Inhalt

15 Dateiverwaltung, Ein- und Ausgabe

Das Beispiel ***punktListe.c*** wird nochmals behandelt, wobei die Punkte jetzt aus einer Datei gelesen werden sollen.

punkte.c

```
/*
 * Sortieren von Punkten aus Punktdatei
 * fopen, fscanf, fclose
 */
# include <stdio.h>
# include <math.h>
# include <stdlib.h>
...

int main()
{
  FILE *d;                    /* Zeiger auf eine Datei */
  struct Punkt p, *Anfang = NULL, *a;
                                /* Oeffnen einer Datei */
  if(( d = fopen( "Punkte.dat", "r")) == NULL)
  {
    printf("Datei konnte nicht geoeffnet werden!\n");
    return 1;
  }
              /* Einlesen der Punkte aus einer Datei */
  while( fscanf( d, "%f%f", &p.pol.rho, &p.pol.phi)
         != EOF )
  { p.kar = kartesisch( p.pol);
    sortier_ein( &Anfang, p);
  }
                  /* Schließen der geoeffneten Datei */
  fclose( d);
  ...
  return 0;
}

/*
 * Koordinatenumrechnung
 */
struct kartes kartesisch( struct polar p)
{ ... }

/*
 * Sortierfunktion
 * von vorn nach hinten
 */
void sortier_ein( struct Punkt **a, struct Punkt p)
{ ... }
```

Funktions-, Typ- und Makrodeklarationen zur Arbeit mit Dateien sind in der Standardbibliothek <**stdio.h**> festgelegt.

15.1 Grundlagen

- **Dateien im Sinne des Betriebssystems:** Verschiedene Betriebssysteme verwenden einen unterschiedlichen Aufbau der Datei und des Dateiensystems. Verschiedene Geräte können als Dateien betrachtet werden: Tastatur, Bildschirm, Plattenlaufwerke, Drucker.
- **Dateien im Sinne einer Programmiersprachen:** Für ein Programm soll sich dieser Unterschied nicht auswirken. Die Dateien im Sinne einer Programmiersprache müssen überall ein identisches Format vorfinden, denn sie sollen portabel sein, d.h. unabhängig vom Rechner.

Die Ein- und Ausgabe hat die Aufgabe zwischen beiden Arten eine Zuordnung vorzunehmen. Eine evtl. erforderliche Änderung des Formates der Betriebssystemdateien hat somit stets Auswirkungen auf die entsprechenden Standardfunktionen zur Ein- und Ausgabe, nicht aber auf die Dateien im Sinne einer Programmiersprache und damit nicht auf die Programme selbst, die diese nutzen.

Dateien im Sinne der Programmiersprache C (oft auch als **Stream** bezeichnet)**:**

- **Binärdateien:** Geordnete nicht strukturierte Folge von Zeichen, Inhalt eines Speicherbereiches. Binärdateien sind nicht portabel. Deren Interpretation ist vom Rechner abhängig.
- **Textdateien:** Geordnete Folge von Zeichen mit Zeilenstruktur. Eine Zeile kann leer sein. Sie wird von einem Zeilenendezeichen abgeschlossen. Textdateien sind portabel. Bei der Zuordnung auf einen anderen Rechner kann eine Zeichenkonvertierung erforderlich sein.
  C-Standard: max. 254 Zeichen pro Zeile, einschließlich dem Zeilenendezeichen.
  Ein- und Ausgabe müssen aus dem Zeilenformat das Betriebssystemformat erzeugen und umgekehrt (Formatierung). Insbesondere ist eine Codierung der Werte der betrachteten Objekte notwendig, d.h. das Erzeugen der Bitfolge, die den Wert intern repräsentiert.

Ein- und Ausgabe:

- **Ungepuffert:** Die Übertragung eines Zeichens erfolgt sofort.
  Bei **ungepufferter Tastatureingabe** wird das angeschlagene Zeichen direkt dem Programm übergeben. Nachfolgende Korrekturen sind vom Programm selbst zu übernehmen. Das ist programmiertechnisch sehr aufwendig.
  Bei **Bildschirmausgaben** wird jedes Zeichen sofort sichtbar gemacht, also ungepuffert übertragen.
- **Gepuffert:** Die Übertragung eines Zeichens erfolgt über einen Puffer, einen dafür speziell bereitgestellten Speicherbereich im Arbeitsspeicher.
  Für die **gepufferter Tastatureingabe** existiert ein **Eingabepuffer**. Eine Eingabeaufforderung liest zuerst Zeichen aus dem Puffer. Sind dort keine mehr vorhanden, so wird eine andere Funktion aktiviert, welche Zeichen von der Tastatur in den Puffer liest. Erst wenn der Puffer voll ist oder durch das Lesen eines entsprechenden Eingabeabschlusszeichens wird sein Inhalt dem Programm übergeben. Durch diese interne Organisation werden Korrekturen innerhalb des Eingabepuffers möglich. Der Komfort dafür ist unterschiedlich. Bei kleineren Editoren lassen sich wenigstens mittels der Backspace-Taste die letzten eingegebenen Zeichen wieder löschen.

