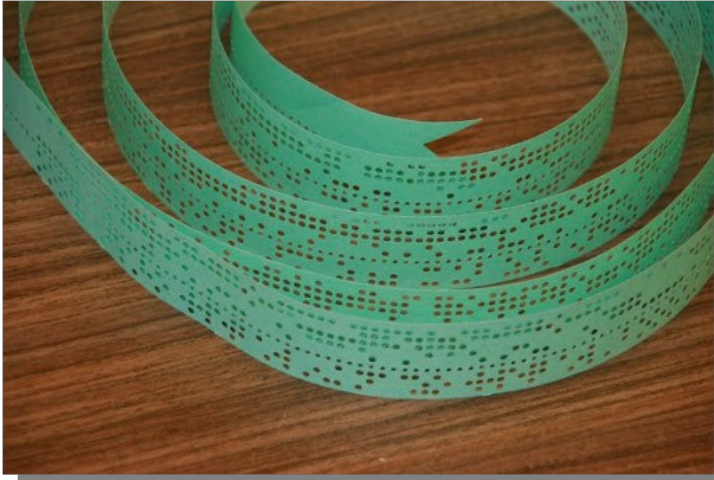
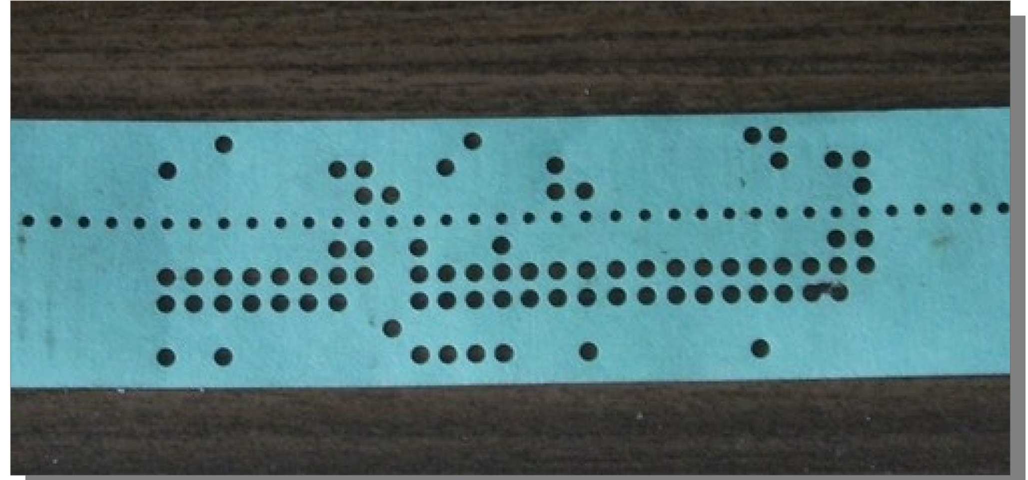
Dateiarbeit in C

- Datei(engl. File) ist ein Menge von Bytes auf einem geeigneten Datenträger.
  - Festplatte
  - USB-Stick
  - Früher: Magnetband, Lochband, Lochkartenstapel, Diskette
  - https://www.youtube.com/watch?v=xqlY0QLmVtw
- Eine Datei enthält Daten und ist in der Regel über ein Verzeichnis und einen Namen identifizierbar.

Lochband





Auf Lochbändern wird das Wesen von Dateien sehr gut deutlich. Jede Reihe (vertikal) kodiert hier ein Textzeichen.
Beim Schreiben von Daten wird Zeichen für Zeichen als Lochreihe in den Streifen gestanzt.
Beim Lesen der Daten wird Reihe für Reihe gelesen und dabei der Lochsteifen durch die Leseeinrichtung bewegt.

