ModProg 10-11, Vorl. 9

Richard Grzibovski

Dez. 16, 2015

Übersicht

Übersicht

1. Kommandozeilenparameter
2. Datenströme und Dateideskriptoren
3. Öffnen und Schließen von Dateien
4. Zeichenorientierte Ein- und Ausgabe
5. Binäre Ein- und Ausgabe

Kommandozeilenparameter

Bislang haben wir in unseren Quelltexten offen gelassen, ob das Hauptprogramm Argumente entgegen nimmt (`main()`), oder wir haben ausdrücklich angegeben, dass das Hauptprogramm keine Argumente erwartet (`main(void)`).
Tatsächlich kann man `main()` Argumente ubergeben, die man beim Start des ubersetzten Programms an der Kommandozeile dem Programmnamen folgen lässt. Diese Kommanzeilenparameter kann man jedoch nicht nach Belieben gestalten.

Parameter für `main`

Bei der Argumentübergabe an `main()` muss man zwar eine feste Konvention beachten, diese ist aber sehr flexibel gestaltet, wie man schon an der Deklaration erkennen kann:

```
int main(int argc, char **argv);
```

Die Parameterbezeichner `argc` und `argv` sind festgelegt. Sie haben die folgende Bedeutung:

- `argc` (argument count) speichert die Anzahl der beim Programmaufruf ubergebenen Argumente, einschließlich des Programmnamens,
- `argv` (argument vector zuweilen auch argument values) ist ein Feld von Strings, dessen Einträge die ubergebenen Argumente sind. Diese werden also in Form von Zeichenketten ubergeben (`char*` entspricht einem String, daher der Doppelzeiger). Die Nummerierung beginnt wie immer mit 0 und alle Zeichenketten im Feld argv sind jeweils durch \0 terminiert.
- Gemäßder Konvention ist `argv[0]` der Programmname selbst und `argv[argc]` besitzt den Wert `NULL`.

```c
#include<stdio.h>
int main(int argc, char **argv)
{
      int i;
      printf(" argc = % i \n - - -\n",argc);
      for( i =0; i < argc ; i ++)
            printf ("argv[%i]: %s\n",i,argv[i]);
      return 0;
}
```

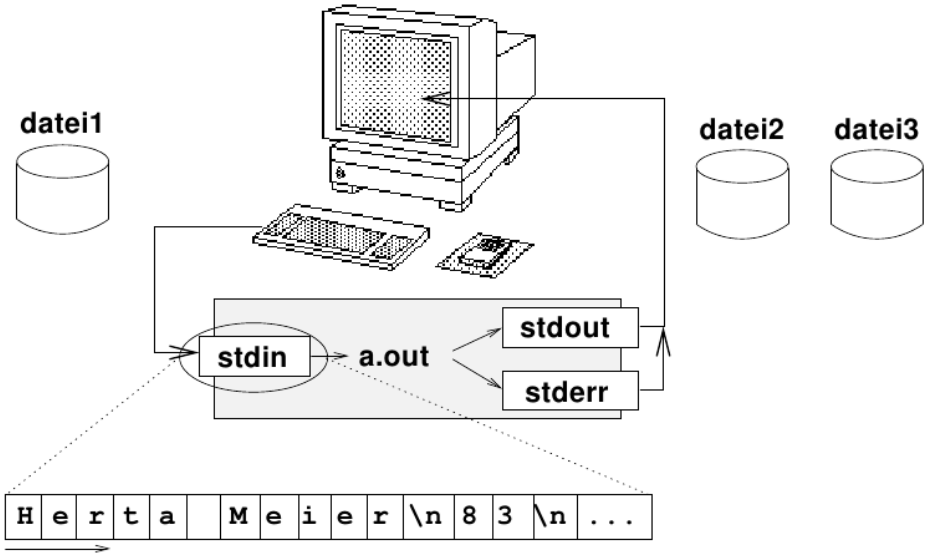
Datenströme

In C werden Ein- und Ausgabe über so genannte Datenströme (engl. (data) streams) realisiert.
In `<stdio.h>` sind die folgenden speziellen Datenströme deklariert:

- `stdout`: Standardausgabestrom (in der Regel der Bildschirm),
- `stdin` : Standardeingabestrom (in der Regel die Tastatur),
- `stderr`: Fehlerausgabestrom (in der Regel der Bildschirm).

