Datei-Eingabe und -Ausgabe

Dateien:

- sind benannte Objekte, die eine Speicherung von Anwendungsdaten auf einem linearen Adressraum zulassen
- unterliegen einer ***persistenten Speicherung***, d.h. Daten bleiben nach Beendigung der Anwendung und auch nach Abschalten des Systems erhalten
- werden im ***Dateisystem verwaltet,*** das ein Teil des Betriebssystems ist. Das Dateisystem sorgt für Zugriff über Verzeichnisse, Dateinamen, für Pufferung, Zugriffsschutz und Speicherplatzverwaltung

Datei-Eingabe und -Ausgabe

Dateien:

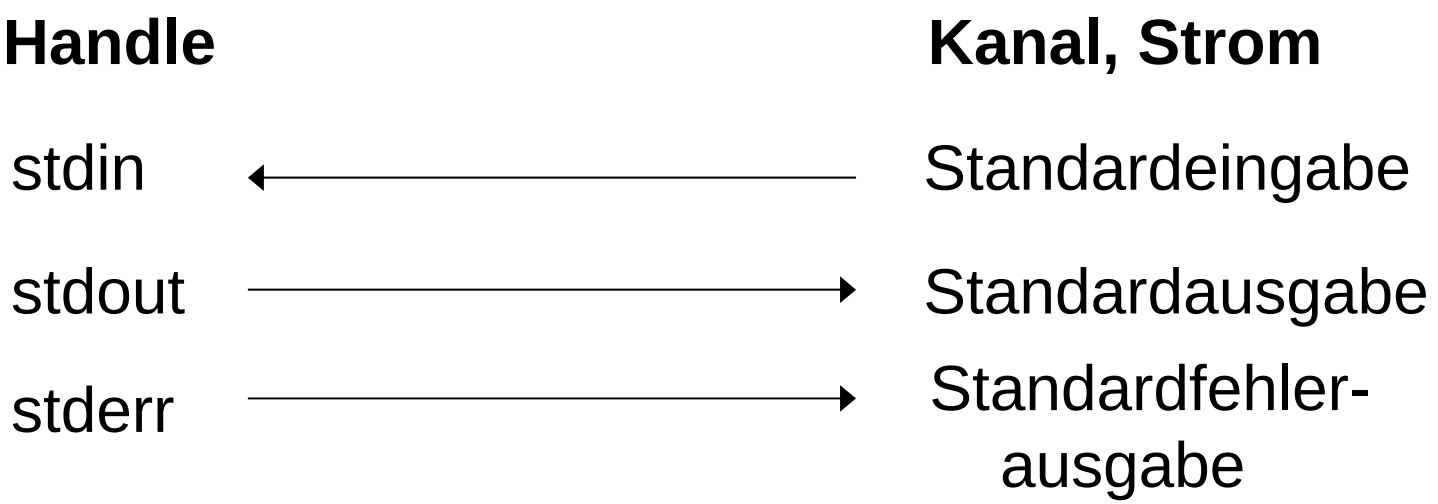
- Dateien können intern beliebig strukturiert sein:
  - Binärdaten (anwendungsspezifische Formate, Bilddateien, Audio-/Video)
  - Textdaten (lesbarer Text, nur die Formatierung ist anwendungsabhängig)
  - XML-Datenformate

- Hinweis auf Typ/Struktur der Datei über Dateityp, z.B. bei bild.jpg wird eine Bilddatei im JPEG-Format angenommen

- Der Zugriff auf Daten in Dateien findet unter vollständiger Kontrolle eines Anwendungsprogramms statt.

Datei-Eingabe und -Ausgabe

In C sind schon eine Reihe von Standarddateien vordefiniert und bei Start des Programms geöffnet.



Diese Standard-Dateien sind für C standardisiert. Auch andere Programmiersprachen (z.B. Java) stellen ähnliche Mechanismen bereit.