Bilder: Mit freundlicher Genehmigung von http://www.robotron-computermuseum.efb-1.de

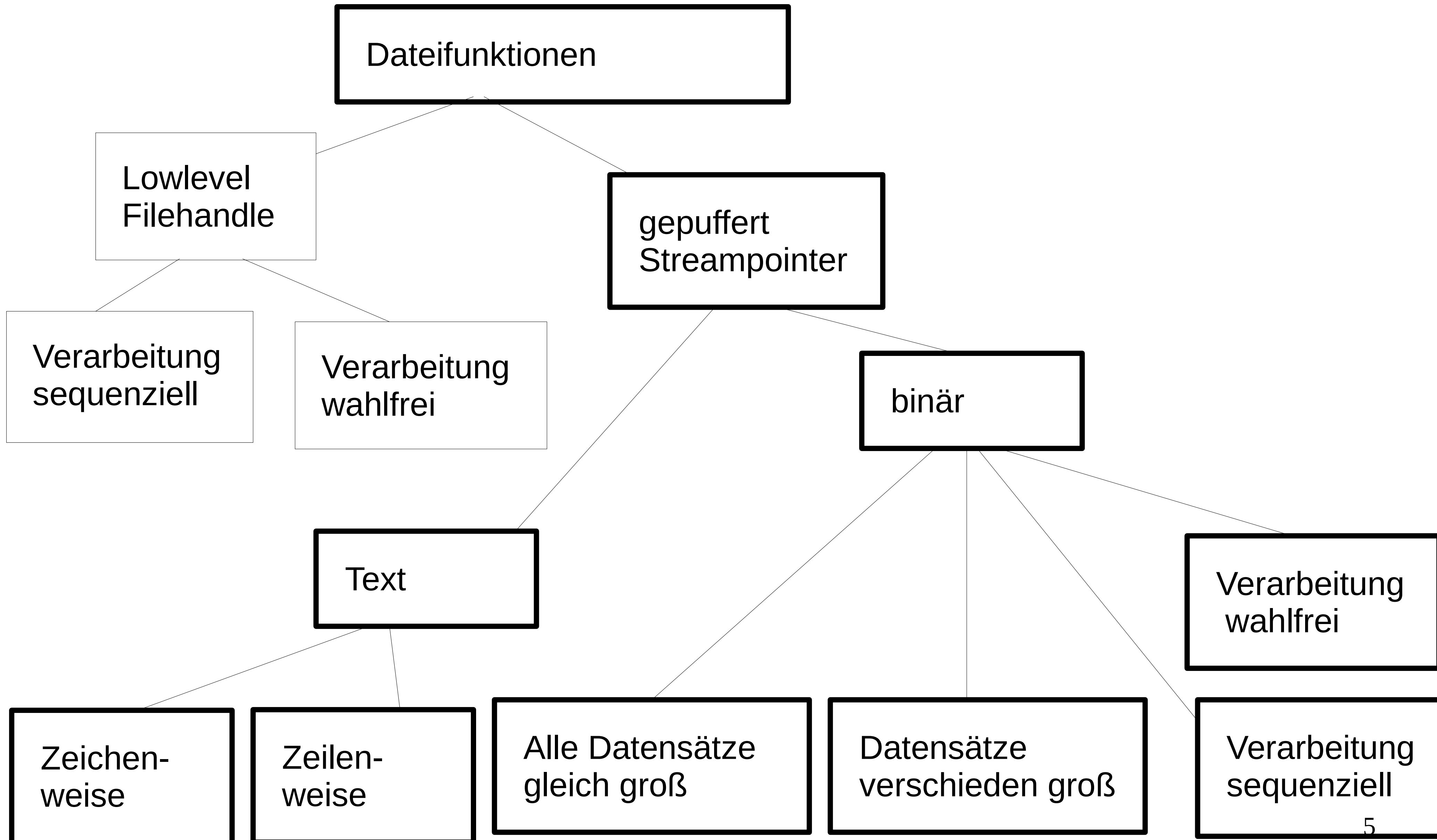
Datenströme

- Bei der Verarbeitung der Daten einer Datei spricht man vom Datenstrom (Stream (I/O)
- Werden die Daten vom Datenträger in den programminternen Speicher gelesen, so spricht man von Eingabestrom.
- Werden die Daten auf den Datenträger aus dem programminternen Speicher geschrieben, so spricht man von Ausgabestrom.

