14. Dateien

14.1. Grundlagen

Dateien sind ein allgemeines Konzept für die permanente Speicherung bzw. Ein-/Ausgabe. Dateien bestehen aus beliebig vielen Komponenten eines bestimmten Datentyps. Je nach Dateiart können die einzelnen Komponenten sequentiell oder wahlfrei gelesen werden. Am Ende einer Datei können weitere Komponenten hinzugefügt werden. Die Abbildung der abstrakten Dateistruktur auf reale Speichergeräte (Platte, Band, Drucker, etc.) erfolgt durch das Betriebssystem.

Eine Datei besitzt also lauter Komponenten gleichen Typs. Zusammen mit der Dateivariable wird implizit auch ein Pufferbereich im Arbeitsspeicher definiert, der mindestens eine Dateikomponente (--> aktueller Wert) aufnehmen kann. Auf diesen Datenwert wird dann über die Dateivariable zugegriffen. Zusammen mit dem Typ "Datei" müssen einige Standardoperationen definiert sein, die den Zugriff auf die Datei erlauben:

- Open
  Eröffnen einer Datei zur Inspektion des Inhalts (Lesen) oder zum Anfügen neuer Komponenten am Ende (Schreiben). Mit dieser Prozedur wird nicht nur der Zugriffsmodus festgelegt, sondern gegebenenfalls auch eine leere Datei generiert (beim Modus "Schreiben" bei einer noch nicht vorhandenen Datei).
- Close
  Beenden des Zugriffs auf eine Datei. Das Betriebssystem wird angewiesen, die Datei zu schließen. Damit verbunden sind Verwaltungsvorgänge des Betriebssystems, z. B. Wegschreiben des Inhalts des Pufferbereichs.
- Read, Write
  Lesen und Schreiben von Komponenten einer Datei. Die Arbeitsweise dieser Operationen hängt von der Art der Datei ab. Auf diese Arbeitsweise wird in einem folgenden Kapitel noch näher eingegangen.

14.2. Zugriffsart

Nach Art des Zugriffs auf die Komponenten wird unterschieden in sequentielle Dateien und Dateien mit wahlfreiem Zugriff.

- Sequentielle Dateien (sequential files)

  In den Anfängen der Computertechnik gab es nur sequentielle Dateien, da das Lesen/Schreiben bei Magnetbändern und Lochstreifen nur streng sequentiell möglich war. Auch der Drucker wird über sequentielle Dateivariablen angesprochen. Beim Öffnen einer Datei muss zwischen Lesen und Schreiben unterschieden werden:

  - Lesen
    Der Dateizeiger wird auf die erste Komponente positioniert. Nach dem Zugriff auf eine Komponente wird auf die nächstfolgende Komponente positioniert. Das Ende einer Datei wird durch eine spezielle Standardfunktion (`feof()` oder einen speziellen Wert(`EOF` = End Of File) zurückgemeldet.

- Schreiben
  Der Dateizeiger wird hinter der letzten Komponente positioniert; durch das Schreiben wird die Datei um eine Komponente erweitert. Danach wird wieder hinter die Datei positioniert.

- Dateien mit wahlfreiem Zugriff (random files)

Bei Plattenspeicher kann wahlfrei auf jeden Datenblock der Platte zugegriffen werden. Mit der Verbreitung solcher Speicher lag es nahe, den Datentyp "Datei", um diese Möglichkeit zu erweitern. Bei sequentiellen Dateien wird die Positionierung auf eine Komponente implizit vorgenommen und unterliegt nicht dem Einfluss des Programms. Bei Dateien mit wahlfreiem Zugriff werden die Lese- und Schreiboperationen um die Angabe eines Komponentenindex ergänzt. Der Komponentenindex wird dabei in der Regel von 1 ab aufwärts gezählt. Damit besitzt die Datei mit wahlfreiem Zugriff eine starke Ähnlichkeit mit dem Datentyp "Array". Der Versuch, eine Komponente zu lesen, die "hinter" dem Dateiende liegt, führt zu einem Fehler. Beim Versuch, eine Komponente zu schreiben, deren Index I größer als die Anzahl N der aktuell vorhandenen Komponenten ist, können zwei Fälle unterschieden werden:

- I = N + 1: Die Datei wird um eine Komponente erweitert (wie bei der sequentiellen Datei).
- I > N + 1: Bei einigen Systemen führt der Versuch zu einem Fehler. Bei anderen Implementierungen wird der "Zwischenraum" mit leeren Komponenten aufgefüllt (gefährlich!).

14.3. Grundlegende Abläufe auf Dateien

Der Zugriff auf Laufwerke bzw. allgemein auf die Peripherie erfolgt in C ebenfalls über Zeiger (sog. *Streams* oder *Filepointer*). Diesbezüglich gilt alles als "Datei"; prinzipiell muss eine Datei vor dem Zugriff auf sie geöffnet, und hinterher wieder geschlossen werden. Verwendet werden Funktionen, Datentypen, etc. aus der Standardbibliothek `<stdio.h>`. Wenn in einem Programm nichts weiter vereinbart wird, so wirken die Schreib- und Lesebefehle auf so genannte Standarddateien. Diese sind durch das Betriebssystem vordefiniert.(z. B. Tastatur als Eingabe und Bildschirm als Ausgabe). Will man das ändern, so kann man die Standard- E/A umlenken.

Programmname < Dateiname

Dieses Kommando bewirkt, dass das Programm von der angegebenen Datei liest.

Programmname > Dateiname

Dieses Kommando bewirkt, dass das Programm auf die angegebenen Datei ausgibt.

Die so genannten *Standard-Filepointer* sind immer initialisiert, die zugehörigen Dateien immer geöffnet:

`stdin`	Standardeingabe	(Tastatur)
`stdout`	Standardausgabe	(Bildschirm)
`stderr`	Fehlerausgabe	(Bildschirm!)

Die Unterscheidung zwischen `stdout` und `stderr` ist beispielsweise dann relevant, wenn man die Bildschirmausgabe (`stdout`) in irgendwelche Files umleitet (z. B. mit '>' unter DOS und UNIX), aber verhindert werden soll, dass auch etwaige Fehlermeldungen dorthin "verschwinden".

In einem C Programm kann man die Dateizeiger mit der Funktion freopen umleiten.

FILE *freopen(const char *pfad,const char *modus,FILE *datei);

freopen() wird hauptsächlich dazu benutzt, die Standard-Streams stdin, stdout und stderr zu verbinden. In einfachen Worten: die Streams der Ein-/Ausgabe umzuleiten.

14. 4. Sequentielle Dateien

Will man zusätzlich zu den Standard-Dateien noch weitere Dateien auf der Platte benutzen, so muss man diese Dateien an das Programm anbinden. Alle Befehle zur Dateibearbeitung sind nicht Teil von C. Die Standardbibliotheken enthalten aber viele Funktionen zum Arbeiten mit Dateien.

Im folgenden Beispiel wird eine Datei sequentiell beschrieben und wieder gelesen. An diesem Beispiel werden einige grundlegende Befehle erklärt, weshalb der Quelltext mit Zeilennummern in eckigen Klammern versehen wurde. Die Zeilennummern finden Sie in den Erklärungen unten wieder:

```
[ 1] #include <stdio.h>
[ 2]
[ 3] int main(void)
[ 4]   {
[ 5]   FILE *fp;
[ 6]   int i,xi;
[ 7]   static char dateiname[]="daten.datei";
[ 9]
[10]   /* Beschreiben der Datei */
[11]   fp = fopen(dateiname,"w");
[12]   if ( fp == NULL)
[13]     {
[14]     fprintf(stderr,"Datei %s kann nicht zum Schreiben\
[15]                     geoeffnet werden\n",dateiname);
[16]     exit(1);
[17]     }
[18]   for ( i=0; i<10; i=i+2)
[19]     fprintf(fp,"%d\n",i);
[20]   fclose(fp);
[21]
[22]   /* Lesen der Datei */
[23]   fp = fopen("meine.dat","r");
[24]   if (fp == NULL)
[25]      {
[26]      fprintf(stderr,"Datei %s kann nicht zum Lesen\
[27]                       geoeffnet werden\n",dateiname);
[28]      exit(2);
[29]      }
[30]   while(feof(fp) == 0)
[31]      {
```

```
[32]        fscanf(fp,"%d",&xi);
[33]        printf("%d",xi);
[34]        }
[35]    fclose(fp);
[36]    exit(0);
[37]    }
```