Beim **Schreiben** bzw. **Lesen von der Festplatte** spielt es programmtechnisch keine Rolle, ob die Übertragung über einen Puffer (Cache) erfolgt oder nicht. Die Auslagerung über einen Cache-Speicher ermöglicht aber schnellere Lese- und Schreibzugriffe.

Gepufferte Ein- und Ausgabe:

- **Zeilengepuffert:** Übertragungsart, die nur bei *Textdateien* auftreten kann. Es wird immer genau eine Zeile in den Puffer aufgenommen.
- **Vollständug gepuffert:** Eine bestimmte Anzahl von Zeichen wird übertragen, wobei bei Textdateien die Zeilenstruktur *keine* Rolle spielt.

15.2 Arbeit mit Dateien

15.2.1 Der Datentyp FILE

In der Bibliotheksdatei **<stdio.h>** ist der *Datentyp* **FILE** deklariert. Er umfasst alle für den Zugriff auf eine Datei benötigten Informationen und ist stark implementationsabhängig.

C-Standard für den *Typ* **FILE** sind folgende Angaben gefordert:

- Aktuelle Positionierung innerhalb der Datei. ***_ptr;**
- Zeiger auf den zugehörigen Puffer, falls Zugriff gepuffert. ***_base;**
- Kennung für das Auftreten eines Fehlers oder das Erreichen des Endes. **_flag;**

Beispiel einer Typdefinition:

```
typedef struct {
                  unsigned char *_ptr;
                  int           _cnt;
                  unsigned char *_base;
                  unsigned char *_bufendp;
                  short         _flag;
                  short         _file;
                  ...
                } FILE;
```

- **_ptr:**
  Bei Ein- und Ausgaberoutinen wird die Position, auf die zuletzt zugegriffen wurde, markiert. Eine solche Markierung ist nicht für Geräte nötig, die sowieso sequentiell arbeiten (Tastatur, Bildschirm, Drucker).
  Um die Positionierung einer Datei braucht sich der Programmierer in der Regel nicht zu kümmern. Die Markierung wird bei jedem Zugriff automatisch verändert.
  In Binärdateien kann der Programmierer die Positionierung uneingeschränkt verändern. In Textdateien hingegen hat er nur die Möglichkeit am Anfang oder am Ende die Marke zu setzen.
- **_flag:** In den 8 Bits werden die Zugriffseigenschaften zusammengefasst (0ooo oktal!):

```
#define _IOFBF   0000      /* vollstaendig gepuffert */
#define _IOREAD  0001                /* Lesezugriff */
#define _IOWRT   0002             /* Schreibzugriff */
#define _IONBF   0004                /* ungepuffert */
#define _IOMYBUF 0010           /* Puffer angelegt */
#define _IOEOF   0020         /* Dateiende erreicht */
#define _IOERR   0040         /* Fehler aufgetreten */
#define _IOLBF   0100            /* zeilengepuffert */
#define _IORW    0200    /* Lese- und Schreibzugriff */
```

- **_cnt:** Zählt, falls gepuffert, die noch vorhandenen Zeichen im Puffer.
- **_bufendp:** Zeiger auf das Pufferende.
- **_file:** Dateiidentifikator, wird beim Öffnen zugeordnet.
  Vordefiniert sind drei Standarddateien mit dem Identifikatoren 0, 1 und 2 und drei Zeiger auf diese. Die Zuordnung auf die entsprechenden externen Geräte erfolgt beim Start des Programms automatisch.

Dateiidentifikator	Zeiger	Standarddateien	externen Geräte
0	stdin	Standardeingabedatei	Tastatur
1	stdout	Standardausgabedatei	Bildschirm
2	stderr	Standardfehlerausgabedatei	Bildschirm

In der <**stdio.h**> wird ein Feld von Dateien deklariert und anschließend die ersten drei Komponenten mit den Standarddateien vordefiniert.

```
#define _NFILE 64
extern FILE _iob[_NFILE];
#define stdin  (&_iob[0])
#define stdout (&_iob[1])
#define stderr (&_iob[2])
```

Das Makro **FOPEN_MAX** gibt an, wie viel Dateien gleichzeitig geöffnet werden dürfen.

```
# define FOPEN_MAX _NFILE
```

Bei dynamischer Dateiverwaltung (Dateien werden ein- bzw. wieder ausgelagert) können im Verlauf des Programms parallel höchsten **FOPEN_MAX** Dateien, in der Summe aber mehr Dateien geöffnet werden als Komponenten in **_iob** deklariert wurden. Die Zuordnung an die noch freien oder frei gewordenen Komponenten geschieht automatisch.