Der Strom `stderr` unterscheidet sich von `stdout` dadurch, dass über ihn die Ausgaben ungepuffert erfolgen. Jedem Programm bzw. Datenstrom wird normalerweise ein Ein- und Ausgabepuffer zugewiesen, in dem die Daten zwischengespeichert werden. Die Ausgabe erfolgt meist erst dann, wenn dieser Puffer voll ist, das Programm beendet ist oder die Leerung des Puffers ausdrücklich veranlasst wird.

Vordefinierte Zeichenströme (streams)

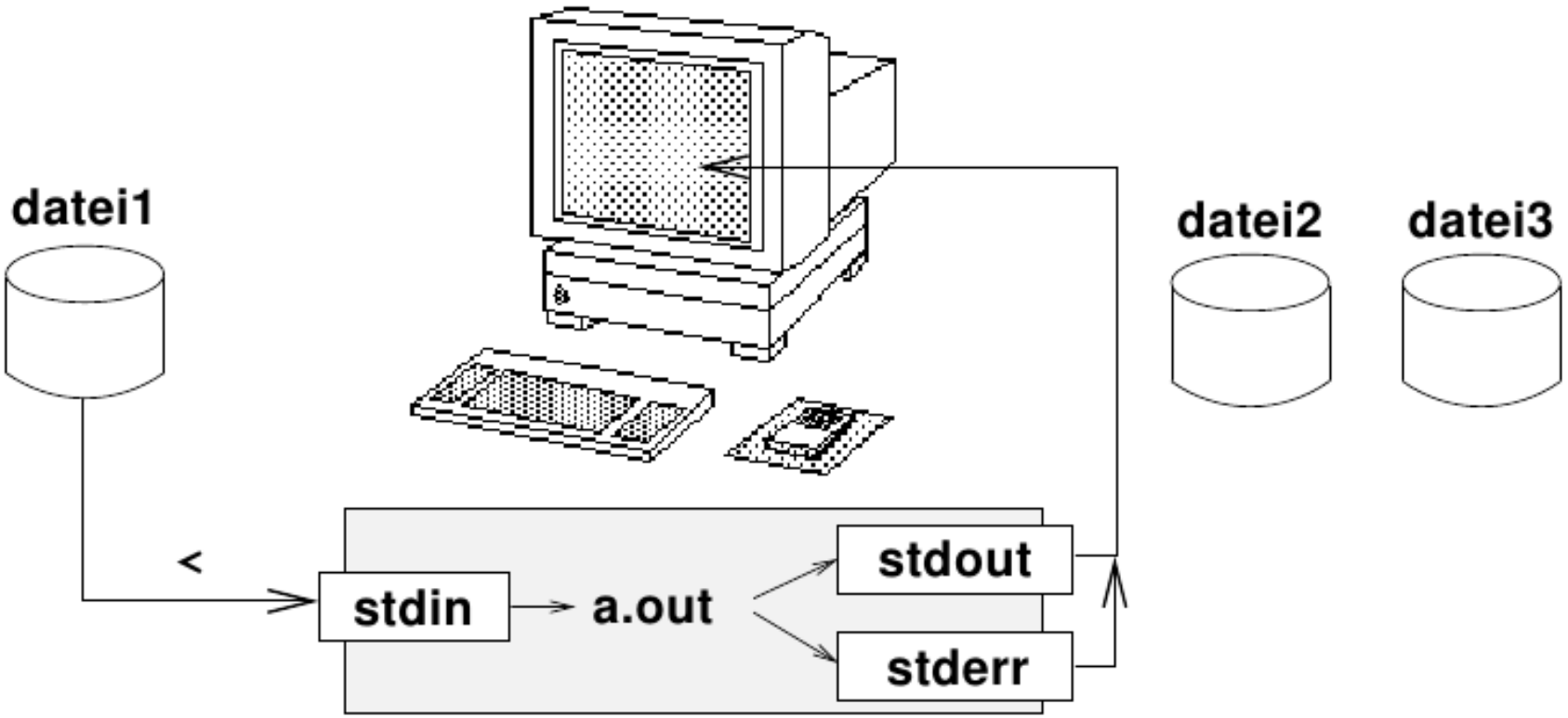


- "Normale" Zuordnung:
  - Standardeingabe `stdin` verbunden mit Tastatur,
  - Standardausgabe `stdout` und
  - Standardfehlerausgabe `stderr` verbunden mit Bildschirm
- Die bisher behandelten Ein-/Ausgabefunktionen
  - lesen von `stdin`: `scanf()`
  - schreiben auf `stdout`: `printf()`
- Warnungen, Fehlermeldungen usw. (z.B. von `assert()`) werden üblicherweise auf `stderr` ausgegeben

Umleitung der Ein-/Ausgabe (unter UNIX)

`a.out < datei1`

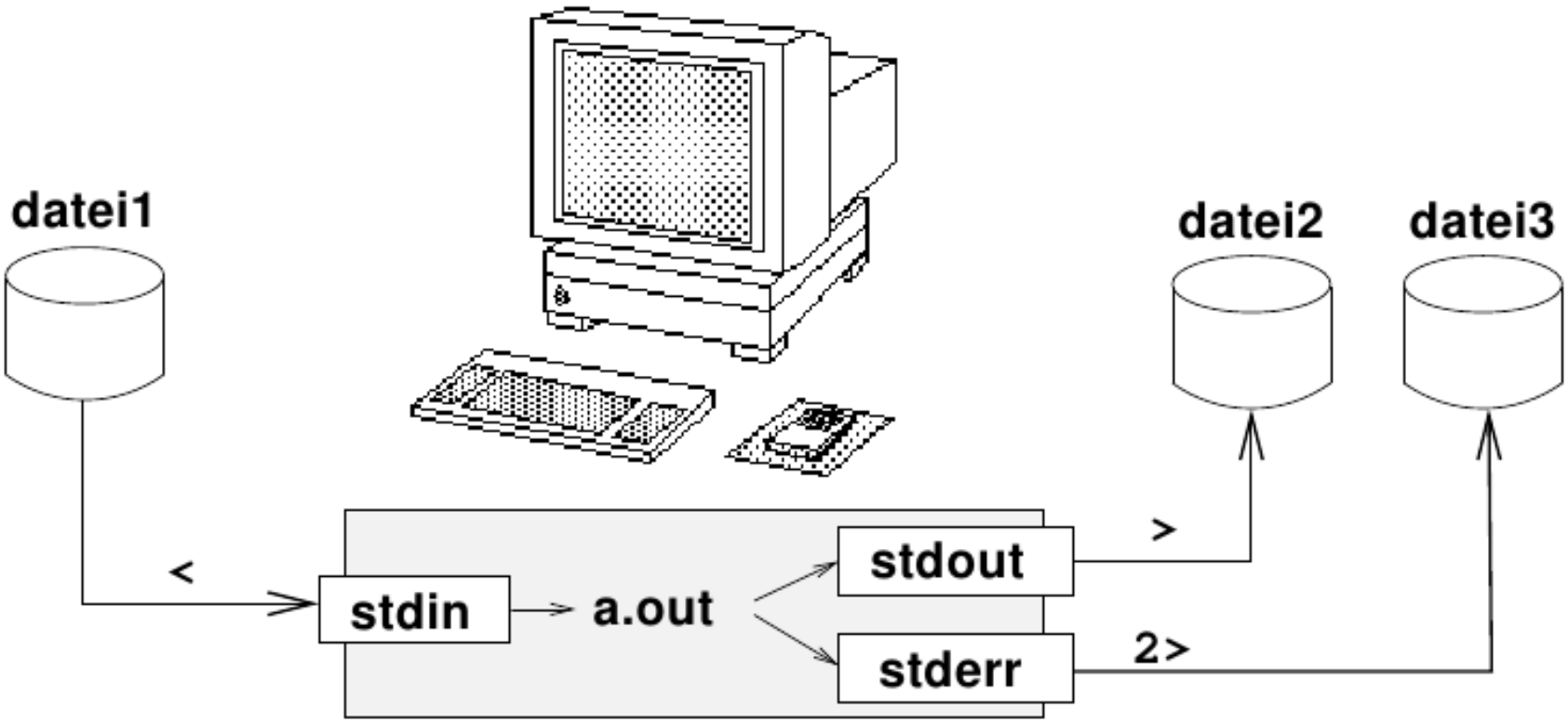
setzt `stdin` auf `datei1`, `stdout` und `stderr` bleiben erhalten