Typische Abfolge der Funktionsaufrufe

1. Datei öffnen

Nach Angabe eines Dateinamens und des Zugriffsmodus wird eine Datei geöffnet. Nachfolgende Funktionen können ein Datei-Handle (FILE *f) als „Referenz“ auf die geöffnete Datei benutzen. Es können mehrere Dateien gleichzeitig geöffnet werden.

```
FILE *f;
f = fopen(“C:\\Daten\\meinedatei.txt“, “rt“);
/* jetzt folgen Zugriffe mit Angabe von f */
```

Typische Abfolge der Funktionsaufrufe

2. (wahlweise Positionieren), Zugriffe:
Zugriffe erfolgen standardmäßig von Anfang des Dateiinhalts. Ein s.g. Dateipositionszeiger wird mit jedem Zugriff weitergestellt und aufeinander folgende Zugriffe beziehen sich damit auf fortlaufende Dateiinhalte (sequentielle Organisation, vgl. Abspielen von Musik von CD). Es gibt Funktionen zum expliziten Positionieren.

```
int n;
double z1, z2;

n = fscanf(f, "%lf %lf ",&z1, &z2);
/* weitere Zugriffe */

rewind(f);
```

Typische Abfolge der Funktionsaufrufe

3. Datei schließen:
Soll kein weiterer Zugriff auf die Datei mehr erfolgen, wird die Datei (über die Angabe des Datei-Handle) geschlossen. Damit werden Inhalte und Änderungen für andere Anwendungsprozesse sichtbar.

```
fclose(f);

/* f kann jetzt wieder zum Öffnen einer anderen Datei
   benutzt werden */
```

Zugriff auf Text- vs. Binärdateien

Textdateien, mit lesbarem Text (z.B. im Editor)

- Funktionen für zeichenweise Ein- und Ausgabe (E/A):
  fgetc(), *fputc()* … wird z.B. als ***f**ile **get c**haracter* gelesen
- zeilenweise E/A: *fgets()* , *fputs()*
- formatierte E/A: *fscanf()*, *fprintf()*

Binärdateien, deren Inhalt die interne Repräsentation der Daten enthält, nicht (ohne weiteres) mit einem Editor lesbar

- blockweise und elementweise E/A:
  fread(), fwrite()

Zugriffsmodi beim Öffnen einer Datei

Der bei der fopen()-Funktion angegebene Zugriffsmodus bestimmt, wie eine Datei prinzipiell zu verarbeiten ist.

Zeichen	Datei öffnen zum	Bedeutung
r	Lesen	Fehler, wenn Datei nicht existiert
w	Schreiben	Datei wird neu erzeugt, ggf. überschrieben
a	Anfügen	Datei wird erzeugt, falls sie nicht existiert
r+	Lesen und Schreiben	Fehler, wenn Datei nicht existiert
w+	Schreiben und Lesen	Datei wird neu erzeugt, ggf. überschrieben
a+	Lesen und Anfügen	Datei wird erzeugt, falls sie nicht existiert

Die Zusatzangabe “t“ (text) oder “b“ (binär) wird dem Zugriffsmodus hinzugefügt, z.B. “rt“, “wb“ , “at+“, “rb+“ usw.

Öffnen und Fehlerbehandlung

In C wird eine Datei mit *fopen* bzw. *fclose* geöffnet bzw. wieder geschlossen. Dabei muss das Datei-Handle (FILE *) deklariert sein:

```
#include <stdio.h>
int main()
{  FILE *fp;       //Deklaration Datei-Handle

   fp = fopen("c:\\meinefreunde.txt","rt");    //Pfad-Angabe

   if (fp==NULL) { printf("Fehler"); return -1; }
    ..........
   fclose(fp);
   return 0;
}
```

Kann die Datei nicht geöffnet werden, ist fp mit NULL belegt. Deshalb sollte nach fopen das Dateihandle gegen NULL getestet werden, ehe eine Verarbeitung eingeleitet wird.