15.2.2 Öffnen und Schießen von Dateien

Dateien, auf die zugegriffen werden soll, müssen Dateien im Sinne des Betriebssystems sein, da sie von der Laufzeit des Programms unabhängig sind.
Die Zuordnung einer Betriebssystemdatei und einer Datei im Sinne von C erfolgt durch einen Zeiger vom Typ **FILE** auf den Speicherbereich der Datei mit einer Standardfunktion.

Öffnen von Dateien:

FILE * fopen (const char * *Dateiname* **, const char *** *Zugriff* **);**

Die Funktion baut eine Struktur **FILE** als Komponente des Dateifeldes **_iob[]** auf und liefert einen Zeiger auf diese zurück, welche die Daten der mit *Dateiname* angegebene Datei bereithält. Konnte die angegebene Datei nicht geöffnet werden, so wird der NULL-Zeiger zurückgeliefert. Der angegebene *Zugriff* bestimmt die Zeigerposition (**_ptr**) und die Zugriffsart (**_flag**).

Dateiname:
- Name der zu öffnenden Datei.

Zugriff:

Zugriff	Dateityp	lesen/schreiben (_flag)	Position (_ptr)
r	Text	nur lesen	Anfang
rb	Binär	nur lesen	Anfang
r+	Text	wahlweise	Anfang
r+b	Binär	wahlweise	Anfang
rb+	Binär	wahlweise	Anfang

w	Text	nur schreiben	neu
wb	Binär	nur schreiben	neu
w+	Text	wahlweise	neu
w+b	Binär	wahlweise	neu
wb+	Binär	wahlweise	neu
a	Text	nur schreiben	Ende
ab	Binär	nur schreiben	Ende
a+	Text	wahlweise	Ende
a+b	Binär	wahlweise	Ende
ab+	Binär	wahlweise	Ende

Zugriff mit r*: Datei muss bereits existieren!
Zugriff mit w*, a*: Existiert die angegebene Datei nicht, so wird eine Datei mit diesem Namen angelegt.

Funktionen zur Dateiverwaltung liefern teilweise als Resultat den Wert des Makros **EOF** (end of file) zurück. **EOF** ist in der **<stdio.h>** definiert und hat den Wert **(-1)**. Die Konstante zeigt an, dass das Dateiende erreicht wurde oder auf die Datei nicht zugegriffen werden kann.

```
# ifndef EOF
# define EOF (-1)
# endif
```

Die Zuordnung einer Datei wird durch eine Standardfunktion wieder aufgelöst.

Schließen von Dateien: **int fclose (FILE *** *Dateizeiger* **) ;**
Der Funktionswert ist **0**, falls das Auslagern der Datei fehlerfrei endet, sonst **EOF**.

- Der Zugriff auf die Datei wird aufgehoben, zuvor werden ggf. Pufferinhalte in die Datei zurückübertragen.
- Durch **return** im Hauptprogramm wird der Zugriff auf alle geöffneten Dateien automatisch geschlossen.

Fehlt `return` im Hauptprogramm bzw. bricht es aus anderen Gründen ab, so ist der Zustand der geöffneten Dateien nach dem Programmende nicht definiert!

15.2.3 Verwaltung der Dateipuffer

Beim Öffnen einer Datei wird i.R. automatisch ein Puffer bereitgestellt (**_base**). Will man diesen durch einen eigenen Speicherbereich ersetzen, so stellt C die Funktion **setvbuf** zur Verfügung.

Bereitstellen eines Dateipuffers:

int setvbuf
(FILE * *Dateizeiger* **, char *** *Pufferzeiger* **, int** *Modus* **, size_t** *Groesse* **) ;**
Die Funktion darf erst nach der Dateiöffnung aufgerufen werden und muss aufgerufen werden, bevor der erste Zugriff auf die Datei erfolgt.
Der Funktionswert ist bei ordnungsgemäßer Ausführung **0**, sonst verschieden von **0**.

*Dateizeiger***:**
- Der so definierte Puffer wird der *Datei*, auf die der angegebene *Dateizeiger* zeigt, zugeordnet.

*Pufferzeiger***:**
- Zeiger auf den Speicherbereich für den *Puffer*, wird in **_base** eingetragen.

*Groesse***:**
- Die Größe des Puffers in Byte muss für **_bufend** zusätzlich angegeben werden.

*Modus***:**
- Der Eintrag erfolgt in **_flag**.