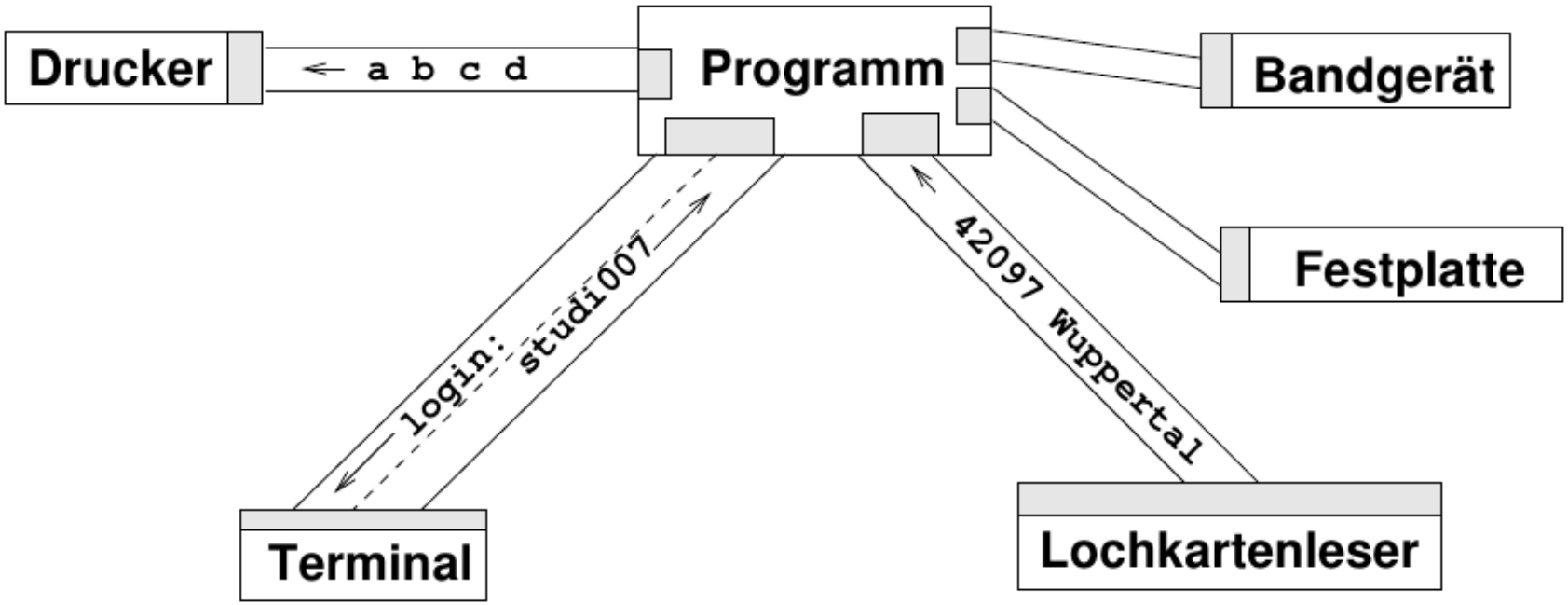
Umleitung der Ein-/Ausgabe (unter UNIX)