Zeichenweises Lesen aus einer Textdatei

In C wird mit *fgetc()* eine Datei zeichenweise gelesen:

```
#include <stdio.h>
void main()
{  char c; FILE *fp;
   fp = fopen("c:\\dat.txt","rt");
   if (fp!=NULL)
   {  c=fgetc(fp);     //Lesen nächstes Zeichen
      if (c==EOF) ... //Prüfen auf EndOfFile
      …
      fclose(fp);

   }
}
```

Nach jeder Leseoperation sollte auf das Erreichen des Dateiendes (EOF) getestet werden.

Zeichenweises Lesen

Typischerweise kann man in C-Programmen beim zeichenweisen Lesen und Verarbeiten einer Datei eine nicht-abweisende Schleife verwenden:

```
#include <stdio.h>
void main()
{   char c; FILE *fp;  fp = fopen("c:\\dat.txt","rt");
    if (fp!=NULL)
    {  do  {
            c=fgetc(fp);     //Lesen nächstes Zeichen
             .....   Verarbeitungsanweisungen
          }  while (c!=EOF) ;   //Prüfen auf End Of File

       if  ( ! fclose(fp))   // ungleich 0 -> Erfolg
           printf("Datei erfolgreich geschlossen");
     }
}
```

Zeichenweises Schreiben

In C wird mit *fputc()* in eine Datei zeichenweise geschrieben:

```
#include <stdio.h>
void main()
{
   char c; FILE *fp;
   fp = fopen("c:\\dat.txt","wt");
   c='A';
   fputc(c,fp);     //Schreiben des Zeichens  c

    ..........
   fclose(fp);
}
```

Zeilenweises Lesen

In C wird mit *fgets()* eine Datei zeilenweise gelesen.
Dabei muss im Programm ein Pufferspeicher als char-Feld vereinbart werden, der die eingelesene Zeile aufnimmt.

Es ist zweckmäßig, einen char-Zeiger zu deklarieren,
der durch fgets() belegt wird und auf den Anfang des Puffers zeigt bzw. auf NULL im Fehlerfall oder bei Erreichen des Dateiendes.

```
char  puffer[81]; char *z;  FILE *fp;

z=fgets(puffer,81,fp);     //Lesen Zeile
```

Zeilenweises Schreiben

In C wird mit *fputs()* eine Datei zeilenweise geschrieben:

```
#include <stdio.h>
void main()
{ char c;
  FILE *fp;
  char  zeile[81]="Das ist eine Zeile!";

  fp = fopen("c:\\dat.txt","wt");

  fputs(zeile, fp);     //Schreiben Zeile
   ..........
   fclose(fp);

}
```

Formatiertes Lesen: fscanf()

In C wird mit *fscanf()* aus einer Datei formatiert gelesen:

```
#include <stdio.h>

struct abteilung{ int nr; char name[20]; char leiter[20];}
int main()
{ FILE *fp;
  int n;
  struct abteilung a;
  fp = fopen("c:\\dat.txt","r");

  n=fscanf(fp,"%d %s %s", &a.nr, a.name, a.leiter);
  /* Rueckkehrwert (n) ist die Anzahl gelesener Obj.
     Rueckkehrwert –1 bei EOF*/
   ..........
}
```

Formatiertes Schreiben: fprintf()

In C wird mit *fprintf()* eine Datei formatiert geschrieben:

```
#include <stdio.h>

struct abteilung {int nr; char name[20]; char leiter[20];};


int main()
{
  FILE *fp;
  int n;
  struct abteilung a={111, "Produktion","Dr. Krause"};
  fp = fopen("c:\\dat.txt","wt");

  n=fprintf(fp,"%d %s %s", a.nr, a.name, a.leiter);
  // Rückkehrwert n ist die Anzahl geschriebener Obj.
  // Rückkehrwert –1 bei EOF
   ...........
}
```

Zugriff auf Binärdateien (1)

Universelle Funktionen *fread()*, *fwrite()*:

Lesen mit *fread*:

```
#include <stdio.h>
int main()
{  FILE *fp;
   …
   fp = fopen("c:\\dat.bin","r");
   n_gelesen=fread(puffer_ptr, laenge, anz_elemente, fp );
   …
}
```

Es werden laenge*anz_elemente Bytes aus der Datei in den Puffer (puffer_ptr) gelesen. Die Anzahl gelesener Elemente wird zurückgegeben.