_IOFBF	vollständige Pufferung
_IOLBF	Zeilenpufferung
_IONBF	keine Pufferung

Es gibt eine Kurzform **setbuf** der Funktion **setvbuf** mit zwei Anwendungen.

int setbuf (FILE * *Dateizeiger* **, (char *) NULL) ;**
int setbuf (FILE * *Dateizeiger* **, char *** *Pufferzeiger* **) ;**

- Im ersten Fall wird kein *Puffer* zur Verfügung gestellt. Der Pufferzeiger zeigt auf **NULL**.
  Groesse: 0 Byte *Modus*: **_IONBF**

- Im zweiten Fall wird vollständig gepuffert.
  Groesse: **BUFSIZ** *Modus*: **_IOFBF**

```
# define BUFSIZ 4096
```

Ein- und Ausgaben erfolgen i. R. gepuffert. Der Wechsel zwischen beiden kann dann Probleme bringen, wenn eine Datei zum Lesen und Schreiben geöffnet wurde. Da sie mit demselben Puffer arbeiten, muss man evtl. den Puffer zwischen zwei Zugriffen leeren. C stellt hierfür eine Funktionen zur Verfügung.

Leeren eines Puffers: **int fflush (FILE *** *Dateizeiger***) ;**
Der Funktionswert ist **0**, falls die Funktion fehlerfrei endet, sonst **EOF**.

Mit den zwei folgenden Makros wird das Pufferende und der noch verbliebende Platz im Puffer bestimmt.

Bestimmen des Pufferendes: **unsigned char * _bufend (FILE *** *Dateizeiger* **) ;**

```
# define _bufend(p) ((p)->_bufendp)
```

Bestimmen des Platzes im Puffer: **int _bufsiz (FILE *** *Dateizeiger* **) ;**

```
# define _bufsiz(p) (_bufend(p)-(p)->_base)
```

15.3 Standardoperationen zur Ein- und Ausgabe in Textdateien

Für die Ein- und Ausgabe werden in der Bibliotheksdatei **<stdio.h>** zahlreiche Funktionen zur Verfügung gestellt. Sie greifen auf verschiedene Datenträger zu und verarbeiten numerische Werte, Zeichen oder Zeichenfolgen.

15.3.1 Formatierte Eingabe

Die Funktionen haben als Wert **EOF**, falls sofort ein Fehler auftrat, sonst wird die Anzahl der korrekt eingegebenen Werte zurückgegeben.

Datei: **int fscanf (FILE *** *Dateizeiger* **, const char *** *Format* **,** *Zeiger* **, . . .) ;**

*Dateizeiger***:**
- Zeiger auf die Quelldatei, aus der gelesen werden soll.

Standardeingabedatei: **int scanf (const char *** *Format* **,** *Zeiger* **, . . .) ;**

String:
int sscanf (const char * *Stringzeiger* **, const char *** *Format* **,** *Zeiger* **, . . .) ;**

*Stringzeiger***:**
- Zeiger auf die Zeichenkette, aus der gelesen werden soll.

*Zeiger***:**
- Zeiger auf den Speicherbereich, in den der gelesene Wert geschrieben werden soll.

*Format***:**
- Formatierungsstring, gibt die Interpretationsvorschrift für die Eingabewerte an.
- Folge von *Formatbeschreibern.*

Grundaufbau eines Formatbeschreiber:

% *[* ***** *] [Länge] [Typ]* *Kennung*

%: Anfangszeichen eines *Formatbeschreibers.*

***:** Eingabe wird zwar interpretiert aber nicht abgespeichert,
d.h. sie wird übergangen.

*Länge***:** Maximalzahl der zu interpretierenden Zeichen.

*Typ***:** Abweichungen vom *Standardtyp*

h	**short** bei ganzzahligen Werten
l	**long** bei ganzzahligen Werten **double** bei Gleitkommawerten
L	**long double** bei Gleitkommawerten

*Kennung***:** Konvertiertungsvorschrift

Kennung	**erwartete Zeichenfolge**	***Standardtyp***
i d	ganzzahlig dezimal, mit oder ohne Vorzeichen.	**signed int ***
u **o** **x** **X**	ganzzahlig dezimal. ganzzahlig oktal. ganzzahlig hexadezimal. ganzzahlig hexadezimal.	**unsigned int ***
f e E g G	Gleitkommawert, mit oder ohne Vorzeichen, mit oder ohne Dezimalpunkt, mit oder ohne Exponentialteil.	**float ***
c	einzelnes Zeichen oder Zeichenfolge, auch „white space“.	**char ***
s	bel. Zeichenfolge, mit „white space“ beendet, \0 wird automatisch angehängt.	**char ***
[*Zf* **]**	wie **s**, wobei *Zf* die Zeichenfolge der zu akzeptierenden Zeichen ist, bei unzulässigen Zeichen stoppt die Übertragung.	**char ***
[^ *Zf* **]**	wie **s**, wobei *Zf* die Zeichenfolge der nicht zu akzeptierenden Zeichen ist, die Übertragung stoppt.	**char ***

Aus der Eingabezeile, die mit einem **int**-Wert beginnt und beliebige Zeichen anschließen, wird der **int**-Wert gelesen und ausgegeben. Der Zeilenrest wird ignoriert:

```
int  a ;
while( scanf( "%d%*[^\n]", &a) != EOF) printf( "%d", a);
```