high level I/O/ low level I/O

- Es wird zwischen high level I/O und low level I/O unterschieden.
- Low level I/O arbeitet sehr systemnah, die Verbindung zur Datei wird über einen Integerwert, den sog. Filedeskriptor oder Filehandle(Win) hergestellt.
- High level I/O arbeitet mit einem Streampointer, einem Pointer vom Typ FILE*. High level I/O arbeitet gepuffert.
- Im Weiteren wird zunächst high level I/O betrachtet.

Dateiarbeit in C



Öffnen von Dateien

- Dateien liegen auf Datenträgern.
- Um mit Dateien zu arbeiten, muss zunächst über das Betriebssystem eine Verbindung zwischen unserem Programm und den Daten auf dem Datenträger hergestellt werden.
- Das geschieht, in dem eine Datei geöffnet wird (fopen).
- Zum Öffnen einer Datei gibt man ihren Namen inclusive Dateipfad im Dateisystem und einen Arbeitsmodus, der besagt, ob und in welcher Weise Daten gelesen oder geschrieben werden sollen, an.

Gepuffertes Lesen/Schreiben

- Die Funktionen zur Dateiarbeit arbeiten gepuffert um höhere Geschwindigkeiten zu erreichen.
- Beim Lesen werden nicht nur die gerade angefor-derten Daten gelesen, sondern es wird ein größe-rer Datenblock in einen Zwischenspeicher (Puffer) übertragen. Weitere Leseoperationen beziehen ihre Daten dann aus diesem Puffer. Wenn der Puffer bis zum Ende gelesen worden ist, erfolgt das nächste Lesen wieder vom Datenträger.

Gepuffertes Lesen/Schreiben

- Beim Schreiben verhält es sich ähnlich.
- Zu schreibende Daten werden zunächst in einen Puffer geschrieben. Der Pufferinhalt wird dann in die Datei übertragen wenn
  - der Puffer voll ist.
  - wenn ein \n ausgegeben wird.
  - Wenn die Datei geschlossen wird (fclose):
- Stürzt ein Programm ab, kann es sein, dass Daten zwar in den Puffer geschrieben, aber noch nicht in die Datei übertragen worden sind.

Schließen von Dateien

- Ein Programm kann nur eine plattformabhängig endliche Zahl von Dateien gleichzeitig offen haben.
- Deshalb müssen Dateien, wenn nicht mehr geschrieben/gelesen wird, wieder geschlossen werden (fclose).
- Dabei werden ggf. Pufferinhalte noch in die Datei übnertragen und Speichbereiche, die sich hinter dem Streampointer vom Typ FILE* verbergen, frei gegeben .
- Beim Verlassen eines Programms auf dem PC (Prozess endet) werden offene Dateien automatisch geschlossen.

Datei Öffen mit fopen

Dateiname Incl. Pfad

Modus

```
FILE * pf;
pf=fopen(“myFile.txt“,“rt“);
```

Bei Microsoft wird als Zeilenende 0x0d 0x0a verwendet. Unter Microsoft werden bei Textdateien diese beiden Bytes zu \n zusammengefasst. Es werden zwei Bytes aus der Datei gelesen, aber nur \n zurückgegeben.
Bei anderen Systemen ist diese Unterscheidung wirkungslos.

b/t	Texdatei ([t])	Binärdatei(b)
r	Zum Lesen öffen	
w	Zum Scheiben (überschreiben) öffnen, ggf. erzeugen	
a	Zum Schreiben am Dateiende öffnen, ggf. erzeugen	
r+	Zum Lesen und Schreiben öffnen (ändern)	
w+	Zum Lesen und Schreiben öffen (überschreiben), ggf. erzeugen	
a+	Zum Anfügen, Lesen, Erzeugen öffnen, ggf. erzeugen	

Der Modus wird als Zeichenkette angegeben und besteht aus zwei Zeichen. Das erste Zeichen gibt den eigentlcihen Lese-/Schreibmodus an, das zweite Zeichen ob es sich um Binär- der Textdaten handelt (siehe Textblase oben).