Zugriff auf Binärdateien (2)

Universelle Funktionen *fread()*, *fwrite()*:

Schreiben mit *fwrite*:

```
#include <stdio.h>
int main()
{  FILE *fp;
    …
   fp = fopen("c:\\dat.bin","w");
   n_geschrieben=fwrite(puffer_ptr, laenge, anz_elemente, fp );
   …
}
```

Es werden laenge*anz_elemente Bytes aus dem Puffer (puffer_ptr) in die Datei geschrieben. Die Anzahl geschriebener Elemente wird zurückgegeben.

Dateiende

Beim Lesen (egal ob zeichen-, zeilenweise, formatiert oder direkt von einer Binärdateien) ist das Erkennen des Dateiendes wichtig.
Die erste Lesefunktion, die nach Erreichen des Dateiendes ausgeführt wird, erzeugt ungültige Inhalte (oder EOF).

Testen mit *feof()*:

```
#include <stdio.h>
int main()
{  int n; …
   FILE *fp = fopen("c:\\dat.bin","rt");

   while (!feof(fp)) {
      n=fscanf(fp,"%d %s %f", &intzahl, name, &floatzahl );
      if (n==3) {
         /* erwartete Anzahl gelesener Werte, jetzt weiterverarbeiten*/
      }

      …
}
```

Positionieren in Dateien (1)

Zurücksetzen einer Datei an den Anfang:

*void rewind(FILE *fp);*

fp ist dabei das Datei-Handle. Die Funktion *rewind()* erzeugt bei Ausführung keinen Rückkehrwert.

Positionieroperation auf eine bestimmte Position:

*int fseek(FILE *fp, long anz, int pos_mode);*

Der Parameter anz gibt die Anzahl der Bytes relativ zu pos_mode an, um die der Dateizeiger bewegt werden soll. Für pos_mode gibt es nur drei Möglichkeiten:

0 für Dateianfang (Konstante SEEK_SET),
1 für aktuelle Position (Konstante SEEK_CURR) und
2 für Dateiende (SEEK_END).

Bei fehlerfreier Ausführung ist der Rückkehrwert 0, sonst ungleich 0.

Positionieren in Dateien (2)

Beispiel:

fseek(fp, -10L, SEEK_CURR);

Der Dateizeiger wird ab der aktuellen Zeigerposition um insgesamt 10 Bytes zurückgesetzt.