Im Programm finden wir folgende Dateianweisungen:

1. **[5] Definieren einer Datei:**
   `FILE *<Dateizeiger>`
   `FILE` ist eine spezielle Stream-Struktur, der Datentyp für Dateien. Er ist in der Standardbibliothek <*stdio.h*> als Struktur festgelegt, die Informationen über die Datei enthält (z. B. Pufferadresse, Zugriffsrechte u.s.w.).
   Mit obiger Anweisung wird ein Zeiger auf den Datentyp FILE definiert.
2. **[11], [23] Öffnen einer Datei:**
   `<Dateizeiger> = fopen(<Dateiname>,<Zugriffsmodus>);`
   Die Funktion fopen verbindet den externen Namen der Datei mit dem Programm und liefert als Ergebnis den Zeiger auf die Beschreibung der Datei. Im Fehlerfall wird der NULL-Zeiger zurückgeliefert. Die Funktion ist definiert als

   ```
   FILE *fopen(const char *filename, const char *modus)
   ```

   als Zugriffsmodus steht zur Verfügung eine Kombination von "a", "r", "w" und "+":

   - 'r' (Lesen (*read*))
   - 'w' (Schreiben (*write*))
   - 'a' (Anhängen (*append*))
   - 'r+' (Lesen und Schreiben)
   - 'w+' (Schreiben und Lesen)
   - 'a+' (Lesen an bel. Position, Schreiben am Dateiende)

   Durch anhängen eines Zusatzes kann festgelegt werden, ob es sich bei der zu bearbeitenden Datei um eine Binär- oder Textdatei handelt:

   - 't' (für *text*)
   - 'b' (für *binary*)

   Die Funktion fopen reagiert folgendermaßen:

   - Beim Öffnen einer existierenden Datei
     - zum Lesen: keine Probleme
     - zum Anhängen: keine Probleme
     - zum Schreiben: Inhalt der Datei geht verloren
   - Beim Öffnen einer nicht existierenden Datei
     - zum Lesen: Fehler, Ergebnis ist NULL-Zeiger
     - zum Anhängen: neue Datei wird angelegt
     - zum Schreiben: neue Datei wird angelegt

   Maximal `FOPEN_MAX` Dateien können gleichzeitig geöffnet werden, maximale Dateinamenlänge: `FILENAME_MAX`.

Zwischen Lesen und Schreiben ist ein Aufruf von `fflush()` oder ein Positionierungsvorgang nötig.

3. **[14], [19], [26] formatierte Ausgabe auf Datei:**
   `fprintf(<Dateizeiger>, "<Format>",<Werte>);`
   Entspricht der Funktion `printf` und schreibt die angegebenen Werte im angegebenen Format auf die Datei. Dateizeiger verweist auf die Datei, auf die geschrieben wird. `fprintf` ist definiert als

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

4. **[32] formatierte Eingabe von Datei:**
   `fscanf(<Dateizeiger>, "<Format>",<Werte>);`
   Entspricht der Funktion `scanf` und liest die angegebenen Werte im vereinbarten Format der Datei. `fscanf` ist definiert als