Im nächsten Beispiel wird als Zweiadressrechner ein einfacher Taschenrechner simuliert, der die Grundrechenarten für **double**-Werte beherrscht.

rechner.c

```
/*
 * Einfacher Tischrechner
 */
# include <stdio.h>

int main()
{
  double Operand1, Operand2;
  char op[ 2];                /* 2. Komponente für \0 */
  while( scanf( "%lf %1s %lf",
                &Operand1, op, &Operand2) != EOF)
  {
    switch( op[ 0])
    {
      case '+': Operand1 += Operand2; break;
      case '-': Operand1 -= Operand2; break;
      case '*': Operand1 *= Operand2; break;
      case '/': if( Operand2 )
                {
```

```
                    Operand1 /= Operand2; break;
                  }
        default : printf( "FEHLER <RECHNER>"); continue;
      }
      printf( " = %g\n", Operand1);
    }
    return 0;
  }
```

15.3.2 Formatierte Ausgabe

Die Funktionen haben als Wert **EOF**, falls ein Fehler auftrat, sonst wird die Anzahl der korrekt übertragenen Werte zurückgegeben.

Datei:

int fprintf (FILE * *Dateizeiger* **, const char *** *Format* **,** *Ausdruck* **, . . .) ;**

*Dateizeiger***:**
- Zeiger auf die Quelldatei, aus der gelesen werden soll.

Standardausgabedatei: **int printf (const char *** *Format* **,** *Ausdruck* **, . . .) ;**

String:

int sprintf (const char * *Stringzeiger* **, const char *** *Format* **,** *Ausdruck* **, . . .) ;**

*Stringzeiger***:**
- Zeiger auf die Zeichenkette, aus der gelesen werden soll.

*Ausdruck***:**
- Der Wert des *Ausdrucks* wird berechnet und übertragen.

*Format***:**
- Formatierungsstring, gibt die Interpretationsvorschrift für die Ausgabewerte an.
- Zeichenkettenkonstante mit ein oder mehreren *Formatbeschreibern.*

Grundaufbau eines Formatbeschreiber:

% *[Modus] [Länge] [* **.** *Stellen][Typ] Kennung*

%: Anfangszeichen eines *Formatbeschreibers.*

*Modus***:** 4 Steuerzeichen:

-	Linksbündig, sonst rechtsbündig.
+	Positive Vorzeichen werden mitgeschrieben, sonst nicht.
Leerzeichen	Positive Vorzeichen werden durch ein Leerzeichen ersetzt.
#	Oktalzahlen beginnen mit **0**. Hexadezimalzahlen beginnen mit **0x** bzw. **0X**. Gleitkommazahlen besitzen immer einen Punkt. Bei *Kennung* **g**, **G** werden Nullen am Ende nicht unterdrückt.

Länge: Mindestzahl der zu erzeugenden Zeichen, evtl. mit Leerzeichen aufgefüllt. Die Angabe wird ignoriert, falls mehr Zeichen ausgegeben werden.

Stellen: Von der jeweiligen *Kennung* abhängig:

i d u o x X	Mindestzahl der Ziffern, evtl. mit Nullen aufgefüllt.
f e E g G	Stellen hinter dem Komma.
c s	Höchstzahl der Zeichen.

Typ: Abweichungen vom Standardtyp

h	**short** bei ganzzahligen Werten
l	**long** bei ganzzahligen Werten
L	**long** bei Gleitkommawerten

(**float**-Werte werden beim Aufruf in **double** umgewandelt.)

Kennung: Konvertiertungsvorschrift

Kennung	**Darstellung**	***Standardtyp***
i d	ganzzahlig dezimal, mit oder ohne Vorzeichen.	**signed int**
u	ganzzahlig dezimal.	**unsigned int**
o	ganzzahlig oktal.	
x	ganzzahlig hexadezimal, mit **x**.	
X	ganzzahlig hexadezimal, mit **X**.	
f	exponentenfrei mit oder ohne Dezimalpunkt, gerundet.	**double**
e	halblogarithmisch, mit **e**, gerundet.	
E	halblogarithmisch, mit **E**, gerundet.	
g	je nach Größe wie **f** oder **g**, Dezimalpunkt und Nullen werden ggf. weggelassen.	
G	je nach Größe wie **f** oder **G**, Dezimalpunkt und Nullen werden ggf. weggelassen.	
c	einzelnes Zeichen	**char**
s	Zeichenfolge	**char ***