Lesen / Schreiben von Bytes

Funktion	Verwendung
`int fgetc(FILE* stream);`	Lesen des nächsten Zei-chens (Bytes) als un-signed char oder EOF
`int fputc(int c, FILE* stream);`	Schreibt das Zeichen c oder Byte in den Stream

EOF hat den Wert -1.
Die Funktion fgetc liest ein Byte aus dem Datenstrom. Es kann Werte zw. 0..255 annehmen. Im Bereich von 8 bit hat der vorzeichenlose Wert 255 und der Wert -1 die selbe Bitfolge. Deshalb ist der Returnwert von fgetc vom Typ int.
Hier unterscheiden sich auf Grund der größeren Verabeitungsbreite -1 und 255.

Lesen und Schreiben von **Text**

Funktion	Verwendung
`int fgets(char*s,int n,FILE*stream);`	Lesen von max. n-1 Zeichen. Das Lesen wird vorher beendet, wenn ein Zeilentrenner gefunden wird, dieser wird mit eingelesen. Eine terminierende 0 wird angefügt.
`int fputs(const char*s, FILE* stream);`	Schreibt die nullterminierte Zeichenkette + ein \n in den Stream.
`int fprintf(FILE* stream, const char* fmt, ...)`	

Lesen und Schreiben von Binärdaten

Funktion	Verwendung

`size_t fread(void* ptr,`
`size_t size,`
`size_t n`
`FILE* stream)`
	Liest aus dem Stream max. n Objekte der Größe size und speichert sie auf die Adresse ptr. Es muß genügend Speicher zur Verfügung stehen, um die Daten aufzunehmen. Der Returnwert liefert die Anzahl der gelesenen Objekte. (eof, oder ferror liefern Fehlerzustand)

`size_t fwrite(void* ptr,`
`size_t size,`
`size_t n,`
`FILE* stream)`
	Schreiben von max. n Objekten der Größe size. Der Returnwert liefert die Anzahl der geschriebenen Objekte.

Postionierung in Datei

Funktion	Verwendung
`int fseek(FILE* stream, long offset, int origin);`	Setzt die Dateiposition auf den Wert Offset, relativ zu DateiAnfang (origin=SEEK_SET), Dateiende(origin=SEEK_END) oder zum aktuellen Stand(origin=SEEK_CUR)
`long ftell(FILE* stream);`	Liefert die aktuelle Lese-/Schreibposition
`void rewind(FILE* stream);`	Stellt die Lese-/Schreibposition auf den Dateianfang.

Beispielbetrachtungen

- Auf den nachfolgenden Seiten werden Beispiele betrachtet, die sich auf das Beispiel tStud beziehen.
- Es wird demonstriert, wie die Daten auf unterschiedliche Weise geschrieben und wieder gelesen werden werden können.
- Es werden Vor- und Nachteile der einzelnen Arten, Daten zu speichern, betrachtet.
- Typische Anwendungen bestehen aus dem Erfassen, Anzeigen, Auswerten und der Pflege von Daten. Dazu wird oft der gesamte Datenbestand aus einer Datei oder mehreren Dateien gelesen in einer programminternen Datenrepräsentation, oft sortiert, verwaltet (z.Bsp.: durch Listen) und am Ende wieder in einer externen Datenrepräsentation in Dateien gespeichert. Interne und externe Datenrepräsentation unterscheiden sich oft.
- Die nachfolgenden Beispiele konzentrieren sich auf die Dateiarbeit.