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

5. **[30] Dateiende abfragen:**
   `feof(<dateizeiger>);`
   Die Funktion `feof` liefert den Integerwert 1, wenn das Dateiende gelesen wurde, sonst 0.(`int feof(FILE*)`)
6. **[20], [35] Schließen einer Datei:**
   `fclose(<Dateizeiger>);` Die Datei wird geschlossen, vom Programm abgehängt und der Platz für den Filebeschreibungsblock wieder freigegeben. Beim Schreiben auf Dateien sollte die Datei geschlossen werden, sobald alle Schreiboperationen abgschlossen sind, da erst beim Schließen die Dateipuffer auf die Platte geschrieben und die Informationen über die Datei in der Dateiverwaltung aktualisert werden. `fclose` ist definiert als `int fclose(FILE*)`.

14. 5. Textdateien

Dies sind Dateien, deren Komponenten Schriftzeichen sind (Typ char). Sie nehmen eine Schlüsselrolle ein, da die Eingabe- und Ausgabedaten der meisten Computerprogramme Textfiles sind (darunter fällt beispielsweise auch die Druckausgabe). Ein Programm kann vielfach allgemein als eine Datentransformation von einer Textdatei in eine andere aufgefasst werden.

Das Zeilenende - Problem

Nun sind Texte i. a. in Zeilen unterteilt, und es stellt sich die Frage, wie diese Zeilenstruktur auszudrücken ist. In der Regel enthält ein Zeichencode spezielle Steuerzeichen, von denen eines als Zeilenende - Zeichen verwendet werden kann. Bedauerlicherweise verhindert die Realisierung von Textdateien auf Betriebssystemebene eine einfache Realisierung der Textdatei.

- Bei einigen Betriebssystemen wird ein einziges Steuerzeichen als Zeilenende interpretiert (z. B. bei UNIX: Linefeed, Mac: Carriage Return). Das verwendete Zeichen ist jedoch nicht einheitlich festgelegt.
- Bei anderen Systemen besteht das Zeilenende aus zwei Zeichen (in der Regel Carriage Return und Linefeed, z. B. bei MS-DOS).

- Es sind auch Systeme bekannt, die keine Steuerzeichen verwenden.

Diese Unterschiede machen es oft notwendig, Textdateien bei der Übertragung zwischen verschiedenen Rechner- bzw. Betriebssystemen zu konvertieren.

Gepufferte E/A vs. ungepufferte E/A

Wie bei den meisten Betriebssystemen wird auch bei DOS oder UNIX die Ausgabe gepuffert. Das heißt, es wird erst etwas ausgegeben, wenn ein Zeilenende an das Ausgabegerät gesendet wird, oder gar erst, wenn das Programm zu ende ist. Falls Sie das nicht möchten, müssen also dafür sorgen, dass nach jedem Zeichen wirklich auch eine Bildschirmausgabe erfolgt. Sie erreichen dies durch den Funktionsaufruf `fflush(stdout);` nach jeder Ausgabe. Die Funktion `fflush (FILE *datei)` sorgt für sofortiges Wegschreiben des internen Dateipuffers.