format.c

```
/*
 * Formatbeschreiber
 */

#include <stdio.h>

int main()
{
  char S[] = "String"; int I = 1234; float G = 9.87;

  printf( "%s\n", S);                /* String       */
  printf( "%3s\n", S);               /* String       */
  printf( "%.3s\n", S);              /* Str          */
  printf( "%8s\n", S);               /* __String     */
  printf( "%8.3s\n", S);             /* _____Str     */
  printf( "%-8s\n", S);              /* String__     */
  printf( "%-8.3s\n", S);            /* Str_____     */
```

```
    printf( "%d\n", I);                 /* 1234          */
    printf( "%2d\n", I);                /* 1234          */
    printf( "%6d\n", I);                /* __1234        */
    printf( "%.7d\n", I);               /* 0001234       */
    printf( "%6.2d\n", I);              /* __1234        */
    printf( "%-6d\n", I);               /* 1234__        */
    printf( "%#x\n", I);                /* 0x4d2         */

    printf( "%f\n", G);                 /* 9.870000      */
    printf( "%e\n", G);                 /* 9.870000e+00  */
    printf( "%g\n", G);                 /* 9.87          */
    printf( "%10.1f\n", G);             /* _______9.9    */
    printf( "%+5.0E\n", G);             /* +1E+01        */

    return;
  }
```

Das nächste Beispiel kopiert zeichenweise eine Datei in eine andere.

kopiere1.c

```
  /*
   * Dateiverwaltung: Kopieren einer Datei
   * scanf(), fscanf(), printf(), fprintf()
   */

  # include <stdio.h>
  void WEITER();

  int main( int i, char *z[])
  {
    FILE *d1, *d2; char zeich; int j;

    if( i < 3)
    {
      printf( "EINGABE: Kopiere1 <dat1> <dat2>");
      WEITER(); return 1;/* Abbruch wegen Syntaxfehler */
    }

    if(( d1 = fopen( z[ 1], "r")) == NULL)
    {
      printf
      ( "FEHLER: Datei %s konnte nicht eroeffnet werden",
              z[ 1]);
      WEITER(); return 2;
    }

    if(( d2 = fopen( z[ 2], "w")) == NULL)
    {
      printf
      ( "FEHLER: Datei %s konnte nicht eroeffnet werden",
              z[ 2]);
      WEITER(); return 3;
```

```
  }

  j = 0;
              /* zeichenweises Lesen aus einer Datei */
  while( fscanf( d1, "%c", &zeich) != EOF)
  {
    if(( zeich == '\x0A') && !( ++j % 23)) WEITER();
            /* zeichenweises Schreiben am Bildschirm */
    printf( "%c", zeich);
            /* zeichenweises Schreiben in eine Datei */
    fprintf( d2, "%c", zeich);
  }
  printf( "\n");
  fclose( d1);
  fclose( d2);

  return 0;
}

void WEITER()                        /* Seitensteuerung */
{
  printf( "   Weiter mit <ENTER>");
             /* Lesen eines Zeichen von der Tastatur */
  getchar();
}
```

15.3.3 Ein- und Ausgabe von Zeichen und Zeichenfolgen

Zeichenweise Eingabe

Die Funktionen lesen das aktuelle Zeichen aus der Datei, auf die der *Dateizeiger* zeigt, und verschieben danach die aktuelle Position in der Datei um das Zeichen weiter.
Funktionswert: Aktuelles Zeichen, **EOF** bei Fehlern oder Dateiende.

Datei: **int fgetc (FILE * *Dateizeiger*) ;**

int getc (FILE * *Dateizeiger*) ;

Als Makro realisiert: Wenn der Puffer leer ist, so wird die Funktion **_filbuf** zum Füllen des Puffers aufgerufen, sonst wird das nächste Zeichen gelesen und die Positionierung weitergezählt(schneller).

```
# define getc(p) (--(p)->_cnt<0?\
                  _filbuf(p):(int)*(p)->_ptr++)
```

Standardeingabedatei **int getchar (void) ;**

Als Makro realisiert: Es wird von der Standardeingabedatei gelesen.

```
#define getchar() getc(stdin)
```

Zeichenweise Ausgabe

Die Funktionen schreiben das angegebene *Zeichen* auf die aktuelle Position der Datei, auf die der *Dateizeiger* zeigt und verschieben danach in der Datei die aktuelle Position um das Zeichen weiter.
Funktionswert: Aktuelles Zeichen, **EOF** bei Fehlern.

Datei: **int fputc (int** *Zeichen* **, FILE *** *Dateizeiger* **) ;**

int putc (int *Zeichen* **, FILE *** *Dateizeiger* **) ;**

Als Makro realisiert: Wenn der Puffer voll ist, so wird die Funktion **_flsbuf** zum Übertragen des Pufferinhalt in die zugeordnete Datei aufgerufen, sonst wird das *Zeichen* in den Puffer geschrieben und die aktuelle Position weitergezählt.

```
# define putc(x, p) (--(p)->_cnt<0?\
                _flsbuf((unsigned char)(x),(p)):\
                (int)(*(p)->_ptr++=(unsigned char)(x)))
```

Standardausgabedatei: **int putchar (int** *Zeichen* **) ;**

Als Makro realisiert: Es wird von der Standardausgabedatei gelesen.

```
# define putchar(x) putc((x), stdout)
```

Zeilenweise Eingabe

Die Funktionen lesen eine Zeichenfolge aus der Datei, auf die der *Dateizeiger* zeigt, in den angegebenen *String* und verschieben danach die aktuelle Position in der Datei um diese eingelesenen Zeichen weiter. Hinter dem letzten übertragenen Zeichen wird **\0** gesetzt.
Funktionswert: Zeiger auf *String*, **NULL**-Zeiger bei Fehlern.

