Zeiger zeigt immer auf eine bestimmte Position im Strom.
 - Lesen oder Schreiben verändert die aktuelle Position des Zeigers.
- `stdin`, `stdout`, `stderr` sind automatisch geöffnete `FILE`-Zeiger.

Öffnen von Dateien in C (1)

- Allgemeine Form

```
FILE *fopen(const char *restrict filename,  
            const char *restrict mode);
```

- filename: Spezifiziert den Dateinamen (darf auch ein Pfad sein).
- mode: Spezifiziert den Zugriffsmodus.

mode	Beschreibung
r	Öffnen zum Lesen
w	Öffnen zum Schreiben (Datei muss davor nicht existieren, ansonsten wird sie überschrieben)
a	Öffnen zum Anhängen (Datei muss davor nicht existieren)
r+	Öffnen zum Lesen & Schreiben, Startet am Anfang
w+	Öffnen zum Lesen & Schreiben, Überschreibt die Datei
a+	Öffnen zum Lesen & Schreiben, Hängt am Ende an

- Liefert einen Zeiger auf den entsprechenden Datenstrom zurück.

Öffnen von Dateien in C (2)

- Schlägt `fopen` fehl, liefert es den NULL-Pointer zurück.
 - z.B. wenn eine zum Lesen geöffnete Datei nicht existiert.
- Wechsel beim Zugriffsmodus
 - Schreiben, dann Lesen: Schreiboperationen müssen mit bestimmten Funktionen (`fflush`, `fsetpos`, `fseek`, `rewind`) abgeschlossen werden!
 - Lesen, dann Schreiben: Nach Leseoperationen muss mit bestimmten Funktionen (`fseek`, `fsetpos`, `rewind`) der Schreibzeiger positioniert werden.
- Zugriffsrechte
 - Zugriffsrechte müssen Operation (Lesen, Schreiben) erlauben!
 - Unter Linux/Unix strenger als unter Windows!

Öffnen von Dateien in C (3)

- Beispiel

...

```
FILE *fp = fopen(filename, "r");
```

```
if (fp != NULL) {
```

```
    printf("Datei zum Lesen geöffnet\n");
```

```
    ...
```

```
} else {
```

```
    printf("Datei konnte nicht geöffnet werden\n");
```

```
    ...
```

```
}
```

```
...
```

Öffnen von Dateien in C (4)

- Dateien können für die Ein- bzw. Ausgabe von Text oder im Binärmodus geöffnet werden.
 - Beides geht mit `fopen`.
- Für den binären Modus wird ein "b" an das Ende des mode-Strings geschrieben.
 - `rb`, `wb`, `ab`, `r+b`, `w+b`, `a+b`
 - Hat bei Unix/Linux keine Auswirkung (keine Bedeutung)!

Schließen von Dateien

- Aufruf

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

- `fclose` schließt den zu `fp` gehörigen Datenstrom.
- Leert vorher alle damit assoziierten Puffer.

- Rückgabewert

- Liefert bei Erfolg den Rückgabewert 0.
- Liefert einen Wert ungleich 0 bei Fehlern.

- Sollte immer verwendet werden, wenn die Verarbeitung der Daten aus einem Strom beendet wurde!

- Wenn das Programm abstürzt, gehen ohne `fclose` möglicherweise Daten verloren.
- Ein Programm darf nur eine begrenzte Anzahl an Datenströmen öffnen.
- Wenn das Programm korrekt terminiert, dann werden alle Datenströme automatisch geschlossen.

Lesen und Zurückstellen

- Einzelne Zeichen einlesen

```
int fgetc(FILE *fp);
```

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

```
int getchar();
```

- Zu beachten

- getc darf als Makro implementiert sein.
- Liefern gelesenes Zeichen zurück.
- Tritt ein Fehler auf, dann erhält man EOF (zid-gpl: Konstante = -1).
- getchar() ist gleichwertig zu getc(stdin).

- Einzelnes Zeichen in einen Datenstrom zurückstellen

```
int ungetc(int c, FILE *fp);
```

- Zurückgeschobenes Zeichen wird retourniert.
- EOF bei Fehler!
- Wird beim nächsten Lesen wieder gelesen.
 - Nicht bei bestimmten Operationen (z.B. fflush)!