Weitere Dateifunktionen für Textdateien

- `int putc(int c, FILE *f)` und `int fputc(int c, FILE *f)`
  Schreiben ein Zeichen in die Datei. Sie arbeiten wie `putchar()`. `putc()` kann ein Makro sein.
- `int getc(FILE *f)` und `int fgetc(FILE *f)`
  Lesen ein Zeichen aus der Datei. Sie arbeiten wie `getchar()`. (`getc()` kann Makro sein)
- `int puts(const char *s)` und `int fputs(const char *s, FILE *f)`
  Schreiben eine Zeichenkette in die Datei. `s` ist ein Zeiger (Verweis) auf die Zeichenkette. Die Funktionen liefern ok (0) oder `EOF`. `puts()` ergänzt die Zeichenkette um ein Zeilenende-Zeichen (newline).
- `char *gets(char *s)`
  Liest eine Zeichenkette aus der Datei. Die Funktion liefert entweder die Zeichenkette `s` oder `NULL`, falls das Dateiende erreicht wurde. Bei einem Lesefehler wird ebenfalls `NULL` zurückgegeben. `gets()` legt statt des Newline-Zeichens (\n) ein Nullbyte am Stringende ab.
- `char *fgets(char *s, int n, FILE *f)`
  Liest eine Zeichenkette aus der Datei. Die Funktion liefert entweder die Zeichenkette `s` oder `NULL`, falls das Dateiende erreicht wurde. Es werden jedoch maximal n-1 Zeichen gelesen. `fgets()` legt ein Nullbyte am Stringende ab.
- `int ungetc(int c, FILE *f)`
  Diese Funktion stellt das Zeichen `c` in den Eingabepuffer zurück. Es kann dann mit dem nächsten `getchar()`, `getc()` oder `fgetc()` wieder gelesen werden. `ungetc())` liefert entweder das Zeichen `c` oder EOF als Status zurück.

14. 6. Binärdateien

Bei binärer Ein- und Ausgabe auf Dateien werden die Daten nicht in "lesbarer" Form abgelegt, sondern die Interndarstellung der Speicherinhalte wird direkt (byteweise) in die Datei übertragen. Binäres Schreiben einer int - Variablen benötigt also stets vier Byte Speicherplatz, wogegen der erforderliche Speicherplatz bei formatierter Ausgabe von der Größe der Zahl bzw. vom Format abhängt.

Die Funktion `fwrite` schreibt eine angegebene Anzahl von Datenelementen gleicher Größe in eine Datei. Übergeben werden muss:

- Die Adresse des ersten Datenelements. Da nicht von vorne herein klar ist, welche Daten geschrieben werden sollen, wird hier ein Zeiger auf void übergeben. Der Zeiger auf die aktuell zu schreibenden Daten kann problemlos in einen solchen gewandelt werden.
- Die Größe eines einzelnen Datenelements. Hierzu gibt es den vordefinierten Typ `size_t`, der normalerweise ein vorzeichenloser Ganzzahltyp ist. Die aktuelle Größe wird normalerweise mit dem `sizeof`-Operator ermittelt.
- Die Anzahl der zu schreibenden Datenelemente
- Die Ausgabedatei

`fwrite` wird also definiert als:

```
size_t fwrite(const void *pt, size_t size, size_t n, FILE *f)
```

Die auszugebenden Daten müssen zusammenhängend im Speicher stehen. Dies ist bei Vektoren stets der Fall, ebenso wie bei Speicherplatz, der durch einen einzelnen `malloc()`-Aufruf (siehe Zeiger) zur Verfügung gestellt wurde.

Beispielprogramm für binäres Schreiben:

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

int main(void)
  {
  int Feld[] = {3, 91, 2134, 6, 33, 267, 9123, -5, 22, 0};
  FILE *pf;
  char dateiname[] = "daten.bin"; /* Dateiname */
  pf = fopen(dateiname, "wb");
  if (pf == NULL)
    {
    printf("Fehler beim Oeffnen der Datei\n");
    exit(1);
    }
  fwrite( (void *)Feld, sizeof(int), 10, pf);
  /* Das Feld wird komplett auf einmal geschrieben */
  fclose(pf);
  return 0;
  }
```

Die Größe der Datei `daten.bin` ist also 10 * sizeof(int).

Die Funktion `fread` ist die zu `fwrite` gehörige analoge Einlesefunktion. Ihr Rückgabewert ist die Anzahl der tatsächlich gelesenen Bytes. Diese Zahl kann kleiner sein, als die Zahl der zu lesenden Bytes, wenn das Dateiende vorzeitig erreicht wurde. SIe ist definiert als:

```
int fread(void *ptr, size_t size, size_t n, FILE *f)
```

14. 7. Dateien mit wahlfreiem Zugriff

Auf die einzelnen Datensätze einer Datei kann direkt zugegriffen werden. Dazu stehen folgende Funktionen zur Verfügung:

`int fseek(FILE *f, long offset, int origin)`

Mit dieser Funktion kann man einen Zeiger auf eine bestimmte Position innerhalb einer Datei setzen
Die Funktion positioniert um einen `offset`, der in Bytes gezählt wird. Der Wert `origin` legt fest, worauf sich `offset` bezieht:

- `SEEK_SET` oder 0:
  Die neue Position ergibt sich aus Dateianfang + Offset.
- `SEEK_END` oder 1:
  Die neue Position ergibt sich aus der aktuellen Position + Offset.
- `SEEK_CUR` oder 2:
  Die neue Position ergibt sich aus Dateiende - Offset.

Der Rückgabewert ist 0, wenn die Positionierung erfolgreich war, sonst -1.

Die Funktion `long ftell(FILE *f)` gibt die aktuelle Position in der Datei an, auf die der Dateizeiger weist. Im Fehlerfall liefert `ftell` den Wert -1.

Die Funktion `void rewind(FILE *f)` positioniert auf den Dateianfang und löscht den Fehlerstatus.

Beispiel: Programm mit wahlfreiem Zugriff auf eine Datei

```
#include <stdio.h>

int main(void)
  {
  long pos;
  int count;
  FILE *fp;
  int mode = 0;
  char c;

  fp = fopen("daten.bin","w+");
  if (fp == NULL)
    {
    printf("Fehler beim Oeffnen der Datei\n");
    exit(1);
    }

  /* Datei beschreiben */
  fputs("abcdefghijklmnopqrstuvwxyz",fp);
  puts("abcdefghijklmnopqrstuvwxyz");
  printf("\n");

  /* Wahlfreier Zugriff auf Datei */
  printf("Eingabe der Position im File (0 bis 25):\n");
```

```
  scanf("%ld",&pos);
  fseek(fp,pos,mode);
  pos = ftell(fp);
  printf("Dateiposition ist %ld\n",pos);
  fread(&c,1,1,fp);
  printf("\nBuchstabe an dieser Position: %c\n\n",c);
  fclose(fp);
  return 0;
  }
```

14. 8. Weitere Dateifunktionen (tabellarisch)

- `FILE *freopen(const char *filename, const char *mode, FILE *stream)` liefert `stream` oder `NULL`. Sie wird oft verwendet zum Umleiten von `stdin`, `stdout` oder `stderr`.
- `int remove(const char *filename)` Löscht eine Datei. Die Funktion liefert `0` oder einen Fehlercode.
- `int rename(const char *oldname, const char *newname)`

- Wenn die letzte Standardeingabefunktion oder -ausgabefunktion eine Fehlerbedingung erzeugt hat, enthält `int errno` (definiert in `<errno.h>`) eine Fehlernummer.
- `int ferror(FILE *f)` liefert bei einem Fehler den Fehlercode oder 0, wenn kein Fehler auftrat.
- `void perror(const char *s)` gibt eine Fehlermeldung aus. Der Aufruf hat die gleiche Wirkung wie: `printf("%s: %s\n",s,strerror(errno))`.
- `void clearerr(FILE *f)` löscht den Fehlerstatus.

14. 9. Verarbeitung von Verzeichnissen

Um ein Verzeichnis auszulesen, wird es zunächst geöffnet (`opendir`), danach werden die Einträge gelesen (`readdir`) und schließlich wird es wieder geschlossen (`closedir`). Analog zum Dateihandle gibt es ein Verzeichnishandle, das ein Zeiger auf den Datentyp DIR ist. Informationen über den Eintrag liefert die von `readdir` gelieferte Struktur `dirent`.