Datei: **char * fgets (char *** *String* **, int** *Anzahl* **, FILE *** *Dateizeiger* **) ;**

Es wird die aktuelle Zeile, höchstens aber werden *Anzahl*-1 Zeichen gelesen.

Standardeingabedatei: **char * gets (char *** *String* **) ;**

Es wird genau eine Zeile von der Standardeingabedatei gelesen.

Zeilenweise Ausgabe

Die Funktionen schreiben den angegebenen *String* auf die aktuelle Position der Datei, auf die der *Dateizeiger* zeigt, und verschieben danach die aktuelle Position in der Datei um den *String* weiter.
Funktionswert: Anzahl der übertragenen Zeichen, **EOF** bei Fehlern.

Datei: **int fputs (const char *** *String* **, FILE *** *Dateizeiger* **) ;**

\0 wird *nicht* mit übertragen.

Standardausgabedatei: **int puts (const char *** *String* **) ;**

Es wird stets in die Standardausgabedatei geschrieben. **\0** wird mit übertragen.

Weitere Funktionen

Mit der folgenden Anweisung setzt man die Positionierung der Datei auf den Dateianfang zurück und löscht das Fehlerbit.

Positionierung der Datei: **void rewind (FILE *** *Dateizeiger* **);**

Das nächste Makro prüft, ob das Dateiende erreicht wurde.
Dateiende: **int feof (FILE *** *Dateizeiger* **);**

```
# define feof(p) ((p)->_flag&_IOEOF)
```

Das folgende Makro prüft, ob Fehler aufgetreten sind.
Fehlerbit: **int ferror (FILE *** *Dateizeiger* **);**

```
# define ferror(p) ((p)->_flag&_IOERR)
```

Das letzte Makro löscht Fehler- und Dateiende.
Löschen der Bits: **void clearerr (FILE *** *Dateizeiger* **);**

```
# define clearerr(p) ((void)((p)->_flag&=~(_IOERR|_IOEOF)))
```

Dieses Programm kopiert eine Datei zeichenweise entweder auf den Bildschirm oder/und in eine Datei.

kopiere2.c

```
/*
 * Dateiverwaltung: Kopieren einer Datei
 * stdin, stdout
 * rewind(), feof()
 * fgetc(), fputc(), getchar(), putchar()
 */

# include <stdio.h>

void WEITER();

/* Kopierunterprogramme */
void copy( FILE *, FILE *);
void page( FILE *);

int main( int i, char *z[])
{
  FILE *d1, *d2;

  if( i == 1 )
  {
    printf
      ( "EINGABE von der Tastatur/Abbruch mit ^D\n");
    printf( "AUSGABE am Bildschirm.\n");
    WEITER();
    page( stdin);
    return 0;
  }
  else
  if(( d1 = fopen( z[ 1], "r")) == NULL)
  {
```

```
    printf( "FEHLER: ");
    printf( "%s konnte nicht eroeffnet werden!\n",
            z[ 1]);
    return 1;
  }

  if( i == 2 )
  {
    printf( "Ausgabe am Bildschirm:\n\n");
    page( d1);
    fclose( d1);
    return 0;
  }
  else
  if(( d2 = fopen( z[ 2], "w+")) == NULL)
  {
    printf( "FEHLER: ");
    printf( "Datei %s konnte nicht angelegt werden!\n",
            z[ 2]);
    return 2;
  }

  copy( d1, d2);

  /* Kontrollausdruck */
  printf( "KOPIERT wurde von \"%s\" nach \"%s\".\n",
          z[ 1], z[ 2]);
  printf( "Ausgabe der Datei \"%s\" am Bildschirm.\n",
          z[ 2]);

  WEITER();
  rewind( d2);      /* Positionierung auf den Anfang */
  page( d2);                /* Ausgabe am Bildschirm */

  fclose( d1);
  fclose( d2);

  return 0;
}

void WEITER()
{
  printf( "   Weiter mit <ENTER>"); getchar(); return;
}

void copy( FILE *f1, FILE *f2)     /* Kopierfunktion */

  char zeich;

  while( 1)
  {
    zeich = fgetc( f1);
    if( feof( f1)) break;
    fputc( zeich, f2);
```

```
    }
    printf( "\n");

    return;
  }

  void page( FILE *f1)/* seitenweise Bildschirmausgabe */
  {
    int j = 0; char zeich;

    while( 1)
    {
      zeich = fgetc( f1);
      if( feof( f1)) break;
      if(( zeich == '\x0A') && !( ++j % 23)) WEITER();
      putchar( zeich);
    }
    printf( "\n");

    return;
  }
```