main1.c

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

#include "student1.h"

char buf[128];

int main(int argc, char** argv)
{
  FILE* pf=NULL;
  tStud stud;
  tStud* pstud;
  if (argc !=4)
  {
      printf("usage: %s <file> <name> <matrikelnr>\n",argv[0]);
      exit (-1);
  }
  pf=fopen(argv[1],"ab");
  if (pf==NULL)
  {
      printf("file: %s could not open/create\n",argv[1]);
      exit (-1);
  }
  strcpy(stud.name,argv[2]);
  stud.matrNr=atoi(argv[3]);
  stud.noteBel=0;
  stud.noteKl =0;
  writeStud(pf, &stud);
  fclose(pf);
  pf=fopen(argv[1],"rb");
  while( (pstud=readStud(pf))!=NULL )
  {
      displayStudent(pstud);
      free(pstud);
  }

  return 0;
}
```

Siehe s. 6 / 10

Das Beispiel öffnet in Zeile 19 eine Datei.
In Zeile 29 wird ein neuer Datensatz mit der Funktion writeStud an die Datei angehängt.
In Zeile 32 werden Daten durch die Funktion readStud gelesen und in zeile 34 angezeigt. Die Funktion readStud verwendet malloc, deshalb wird in Zeile 35 free aufgerufen.

Die aufgerufenen Funktionen sind in der Quelldatei student1.h deklariert und in student1.c definiert. Der Datentyp tStud ist ebenfalls in student1.h deklariert.

Headerfile student1.h

```
1
2   extern char buf[];
3
4   typedef struct tStudent
5   {
6     char  name[30+1];
7     int   matrNr;
8     float noteKl;
9     int   noteBel;
10  }tStud;
11
12  // displays the one Student
13  void   displayStudent(tStud *s);
14
15  // gets on Student vom stdin vial Textdialog
16  tStud* getStudent();
17
18  // writes one student pointed to by ps into ther open file pf
19  int    writeStud(FILE* pf, tStud* ps);
20
21  // reads one student out of the open File pf
22  // the returned buffer has to free by the caller
23  tStud* readStud(FILE* pf);
24
```

Implementationsfile student1.c

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "student1.h"
5
6
7  void displayStudent(tStud *s)
8  {
9      printf("%-30s, %6d, %3.1f, %d\n",
10             s->name,
11             s->matrNr,
12             s->noteKl,
13             s->noteBel);
14 }
15
16 tStud* getStudent()
17 {
18     tStud* ps=malloc(sizeof(tStud));
19     if (ps)
20     {
21         printf("%-10s: ","Name");
22         fgets(buf,32,stdin);
23         buf[strlen(buf)-1]=0;
24         strcpy(ps->name,buf);
25         printf("%-10s: ","MatrikelNr");
26         fgets(buf,32,stdin);
27         ps->matrNr=atoi(buf);
28         ps->noteKl=0;
29         ps->noteBel=0;
30     }
31     return ps;
32 }
```

```c
33
34 int writeStud(FILE* pf, tStud* ps)
35 {
36     return fwrite(ps, sizeof(*ps),1,pf);
37 }
38
39 tStud* readStud(FILE* pf)
40 {
41     tStud *tmp=malloc(sizeof(tStud));
42     if (tmp)
43     {
44         int ret=fread(tmp, sizeof(tStud), 1, pf);
45         if (ret!=1){free (tmp); tmp=NULL;}
46     }
47     return tmp;
48 }
```

Die Funktionen displayStudent und getStudent sollten bereits hinlänglich bekannt sein und werden hier nicht näher betrachtet.
Die Funktion writeStud speichert eine 1:1 Kopie der Struktur am Dateiende in die Datei.
Die Funktion readStud liest einen Studentdatensatz, wie er zuvor gespeichert worden ist. Beim nächsten Aufruf wird der nächste Datzensatz gelesen. Die Daten werden in einem Speicherbereich aus dem heap (malloc/free) zurückgegeben.

