6.1.F Extras/Schreiben in Dateien – Ergänzungen und Bilder

6.1.F.1 Fehler und Rückgabewerte beim Öffnen – für Profis

Manchmal ist es notwendig, wenn man die Art des Fehlers genauer bestimmen kann:

a) Der Benutzer bekommt die Art des Fehlers genannt (z. B. fehlende Zugriffsberechtigungen auf dem Datenträger) und kann etwas unternehmen

b) Das Programm kann je nach Fehlerart eine alternative Lösung anbieten (z. B. einen anderen Speicherort)

Und so kann man Fehlerart genauer bestimmen:

a) Für den Benutzer wie im ersten Beispiel per Fehlermeldung:

```
1    perror("datei.txt");
```

b) Für das Programm gibt es die (globale) `int`-Variable `errno`, die eine Fehlernummer enthält. Die Fehlernummer 0 sagt aus, dass kein Fehler existiert, die anderen Fehlernummern geben jeweils eine Fehlerart an. Meistens arbeitet man jedoch nicht direkt mit den Nummern, sondern mit symbolischen Konstanten. Tabelle 1 zeigt eine Auswahl der Konstanten, die beim Öffnen von Dateien auftreten können. Auf einem Linux-System erhält man mit dem Befehl `errno -l` eine Liste aller Fehler, die in der C-Standardbibliothek benutzt werden.

Nr.	Konstante	Was ist passiert?
0	—	Kein Fehler
2	ENOENT	Nicht exist. Verzeichnis oder tote Verknüpfung im Pfadnamen
13	EACCES	Zugriff verweigert (Berechtigung fehlt)
14	EFAULT	Pfadname liegt an nicht lesbarer Adresse
21	EISDIR	Pfadname ist der Name einer bestehenden Verzeichnisses
40	ELOOP	Im Pfadnamen befindet sich eine Schleife von Verknüpfungen
24	EMFILE	Zu viele offene Dateien für diesen Prozess
36	ENAMETOOLONG	Zu langer Pfadname
23	ENFILE	Zu viele offene Dateien für dieses System
28	ENOSPC	Kein Platz für eine neue Datei
30	EROFS	Gerät ist schreibgeschützt
26	ETXTBSY	Pfadname ist Name eines Programms, das gerade ausgeführt wird

Tabelle 1: Fehlernummern und symbolische Konstanten beim Öffnen

Das folgende Programm versucht, möglichst viele dieser Fehler zu erzeugen (Pfadnamen passend für ein Linux-System):

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <errno.h>
4  #include <stdlib.h>
5  #include <locale.h>
6  #define ANZEIGE fprintf(stderr, "errno = %3d  ", errno); perror("")
7
8  int main(void)
9  {
10     char dateiname[80000];
11     FILE *dm;
```

```
12      printf("Bitte␣eine␣DVD␣einlegen,␣falls␣moeglich␣;-)\n"
13             "Dann␣eine␣Taste␣druecken");
14      fflush(stdout);
15      getchar();
16      setlocale(LC_ALL, "");
17                                              ANZEIGE;
18      dm=fopen("", "w");                      ANZEIGE;
19      dm=fopen("/beispiel.txt", "w");         ANZEIGE;
20      dm=fopen(NULL,"w");                     ANZEIGE;
21      dm=fopen("./", "w");                    ANZEIGE;
22      dm=fopen("a.out","w");                  ANZEIGE;
23      dm=fopen("/media/cdrom/x", "w");        ANZEIGE;
24
25      memset(dateiname, 'A', 79999);
26      dm=fopen(dateiname,"w");                ANZEIGE;
27
28      system("ln␣-s␣x␣y");
29      system("ln␣-s␣y␣x");
30      dm=fopen("x","w");                      ANZEIGE;
31      remove("x");
32      remove("y");
33
34      do{
35          dm=fopen("x.txt","w");
36      }while(dm!=NULL);                       ANZEIGE;
37
38      return 0;
39  }
```

Terminal