15.4 Standardoperationen zur Ein- und Ausgabe in Binärdateien

Die Funktionen zum Lesen und Schreiben auf Binärdateien arbeiten blockweise. Speicherbereiche der angegebenen Größe und Datenstruktur werden übertragen, die aktuelle Position in der Datei um die übertragenen Zeichen weitergestellt.
Funktionswert: Anzahl der tatsächlich fehlerfrei übertragenen Datenstrukturen.
Ist der Funktionswert kleiner als die Anzahl der zu übertragenden Datenstrukturen, so ist das Dateiende erreicht oder es sind Fehler aufgetreten.

Eingabe:

size_t fread
(void * *Daten* **, size_t** *Groesse* **, size_t** *Anzahl* **, FILE *** *Dateizeiger* **) ;**

Ausgabe:

size_t fwrite
(const void * *Daten* **, size_t** *Groesse* **, size_t** *Anzahl* **, FILE *** *Dateizeiger***);**

*Daten***:**
- Zeiger auf den Anfang eines Feldes, dessen Komponenten zu lesen bzw. zu schreiben sind.

*Groesse***:**
- Größe der einzelnen Feldkomponenten, wie sie durch den Operator **sizeof()** geliefert wird.

*Anzahl***:**
- Anzahl der zu lesenden bzw. zu schreibenden Feldkomponenten.

*Dateizeiger***:**
- Zeiger auf die Datei, auf die zugegriffen werden soll.

Das folgende Beispiel kopiert nur die erste Zeile und von der zweiten die ersten beiden Zeichen.

puffer.c

```
/*
 * Dateiverwaltung: Kopieren einer Datei
 * setbuf()
 * gets(), fgets(), fputs(), fread(), fwrite()
 */

# include <stdio.h>
# define MAX 512

int main()
{
  FILE *d1, *d2;
  char dat_name[ 64], buf[ MAX];

  printf( "von Datei:\n");
              /* Einlesen des Dateinamens als String */
```

```
    gets( dat_name);

    if(( d1 = fopen( dat_name, "r")) == NULL)
    {
      printf( "FEHLER: ");
      printf( "Datei %s konnte nicht eroeffnet werden\n",
              dat_name);
      return 1;
    }

    printf( "nach Datei:\n");
                /* Einlesen des Dateinamens als String */
    gets( dat_name);
    if (( d2 = fopen( dat_name, "w")) == NULL)
    {
      printf( "FEHLER: ");
      printf( "Datei %s konnte nicht eroeffnet werden\n",
              dat_name);
      return 1;
    }

    setbuf( d1, NULL);        /* ungepufferter Zugriff */
    setbuf( d2, NULL);
                      /* Die erste Zeile wird kopiert. */
    fgets( buf, MAX, d1);       /* Zugriff über string */
    fputs( buf, d2);
               /* Zwei weitere Zeichen werden kopiert. */
    fread( buf, 2, sizeof( char), d1);
    fwrite( buf, 2, sizeof( char), d2);

    fclose( d1);
    fclose( d2);

    return 0;
  }
```

Weitere Funktionen zur Positionierung:
Außer der bereits erwähnten Funktion **rewind,** mit der man die Positionierung auf den Dateianfang zurücksetzt und das Fehlerbit löscht, gibt es weitere Funktionen zur Positionierung, auf die hier nicht eingegangen werden soll (s. **man**):
fgetpos, **ftell:** Liefert die aktuelle Positionierung einer Datei.
fsetpos: Stellt die aktuelle Positionierung, die mit **fgetpos** geliefert wurde, wieder her.
fseek: Positionierung relativ zum Dateianfang, Dateiende oder zur aktuellen Positionierung, bei Textdateien nur mit Einschränkung anwendbar.

15.5 Verwaltung von Betriebssystemdateien

Die letzten beiden Funktionen beziehen sich auf Dateien im Sinne des Betriebssystem und haben nur indirekt mit der Ein- und Ausgabe zu tun. Die angesprochenen Dateien müssen geschlossen sein.

Löschen einer Datei: **int remove (const char *** *Dateiname* **) ;**

Umbenennen einer Datei:

int rename (const char * *alter_Name* **, const char *** *neuer_Name* **) ;**

datVerw.c

```
/*
 * Verwaltung von Betriebssystem-Dateien
 */

#include <stdio.h>

int main()
{
  ...

  rename( "f1.c", "f2.c");                /* Umbennen */
  remove( "f2.c");                        /* Loeschen */

  return 0;
}
```