Anmerkungen

- Die Daten werden hier 1:1 in die Datei kopiert. Das geht nur, wenn die Daten keine Pointer enthalten.
- Die Daten werden hier plattformabhängig gespeichert, sie können auf einer anderen Plattform möglicherweise nicht verabeitet (Byteorder/Verarbeitungsbreite) werden.
- Die Datensätze sind alle gleich groß. Das bietet die Möglichkeit des wahlfreien Lesens/Schreibens von Daten. (fseek)
- Die Daten in der Datei können visuell nicht ohne Weiteres gelesen und in der Regel nicht modifiziert werden.

Arbeitsweise des Programms

- Nach mehrmaligem Aufruf sind mehrere Datensätze in der Datei gespeichert.
- Es ergibt sich folgende Ausgabe:

```
beck@silent110:~/serv/DOC/C/Praesentationen/V9_Dateien$ ./a.out fbin Otto 13213
Otto                          ,  13213, 0.0, 0
beck@silent110:~/serv/DOC/C/Praesentationen/V9_Dateien$ ./a.out fbin Lehmann 12123
Otto                          ,  13213, 0.0, 0
Lehmann                       ,  12123, 0.0, 0
beck@silent110:~/serv/DOC/C/Praesentationen/V9_Dateien$ ./a.out Huckebein 12111
usage: ./a.out <file> <name> <matrikelnr>
beck@silent110:~/serv/DOC/C/Praesentationen/V9_Dateien$ ./a.out fbin Huckebein 12111
Otto                          ,  13213, 0.0, 0
Lehmann                       ,  12123, 0.0, 0
Huckebein                     ,  12111, 0.0, 0
beck@silent110:~/serv/DOC/C/Praesentationen/V9_Dateien$
```

Bei folgendem Dateiinhalt:

```
beck@silent110:~/serv/DOC/C/Praesentationen/V9_Dateien$ hexdump -C fbin
00000000  4f 74 74 6f 00 00 00 00  7d fd 40 41 2f 56 00 00  |Otto....}.@A/V..|
00000010  60 4b 29 1a f7 7f 00 00  00 00 00 00 00 00 00 00  |`K).............|
00000020  9d 33 00 00 00 00 00 00  00 00 00 00 4c 65 68 6d  |.3..........Lehm|
00000030  61 6e 6e 00 7d 1d 80 e4  97 55 00 00 60 0b c0 65  |ann.}....U..`..e|
00000040  06 7f 00 00 00 00 00 00  00 00 00 00 5b 2f 00 00  |............[/..|
00000050  00 00 00 00 00 00 00 00  48 75 63 6b 65 62 65 69  |........Huckebei|
00000060  6e 00 5f 48 00 56 00 00  60 db de 0b 5a 7f 00 00  |n._H.V..`...Z...|
00000070  00 00 00 00 00 00 00 00  4f 2f 00 00 00 00 00 00  |........O/......|
```

Textdateien

- Daten werden heute häufig in Textdateien abgelegt.
- Sie bieten den Vorteil, dass sie visuell lesbar sind und mit einem gewöhnlichen Editor manipuliert werden können.
- Jede Information in den Daten muss dabei als Text ausgegeben werden.
- Die einzelnen Informationen müssen durch Trennzeichen voneinander abgesetzt werden.
- Dies kann zeilenweise oder durch andere Trennzeichen geschehen.

Einfache Textdateien

Zeilenweise Trennung der Informationen	Commaseparated Values (CSV)
Hans Huckebein 12121 2.3 2 Anna Haberland 12321 2.0 1	Hans Huckebein,12121,2.3,2 Anna Haberland,12321,2.0,1
Alle Informationen werden einfach zeilenweise in die Datei geschrieben. Beim Wiedereinlesen in der selben Reihenfolge, werden die Daten dann intern wieder den Strukturobjekten zugeordnet.	Die Daten eines Datensatzes (eines Strukturobejtes) werden auf einer Zeile der Textdatei gespeichert. Jeder Zeile repräsentiert einen Datensatz. Das Trennzeichen (hier Komma) darf nicht Bestandteil der Daten sein.