```
schueler@debian964:~$ gcc -o fehlererz fehlererz.c
schueler@debian964:~$ fehlererz
Bitte eine DVD einlegen, falls moeglich ;-)
Dann eine Taste druecken
errno =   0  Erfolg
errno =   2  Datei oder Verzeichnis nicht gefunden
errno =  13  Keine Berechtigung
errno =  14  Ungültige Adresse
errno =  21  Ist ein Verzeichnis
errno =  26  Das Programm kann nicht ausgeführt oder verändert werden
errno =  30  Das Dateisystem ist nur lesbar
errno =  36  Der Dateiname ist zu lang
errno =  40  Zu viele Ebenen aus symbolischen Links
errno =  24  Zu viele offene Dateien
```

6.1.F.2 Fehler und Rückgabewerte beim Schreiben – für Profis

Tabelle 2 zeigt eine Auswahl der Konstanten, die beim Schreiben von Dateien auftreten können.

6.1.F.3 Die Funktion `freopen()`

Mit der Funktion `freopen()` kann man einen Datenstrom schließen und sofort mit einer neuen Datei öffnen:

Nr.	Konstante	Was ist passiert?
0	—	Kein Fehler
9	EBADF	Datei ist zum Lesen geöffnet worden
122	EDQUOT	Für den Nutzer erlaubte Speichermenge (*Quota*) überschritten
27	EFBIG	Datei zu groß (z. B. beim FAT-Dateisystem)
28	ENOSPC	Kein Platz mehr auf dem Laufwerk

Tabelle 2: Fehlernummern und symbolische Konstanten beim Schreiben

```c
#include <stdio.h>

int main(void)
{
    FILE *dm;
    dm=fopen("datei1.txt","w");
    if(dm==NULL)
    {
        perror("datei1.txt");
        return 0;
    }
    fprintf(dm, "Hallo an die erste Datei\n");

    dm=freopen("datei2.txt","w",dm);
    if(dm==NULL)
    {
        perror("datei2.txt");
        return 0;
    }
    fprintf(dm, "Hallo an die zweite Datei\n");
    fclose(dm);

    return 0;
}
```

Meistens wird das benutzt, um bei einem Programm, das keine Bildschirmausgabe braucht, die Standardausgabe in eine Datei zu leiten:

```c
    freopen("datei.txt", "w", stdout);
```

Nach diesem Funktionsaufruf gehen alle Ausgaben von `printf()` in die Datei `datei.txt`.

Seit C99 gibt es einen weiteren Zweck für `freopen()`: Man kann den Modus eines bestehenden Datenstroms ändern, z. B. vom Schreiben zum Lesen (dazu später). Der erste Parameter des Funktionsaufrufs muss dann NULL sein:

```c
    freopen(NULL, "r", dm);
```

Danach kann mit dem neuen Dateimodus weitergearbeitet werden.

6.1.F.4 Weitere Möglichkeiten für den Dateimodus

Die Funktion `fopen()` ermöglicht Zusatzangaben im Dateimodus:

- `"w+"` und `"a+"`: Man kann aus dem Datenstrom auch lesen

- Auf verschiedenen Betriebssystemen gibt es unterschiedliche Varianten von Textdateien: Bei Linux ist `\n` das Zeilenende, bei Windows ist es `\r\n`, und bei MacOS ist es `\r`. In C dagegen ist das Zeilenende immer nur `\n`. Wie funktioniert das?

  Wenn man unter Windows in einen Datenstrom ein `\n` schreibt, dann sorgen `fprintf()` und seine Verwandten dafür, dass in die Datei ein das dort übliche `\r\n` geschrieben wird. Unter MacOS wird stattdessen das `\r` geschrieben, und unter Linux wird das `\n` unverändert in die Datei geschrieben. Wenn die Datei später gelesen wird, wird das Zeilenende wieder zurück in ein C-übliches `\n` verwandelt.

  Was beim Portieren von Programmen nützlich ist, macht an zwei Stellen Probleme:

  - Das Übertragen von Textdateien von einem System zum anderen ändert leider nicht automatisch die Zeilenenden; Editoren können damit umgehen, bei C-Programmen ist das schwieriger
  - Bei Binärdateien (Bilder, Audio, Filme, Programme) ist es gar nicht schön, wenn plötzlich einzelne Bytes hinzukommen oder verändert werden. Meistens werden sie dadurch unbrauchbar.

  Abhilfe gibt der Modus `"wb"` (und ebenso `"ab"`): Wenn man `\n` in den Datenstrom schreibt, wird in der Datei tatsächlich nur `\n` abgelegt.

  Auf Windows- und MacOS-Systemen ist es meistens sinnvoll, diesen Binär-Modus zu benutzen.

- `"wx"` (seit C11): Die Datei wird exklusiv angelegt: Falls die Datei schon existiert, schlägt der Aufruf fehl. Außerdem kann kein anderer Datenstrom die Datei öffnen, solange dieser Datenstrom nicht geschlossen wurde. Das ist sinnvoll für die Synchronisation verschiedener Prozesse über Sperrdateien (*lock files*).

6.1.F.5 Doppeltes Schließen von Datenströmen vermeiden

Man darf einen Datenstrom immer nur einmal schließen. Dieses Programm ist also falsch:

```c
#include <stdio.h>
int main(void)
{
    FILE *dm;
    dm=fopen("datei.txt", "w");
    if(dm==NULL){perror("");return 0;}
    fputs("hallo",dm);
    fclose(dm);
    fclose(dm); // Fehler, Datei wurde bereits geschlossen
    return 0;
}
```

Beim Ausführen merkt man:

Terminal

```
schueler@debian964:~$ gcc -o doppelfclose doppelfclose.c
schueler@debian964:~$ doppelfclose
free(): double free detected in tcache 2
Abgebrochen
```

Bei einem übersichtlichen Programmaufbau kann es eigentlich nicht passieren, dass man einen Datenstrom zweimal nacheinander schließt. Falls es doch passiert, hilft es, nicht (mehr) benutzte Dateimerker mit NULL zu belegen und vor jedem Aufruf von `fclose` den Dateimerker auf NULL abzufragen:

```c
#include <stdio.h>

int main(void)
{
    FILE *dm = NULL;
    dm=fopen("datei.txt", "w");
    if(dm==NULL){perror(""); return 0;}
    fputs("hallo",dm);
    if(dm!=NULL) fclose(dm); dm=NULL;
    if(dm!=NULL) fclose(dm); dm=NULL;
    return 0;
}
```

6.1.F.6 Fragmentierung

Warum dürfen Dateien im Vergleich zu Arrays beliebig groß sein?

- Ein Array in C belegt immer einen zusammenhängenden Speicherbereich, denn auf jede Adresse wird per Zeigerarithmetik (`a[i]=*(a+i)`) zugegriffen[1].

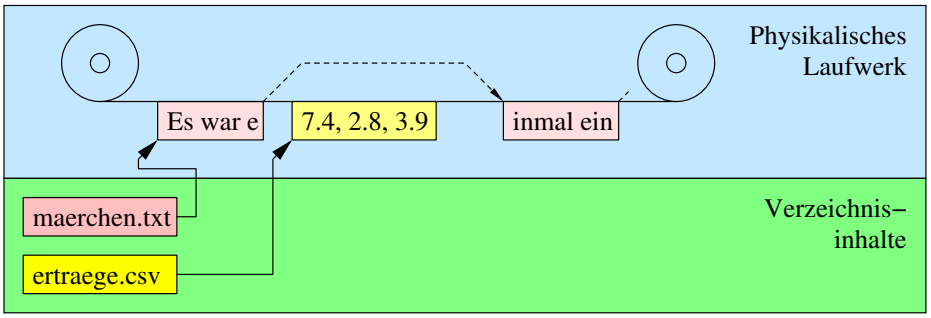

Abbildung 1: Speicherung zweier Dateien mit Fragmentierung

- Bei einer Datei ist es dagegen so, dass der Speicherbereich unterbrochen sein kann (Abbildung 1). Das Betriebssystem sorgt dafür, dass der Schreibzeiger trotzdem an die richtige Stelle gesetzt wird. Wenn auf einem Massenspeicher viele Dateien derart zerstückelt sind, spricht man von *Fragmentierung*. Der Zugriff auf einen solchen Datenträger ist dann verlangsamt. Betriebssysteme, bei denen dies problematisch ist, verfügen über Dienstprogramme, die diese Fragmentierung beseitigen.

[1] Da die heutigen CPUs (ab 32-Bit-CPUs) meistens eine MMU (*memory management unit*) auf dem Chip haben, muss so ein Speicherbereich nicht tatsächlich durchgehend sein.

6.1.F.7 Ergänzende Tabellen oder Bilder

Modus	`"w"`	`"a"`
Datei vorhanden	Alter Inhalt wird überschrieben	Neuer Inhalt wird an alten Inhalt angehängt
Datei fehlt	Datei wird angelegt	Datei wird angelegt

Tabelle 3: fopen mit Modus w oder b

Modus	`"w"`	`"wb"`
`\n` wird geschrieben als	`\n` oder `\r\n` oder `\r`	`\n`
dto. unter Linux	`\n`	`\n`
Textdatei kompatibel zum Editor	+	−
dto. unter Linux	+	+
Datei kompatibel zwischen verschiedenen Systemen	−	+

Tabelle 4: fopen mit Modus w oder wb (genauso a oder ab)