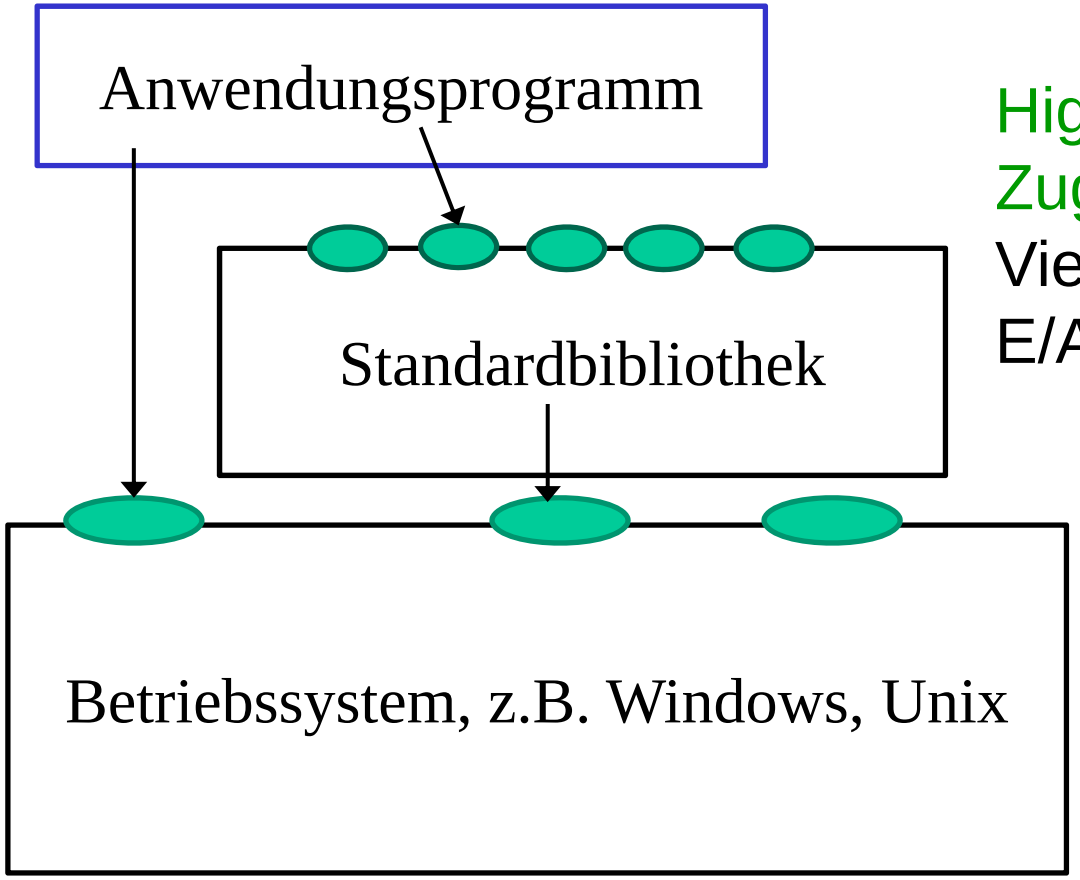
Mit der Funktion

*long ftell(FILE *fp)*

wird der aktuelle Dateizeiger relativ zum Dateianfang (als Anzahl der Bytes) zurückgegeben.
ftell sollte wegen der unterschiedlichen Darstellung des Zeilenendes nur im Binärmodus verwendet werden.

Verschiedene Zugriffs-Ebenen

Unterschied der Zugriffsebenen:



High-Level-Zugriffsfunktionen:
Vielzahl von Funktionen,
E/A-gepuffert, komfortabel

Low-Level-Zugriffsfunktionen:
Minimalumfang,
einfach, schnell,
aber wenig komfortabel

Low-Level-Funktionen hier nicht weiter ausgeführt.

Verschiedene Funktionen für die Dateiarbeit

In C werden für Hilfszwecke rund um die Datei-Verarbeitung eine Reihe von Funktionen angeboten:

Das sind vor allem Funktionen für

- die Arbeit mit Verzeichnissen und Laufwerken
- das Umbenennen und Löschen von Dateien/Ordnern
- das Positionieren des Dateizeigers und
- die Fehlerbehandlung

Funktionen für die Arbeit mit Verzeichnissen (1)

Das aktuelle Laufwerk kann man mit der Funktion

*void _dos_getdrive(unsigned *lw)*

ermitteln.

Beispiel:

```
unsigned d;
_dos_getdrive(&d);          //ermittelt aktuelles LW
printf("Das aktuelle LW ist: %c\n", d + 'A' - 1);
```

Es wird die aktuelle Laufwerks-Nummer zurückgegeben,
A=1, B=2, C=3,...

Nur in Windows möglich! Andere Betriebssysteme benutzen keine Laufwerke.

Funktionen für die Arbeit mit Verzeichnissen (2)

Das aktuelle Laufwerk kann man mit der Funktion

*void _dos_setdrive(unsigned lw,unsigned *anz)*

neu gesetzt werden.