Schreiben

- Einzelne Zeichen schreiben

```
int fputc(int c, FILE *fp);
```

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

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

- Zu beachten

- Zurückgegeben wird das geschriebene Zeichen oder EOF im Falle eines Fehlers!
- putc darf als Makro implementiert sein.
- putchar(int c) ist gleichwertig zu putc(c, stdout).

Beispiel

```
#include <stdio.h>
#include <stdlib.h>
#define FILENAME "datei.txt"
#define COPY "kopie.txt"
int main(void) {
    FILE *fpr = fopen(FILENAME, "r");
    FILE *fpw = fopen(COPY, "w");
    if (fpr == NULL || fpw == NULL) {
        printf("Fehler beim Öffnen von %s bzw. %s", FILENAME, COPY);
        return EXIT_FAILURE;
    }
    for (int c; (c = fgetc(fpr)) != EOF; ) {
        if (fputc(c, fpw) == EOF) {
            printf("Fehler beim Kopieren von %s bzw. %s", FILENAME, COPY);
            return EXIT_FAILURE;
        }
    }
    if (fclose(fpr) != 0 || fclose(fpw) != 0){
        printf("Fehler beim Schließen von %s bzw. %s", FILENAME, COPY);
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Interaktive Aufgabe

- Im folgenden Codeausschnitt soll die Datei datei1.txt zeichenweise in eine neue Datei datei2.txt kopiert werden. Allerdings sind nach diesem Vorgang beide Dateien leer. Was wurde falsch gemacht?

...

```
#define FILENAME1 "datei1.txt"
```

```
#define FILENAME2 "datei2.txt"
```

...

```
FILE *fpr, *fpw;
```

```
int c;
```

```
fpr = fopen(FILENAME1, "w");
```

```
fpw = fopen(FILENAME2, "a+");
```

...

```
while ( (c=fgetc(fpr)) != EOF ) {
```

```
    fputc(c, fpw);
```

```
}
```

...

Zeilenweise einlesen (Wiederholung)

- Zeilenweise einlesen

```
char *fgets(char *restrict buf,  
            int n,  
            FILE *restrict fp);
```

- Zu beachten

- Vom Strom fp werden bis zu n-1 Zeichen in den Puffer buf eingelesen.
- Am Ende wird ein Terminierungszeichen angehängt.
- Lesevorgang wird bei Zeilen- oder Dateiende oder nach n-1 Zeichen beendet.
 - Die Anzahl der Zeichen (kann auch weniger als n-1 Zeichen sein) wird dann eingelesen.
 - Das Newline-Zeichen wird auch (falls in den eingelesenen Zeichen enthalten) im Puffer gespeichert.
- Retourniert einen NULL-Zeiger, wenn nichts gelesen wurde oder ein Fehler auftrat.
 - Sonst wird die Anfangsadresse von buf zurückgegeben.

Zeilenweise schreiben

- Zeilenweise schreiben (2 Möglichkeiten)
 - `int fputs(const char *restrict str, FILE *restrict fp);`
 - `int puts(const char *str);`
- Zu beachten
 - `fputs` schreibt den nullterminierten String `str` in den Strom `fp`.
 - Das Terminierungszeichen wird **nicht** mit in den Strom geschrieben!
 - `puts` gibt den String `str` auf `stdout` aus.
 - Newline-Zeichen wird hinzugefügt.
 - Rückgabewert ist nicht negativ bei Erfolg, ansonsten EOF.

Formatiertes Schreiben

- Formatierte Ausgabe

```
int fprintf(FILE *restrict fp,  
            const char *restrict format, ...)
```

- Zu beachten

- Arbeitet wie printf, jedoch für den angegebenen Datenstrom.
 - Formatierung etc. siehe Folien über printf!
- Der erste Parameter ist ein Zeiger auf den Strom, in den geschrieben wird.
- Der Rückgabewert zeigt an, wie viele Zeichen tatsächlich geschrieben wurden.
 - Bei Schreibfehlern wird ein negativer Wert zurückgegeben.

- Beispiel

- `if (fprintf(fp, "Hallo, File!") < 0) ...;`

Formatiertes Lesen

- Formatiertes Einlesen