Zeilenweises Lesen v. Textdateien

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

char buf[128];

int main(int argc, char* argv[])
{
  FILE* f;
  if (argc!=2) {fprintf(stderr, "usage %s <file>\n",argv[0]); exit(-1);}
  f=fopen(argv[1],"rt");
  if (f==NULL) {fprintf(stderr, "could not open %s \n",argv[1]); exit(-1);}
  while(fgets(buf,128,f))
  {
      printf(buf);
  }
  return 0;
}
```

Beipiel zeilenweise Speicherung

```
34  int writeStud(FILE* pf, tStud* ps)
35  {
36      return fprintf(pf, "%s\n%d\n%f\n%d\n",
37                        ps->name,
38                        ps->matrNr,
39                        ps->noteKl,
40                        ps->noteBel);
41  }
42
43  tStud* readStud(FILE* pf)
44  {
45      int ret;
46      tStud *tmp=malloc(sizeof(tStud));
47      if (tmp)
48      {
49          ret=fscanf(pf, "%s\n%d\n%f\n%d\n",
50                       (tmp->name),
51                       &(tmp->matrNr),
52                       &(tmp->noteKl),
53                       &(tmp->noteBel));
54          if (ret<=0) // es wurde nichts gelesen
55              {free(tmp); tmp=NULL;}
56      }
57      return tmp;
58  }
```

Das Headerfile und main wird unverändert übernommen, es ändert sich lediglich Implementation von writeStud bzw. readStud. Unter **Microsoft** sollt in main das **Openflag b in t** geändert werden.

- Daten werden zeilenweise mit fprintf gespeichert.
- Das Einlesen erfolgt hier mit fscanf.
- Das zeilenweise Einlesen mit fgets und anschließender Konvertierung bei Zahlen ist ebenfalls möglich.

Beipiel CSV

```c
34    int writeStud(FILE* pf, tStud* ps)
35    {
36        return fprintf(pf, "%s,%d,%f,%d\n",
37                       ps->name,
38                       ps->matrNr,
39                       ps->noteKl,
40                       ps->noteBel);
41    }
42
43    tStud* readStud(FILE* pf)
44    {
45        int ret;
46        tStud *tmp=malloc(sizeof(tStud));
47        if (tmp)
48        {
49            ret=fscanf(pf, "%s,%d,%f,%d\n",
50                       (tmp->name),
51                       &(tmp->matrNr),
52                       &(tmp->noteKl),
53                       &(tmp->noteBel));
54            if (ret<=0) // es wurde nichts gelesen
55              {free(tmp); tmp=NULL;}
56        }
57        return tmp;
58    }
```

Das Headerfile und main wird unverändert übernommen, es ändert sich lediglich Implementation von writeStud bzw. readStud. Unter **Microsoft** sollt in main das **Openflag b in t** geändert werden.

- Daten eines Datensatzes werden durch Komma getrennt mit fprintf gespeichert.
- Jeder Datensatz bildet eine Zeile.
- Das Einlesen erfolgt hier mit fscanf.
- Das zeilenweise Einlesen mit fgets und anschließender Konvertierung bei Zahlen ist ebenfalls möglich.
- CSV-Dateien können auch mit Office-Systemen bearbeitet werden.

JSON

Textuelle Darstellung von Daten durch Schlüssel-/Wertpaare. Die Datenstruktur wird durch Klammerungen abgebildet. Gute Erklärungen gibt es bei Wikipedia und bei json.org.

```
{
 "studenten": [
  {
       "name":    "Hans Huckebein"
       "matrNr":   12121
       "noteKl":   2.3
       "noteBel":  2
  }
  {
       "name":    "Anna Haberland"
       "matrNr":   12321
       "noteKl":   2.0
       "noteBel":  1
  }
 ]
}
```

XML

Textuelle Darstellung von Daten und Struktur durch sogenannte tags, bekannt von html. Dabei wird jedes Datenelement von einem öffenenden und einem schließenden Tag eingeschlossen. Schließende Tags sind durch den / gekennzeichnet.