```
#include <sys/types.h>   /* manchmal benoetigt */
#include <dirent.h>      /* Headerfile fuer die Verzeichnis-
Funktionen */

DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
int closedir(DIR *dir);
```

opendir()

Die Funktion `opendir()` erhält als Parameter den Namen des Verzeichnisses als Zeichenkette. Der Rückgabewert ist ein Zeiger auf das Verzeichnishandle. Ein Fehler wird dadurch angezeigt, daß dieser Zeiger den Wert NULL hat.

readdir()

Die Funktion `readdir()` liest den nächsten Eintrag im Verzeichnis und erhält als Rückgabewert einen Zeiger auf eine Struktur `dirent`. Dieser Zeiger verweist auf eine statische Variable, die nur bis zum nächsten `readdir()` gültig ist (wird jedesmal überschrieben). Unterschiedliche DIR-Handles habe aber unterschiedliche Variablen. Die Variable hat unter UNIX folgende Struktur (bei anderen Betriebssystemen kann auch die Struktur anders sein):

```
struct dirent
  {
  long            d_ino;              /* Inode Nummer */
  off_t           d_off;              /* Offset zum naechsten dirent */
  unsigned short  d_reclen;           /* Laenge dieses Eintrags */
  char            d_name[NAME_MAX+1]; /* Dateiname */
  };
```

Für das Anwenderprogramm ist meist nur der Name des Eintrags interessant. Will man mehr über diesen Eintrag erfahren, beispielsweise, ob es wieder ein Verzeichnis ist, verwendet man andere Systemaufrufe, z. B. `stat()`.

closedir()

Zuletzt wird das Verzeichnis mit `closedir()` wieder geschlossen. Ein Beispielprogramm für das Auslesen eines Verzeichnisses sieht so aus:

```
#include <sys/types.h>
#include <dirent.h>

int main(void)
  {
  DIR *dirHandle;
  struct dirent * dirEntry;

  dirHandle = opendir("."); /* oeffne aktuelles Verzeichnis */
  if (dirHandle != NULL)
    {
    while ((dirEntry = readdir(dirHandle)) != NULL)
      {
      puts(dirEntry->d name);
      }
    closedir(dirHandle);
    }
  return(0);
  }
```

rewinddir()

Diese Funktion setzt den Lesezeiger wieder auf den Anfang des Verzeichnisses. Die Syntax des Befehls lautet:

```
void rewinddir(DIR *dir);
```

getcwd()

Die Funktion `getcwd()` ermittelt das aktuelle Arbeitsverzeichnis. Dazu hat das aufrufende Programm einen Puffer für den Namen zur Verfügung zu stellen, der groß genug ist. Die

Größe wird als weiterer Parameter übergeben. Reicht dieser Speicher nicht aus, gibt `getcwd()` NULL zurück.

```
#include <unistd.h>

char * getcwd(char *namebuffer, size t size);
```

In einigen Systemen ist es zulässig, NULL als Parameter für `namebuffer` zu übergeben. Dann alloziiert `getcwd()` selbst den benötigten Speicher und gibt den Zeiger darauf zurück. Die Anwendung muß dann durch einen Aufruf von `free()` den Speicher wieder freigeben.

chdir()

Mit der Funktion `chdir()` wird das aktuelle Arbeitsverzeichnis gewechselt.

```
#include <unistd.h>

int chdir(char *Pfad);
```

Bei Erfolg gibt die Funktion 0, sonst -1 zurück. Die Fehlernummer findet sich in der Variablen `errno`.

Anlegen und Löschen von Verzeichnissen: mkdir(), rmdir()

Die Funktionen zum Anlegen und Löschen der Verzeichnisse heißen wie die entsprechenden UNIX-Befehle. Beim Anlegen werden Zugriffsrechte übergeben. Wie das UNIX-Kommando `rmdir` kann auch die Funktion nur leere Verzeichnisse löschen.

```
#include <fcntl.h>
#include <unistd.h>

int mkdir(char *Pfadname, mode t mode);
int rmdir(char *Pfadname);
```

Bei Erfolg geben die Funktionen 0, ansonsten -1 zurück. Die Fehlernummer findet sich in der Variablen `errno`.