```
int fscanf(FILE *restrict fp,  
           const char *restrict format, ...)
```

- Zu beachten

- Arbeitet wie scanf, jedoch für den angegebenen Datenstrom.
- Der erste Parameter ist ein Zeiger auf den Strom, aus dem gelesen wird.
- Der Rückgabewert zeigt an, wie viele Konvertierungen vorgenommen wurden:
 - Ist 0, wenn die vorgefundenen Daten nicht den geforderten Datentypen entsprechen.
 - Wenn ein Fehler auftritt, oder das Dateiende erreicht wurde, bevor Daten gelesen werden konnten, wird EOF zurückgegeben.

- Allgemeine Syntax
 - %[*][W][L]U
 - * = Einlesen, aber nicht speichern
 - W = Weite, Feldbreite
 - L = Längenangabe
 - U = Umwandlungszeichen, Konvertierungsspezifizierer
- Suchmengenkonvertierung
 - Kann anstelle des Umwandlungszeichens verwendet werden.
 - %[bezeichner] = Es wird eingelesen, bis ein Zeichen vorkommt, das nicht in der Liste bezeichner vorkommt.
 - %[^bezeichner] = Es wird eingelesen, bis ein Zeichen vorkommt, das in der Liste bezeichner vorkommt.

Beispiel

Textdatei (Format: String, String, long, String):

```
Haller,Fritz,73737,Grossenbach
Albrich,Toni,78373,Studenstadt
Zeppelin,Helmut,83837,Unterwasserbach
...
```

```
FILE *fp = fopen("test.txt", "r"); /* Datei test.txt öffnen */
if (fp == NULL) {
    printf("Kann Datei test.txt nicht zum Lesen öffnen\n");
    ...
}

while (fscanf(fp,"%[^,],%[^,],%lu,%s\n", name, vorname, &plz, ort) != EOF) {
    fprintf(stdout, "%s,%s,%lu,%s\n", name, vorname, plz, ort);
    ...
}
```


Lesen und Schreiben von Binärdaten (1)

- Lesen von binären Daten
 - `size_t fread(void *restrict ptr, size_t size_of_elements, size_t number_of_elements, FILE *restrict fp);`
- Zu beachten
 - Liest Blöcke von Daten.
 - Daten werden rein binär behandelt.
 - `ptr`: Zieladresse für die Daten.
 - `size_of_elements`: Die Größe einer zu lesenden Einheit, in Byte (z.B. `sizeof(int)`, `sizeof(my_struct)`).
 - `number_of_elements`: Gibt die Anzahl der zu lesenden Einheiten an.
 - `fp`: Strom, aus dem gelesen werden soll.
 - Gibt die Anzahl der gelesenen Einheiten zurück.

Lesen und Schreiben von Binärdaten (2)

- Schreiben von binären Daten
 - `size_t fwrite(const void *restrict ptr, size_t size_of_elements, size_t number_of_elements, FILE *restrict fp);`
- Zu beachten
 - Schreibt Blöcke von binären Daten.
 - Geschriebene Daten sind plattformabhängig!
 - `ptr`: Leseadresse der Daten
 - `size_of_elements`: Die Größe einer zu schreibenden Einheit, in Byte (z.B. `sizeof(int)`, `sizeof(my_struct)`)
 - `number_of_elements`: Gibt die Anzahl der zu schreibenden Einheiten an.
 - `fp`: Strom, in den geschrieben werden soll.
 - Gibt die Anzahl der geschriebenen Einheiten zurück.

Bewegen des FILE-Zeigers

- Position ändern

```
int fseek(FILE *fp, long offset, int position);
```

- Zu beachten

- fp: Der zu verändernde File-Zeiger.
- offset: Die Anzahl der Bytes, um welche die aktuelle Position des Zeigers im Strom verschoben werden soll, ausgehend von der Stelle, die durch position angegeben wird.
 - Kann auch negativ sein.
- position: Einer von 3 Werten
 - SEEK_SET: Anfang der Datei
 - SEEK_CUR: Aktuelle Position des angegebenen FILE-Zeigers
 - SEEK_END: Ende der Datei

- Rückgabewert

- 0 bei Erfolg, EOF-Flag gelöscht.
- ungleich 0 bei Fehler

Positionen behandeln

- Position ermitteln/setzen