```
<?xml version="1.0" ?>
<students>
  <student>
    <name>Hans Huckebein</name>
    <matrNr>12121</matrNr>
    <noteKl>2.3</noteKl>
    <noteBel>2</noteBel>
  </student>
  <student>
    <name>Anna Haberland</name>
    <matrNr>12321</matrNr>
    <noteKl>2.0</noteKl>
    <noteBel>1</noteBel>
  </student>
</students>
```

Wahlfreies Lesen und Schreiben

- Wahlfreies Lesen und Schreiben bedeutet, Lesen und/oder Schreiben von Daten an gezielter Stelle in der Datei.
- Sinnvoll ist dies nur bei Datensätzen mit einheitlicher Länge, das sind in der Regel Binärdaten (Beispiel student1).
- Das wahlfreie Lesen/Schreiben vollzieht sich immer in zwei Schritten:
  - Positionierung mit fseek
  - Lesen/schreiben ab der voreingestellten Position

Wahlfreies Lesen und Schreiben

- Eine Anwendung kann gezielt einen Datensatz aus der Datei herauspicken, damit etwas machen und ihn ggf. wieder an seine Stelle zurückschreiben. Aber Achtung: Dabei muss die Leseschreibstelle neu positioniert werden.
- Durch viele Dateizugriffe werden Programme langsam.
- Der Speicherbedarf für eine interne Datenrepräsentation wird auf Speicherplatz für bis zu einen Datensatz reduziert, da die Daten auf dem Datenträger verbleiben.

Beispiel Wahlfreies Lesen

- Das nachfolgende Beispiel bezieht sich auf student1 (Binärdaten).
- Im 2. Teil wird die Größe der Datei bestimmt und daraus die Anzahl der enthaöltenen Datensätze berechnet (Zeilen 35-37).
- Danach wird hinten beginnend Satz für Satz in der Datei mit fseek adressiert, gelesen und ausgegeben (Zeilen 38-45).

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

#include "student1.h"

char buf[128];

int main(int argc, char** argv)
{
  FILE* pf=NULL;
  tStud stud;
  tStud* pstud;
  int i;
  if (argc !=2 && argc!=4)
  {
      printf("usage: %s <file> <name> <matrikelnr>\n",argv[0]);
      exit (-1);
  }
  pf=fopen(argv[1],"ab");
  if (pf==NULL)
  {
      printf("file: %s could not open/create\n",argv[1]);
      exit (-1);
  }
  if (argc==4) // Daten erfassen, falls angegeben
  {
    strcpy(stud.name,argv[2]);
    stud.matrNr=atoi(argv[3]);
    stud.noteBel=0;
    stud.noteKl =0;
    writeStud(pf, &stud);
    fclose(pf);
  }
```

```c
  // Anzahl der Datensaetze bestimmen
  pf=fopen(argv[1],"rb");
  fseek(pf,0,SEEK_END);
  i=ftell(pf)/sizeof(tStud) -1;
  // Daten rueckwärts ausgeben
  while(i>=0)
  {
      fseek(pf, i*sizeof(tStud), SEEK_SET);
      if ((pstud=readStud(pf))!=NULL)
        displayStudent(pstud);
      free(pstud);
      i--;
  }

  return 0;
}
```

Wahlfreies Lesen und Schreiben

Eine anspruchsvolle Übung könnte sein:

Nach Implementierung des Beispiels student1 mit Binärdatei, könnte man die Daten in der Datei sortieren und dabei immer nur zwei Datensätze und deren Position im Speicher halten, diese vergleichen und ggf. vertauscht wieder in die Datei schreiben. Muss nicht getauscht werden, muss auch nicht geschrieben werden.