Beispiel:

```
unsigned n;
_dos_setdrive(3,&n);    //setzt aktuelles LW auf C
printf("Es gibt insgesamt %d LW \n",n);
```

Es wird die max. Anzahl von logischen LW-Nummern zurückgegeben.

Funktionen für die Arbeit mit Verzeichnissen (3)

UNIX-Dateisysteme (POSIX Standard)

Ermitteln des aktuellen Arbeitsverzeichnisses:

*char *getcwd(char *buf, size_t size);*

Schreibt das aktuell eingenommene Verzeichnis in die Zeichenkette buf mit maximal size Zeichen.

Bei Erfolg wird der Zeiger buf zurückgegeben, anderenfalls NULL. Die Variable errno zeigt den Grund des Fehlers an, z.B. ERANGE wenn size zu kurz ist.

Setzen:

*int chdir(const char *pfad); // gibt 0 bei Erfolg zurück*

Durchlaufen von Verzeichnissen

Mit den Funktionen

*unsigned _findfirst(const char *pfad, struct _finddata_t *fblk)*

und

*unsigned _findnext(struct _finddata_t *fblk)*

kann der Inhalt eines Verzeichnisses schrittweise gelesen werden.

Der pfad-Parameter ist die Adresse eines Pfades (z.B. c:\temp) ..), der Parameter fblk ist die Adresse einer Strukturvariablen, die dann die Resultate enthält (z. B.Namen der Datei, Größe in Bytes, Änderungsdatum und –zeit).

Durchlaufen von Verzeichnissen

Beispiel:

```
_finddata_t filesfound;
 intptr_t f = _findfirst("C:\\TEMP\\*.*",&filesfound);
 //Alle Dateien in C:\TEMP
 if ( f != -1L ) {
    do {
      if ( !( filesfound.attrib & _A_SUBDIR ) )
      {   //Verzeichnis ausschliessen
         printf("%s %lu\n",filesfound.name,filesfound.size);
      }
    } while ( _findnext(f,&filesfound)==0 ); //Nächste Datei
    _findclose(f); //Handle schliessen
 }
```

Funktionen für die Arbeit mit Verzeichnissen

Das Anlegen (Erzeugen) eines Ordners erfolgt mit der C-Funktion

*int mkdir(cont char *pfad).*

Bei fehlerfreier Ausführung ist der Rückkehrwert 0, sonst –1.

```
char pfad[30]=„c:\\temp\\neu“;
 if (mkdir(pfad))
    printf(“\nFehler!“);
   else
    printf(“\nOrdner erfolgreich erzeugt“);
```

Umbenennen und Löschen von Dateien/Ordnern

Das Umbenennen einer Datei/ eines Ordners erfolgt mit der C-Funktion

*int rename(const char *alt, const char *neu).*

Die Parameter alt und neu können vollständige Pfadangaben enthalten. Beide Pfade müssen gleich sein. Der Zugriff auf die Datei/Ordner muss nach Zugriffsrechten erlaubt sein.
Bei fehlerfreier Ausführung ist der Rückkehrwert 0, sonst –1.

```
char  a[20]="c:\\temp\\x.txt";
char  n[20]="c:\\temp\\y.txt";
// bei einem Ordner "texte" würde der Pfad z.B. sein
//  char  a[20]="c:\\temp\\texte";
 if (! rename(a,n))    printf("\nFehler!");
```

Löschen von Dateien/Ordnern

Das Löschen einer Datei erfolgt mit der C-Funktion
*int remove(const char *name).*

Beispiel:

```
char  datei[20]="c:\\temp\\x.txt";
if (! remove(datei))   printf("\nFehler!");
  else printf("\nDatei erfolgreich gelöscht");
```

Gesperrte Dateien:
Die im Parameter angegebene Datei darf zu diesem Zeitpunkt nicht in Nutzung sein, d.h. muss geschlossen sein. Bei fehlerfreier Ausführung ist der Rückkehrwert 0, sonst –1.