```
int fgetpos(FILE *fp, fpos_t *restrict pos);
```

```
int fsetpos(FILE *fp, const fpos_t *pos);
```

- Ermittelt bzw. setzt die aktuelle Position in einem Strom (Abstand zum Anfang in Bytes).
 - pos muss für fsetpos von fgetpos ermittelt werden!
- Geben bei Erfolg 0 zurück (EOF-Flag wird bei fsetpos gelöscht), ansonsten ungleich 0.

- Weitere Funktionen

```
long ftell(FILE *fp);
```

```
void rewind(FILE *fp);
```

- ftell – Liefert die aktuelle Schreib-/Leseposition im Strom zurück.
 - Mit fseek gemeinsam benutzen.
- rewind – Setzt die aktuelle Schreib-/Leseposition an den Anfang des Stroms.

Fehler bei File-Operationen

- Die meisten File-Operationen liefern bei Erreichen des Dateiendes sowie bei Fehlern den Wert EOF zurück.
- Überprüfen ob Fehler oder Dateiende
 - **int feof(FILE *fp);**
 - Ergibt ungleich 0, wenn EOF Flag für fp gesetzt ist.
 - 0 sonst
 - **int ferror(FILE *fp);**
 - Ergibt ungleich 0, wenn Fehler-Flag für fp gesetzt ist.
 - 0 sonst
- Löschen des Fehler- und EOF-Flags
 - **void clearerr(FILE *fp);**

Datei löschen oder umbenennen

- Löschen

- `int remove(const char *pathname);`
 - Bei Erfolg wird 0 zurückgeliefert.
 - Bei Fehler wird -1 zurückgeliefert.
 - Die erforderlichen Zugriffsrechte müssen vorhanden sein!

- Umbenennen

- `int rename(const char *oldname, const char *newname);`
 - Bei Erfolg wird 0 zurückgeliefert.
 - Im Fehlerfall wird -1 zurückgeliefert.

Pufferung

- Pufferung für Datenstrom setzen
 - `int setvbuf(FILE *restrict fp, char *restrict buf, int type, size_t size);`
 - type
 - `_IONBF`: Ungepuffert
 - `_IOLBF`: Zeilenpufferung
 - `_IOFBF`: Vollpufferung
 - buf = Puffer, size = Größe des Puffers.
- Puffer leeren
 - `int fflush(FILE *stream);`
 - Alle gepuffert Daten des angegebenen Ausgabestroms werden geschrieben.
 - Ist der Parameter NULL, werden alle geöffneten Ausgabeströme geleert.
 - Bei Erfolg wird 0 zurückgeliefert, ansonsten EOF.

- Einige Funktionen setzen im Fehlerfall auch noch die globale Variable `errno` auf einen bestimmten Wert.
 - Der Rückgabewert von -1 deutet nur auf einen Fehler hin.
 - Die Art des Fehlers wird noch in `errno` hinterlegt.
- Funktionen die `errno` setzen
 - `fopen`, `fclose`, `remove`, `rename`, `fsetpos`, `fgetpos`, `fseek`, `ftell`, `fflush`
- Welcher Wert wird gesetzt?
 - Vordefinierte Konstanten, auf die man dann überprüfen kann.
 - Konstanten sind systemabhängig!
 - Wert ist nur bis zum nächsten Fehler gültig!
- Ausgabe von Fehlermeldungen
 - Ausgabe von Fehlermeldungen mit den Funktionen `perror` oder `strerror`
 - Manual-Seiten lesen!
 - Headerdatei `errno.h` muss möglicherweise eingebunden werden.

Dateien in UNIX/Linux (1)

- Elementare E/A Funktionen bieten erweiterte Möglichkeiten.
 - Sind aber **NICHT** Bestandteil des C Standards.
 - Verwenden nicht FILE-Zeiger sondern File-Deskriptoren (realisiert als `int`).
- Erlauben Arbeit mit
 - Verzeichnissen
 - Verzeichnisbäumen
 - Zugriffsrechten
 - Zeitstempeln
 - und vieles mehr...
- Werden im 2. Semester besprochen (LV Betriebssysteme)

Dateien in UNIX/Linux (2)

- Beispiele

```
int open(const char *path, int oflag);
```

```
int close(int fd);
```

```
ssize_t read(int fd, void *puffer, size_t bytes);
```

```
ssize_t write(int fd, void *puffer, size_t bytes);
```

- Umwandlung zwischen File-Deskriptor und FILE*

- `FILE *fdopen(int fd, const char *mode);`