`a.out < datei1 > datei2 2> datei3`

setzt `stdin` auf `datei1` um, `stdout` auf `datei2` und `stderr` auf `datei3`



Datenströme (streams)



`stream`: Zeichen-/Byte- "Strom"

Der Begriff bietet

- Abstraktion von konkreten Ein-/Ausgabegeräten
- geräteunabhängige, einheitliche Behandlung

Datenströme (streams)

`stream`: Zeichen-/Byte- "Strom"

Der Begriff bietet

- Abstraktion von konkreten Ein-/Ausgabegeräten
- geräteunabhängige, einheitliche Behandlung

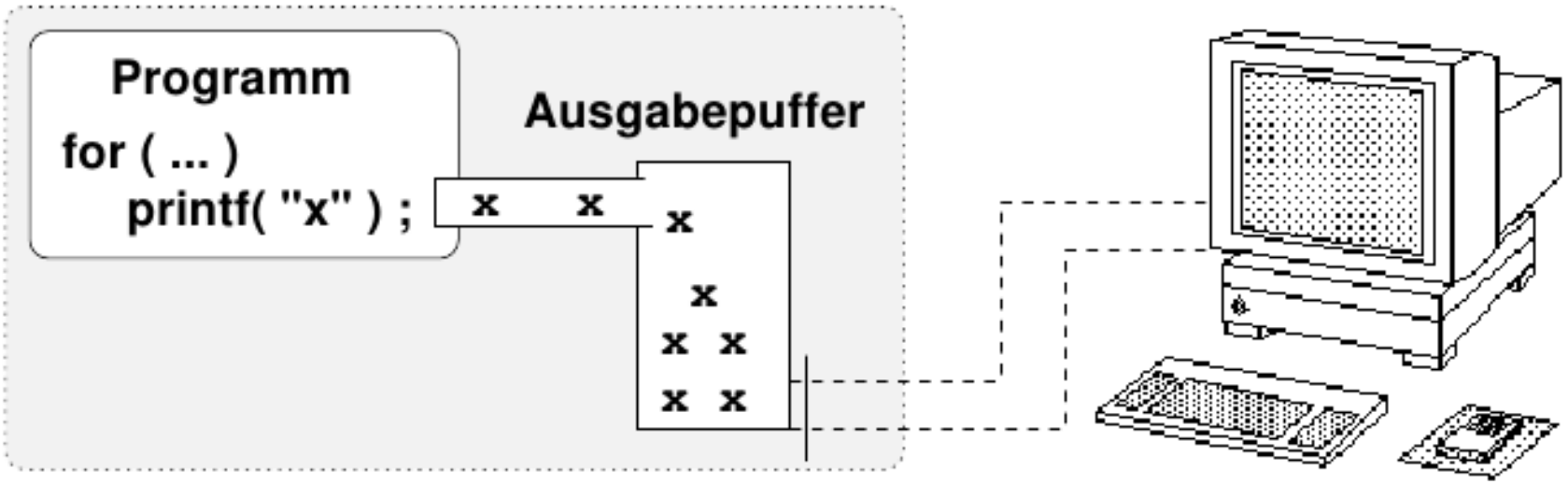
Text-Ströme vs Binär-Ströme:

- Text-Streams haben Zeilenstruktur:
  - ggf. erfolgt eine Betriebssystem-abhängige Umwandlung von Zeilenende-Zeichen und spezielle Behandlung von Sonderzeichen
  - Programm bleibt ohne Änderung übertragbar
- Bei Binär-Streams erfolgt keine Umsetzung

Pufferung

Gepufferte Ausgabe:

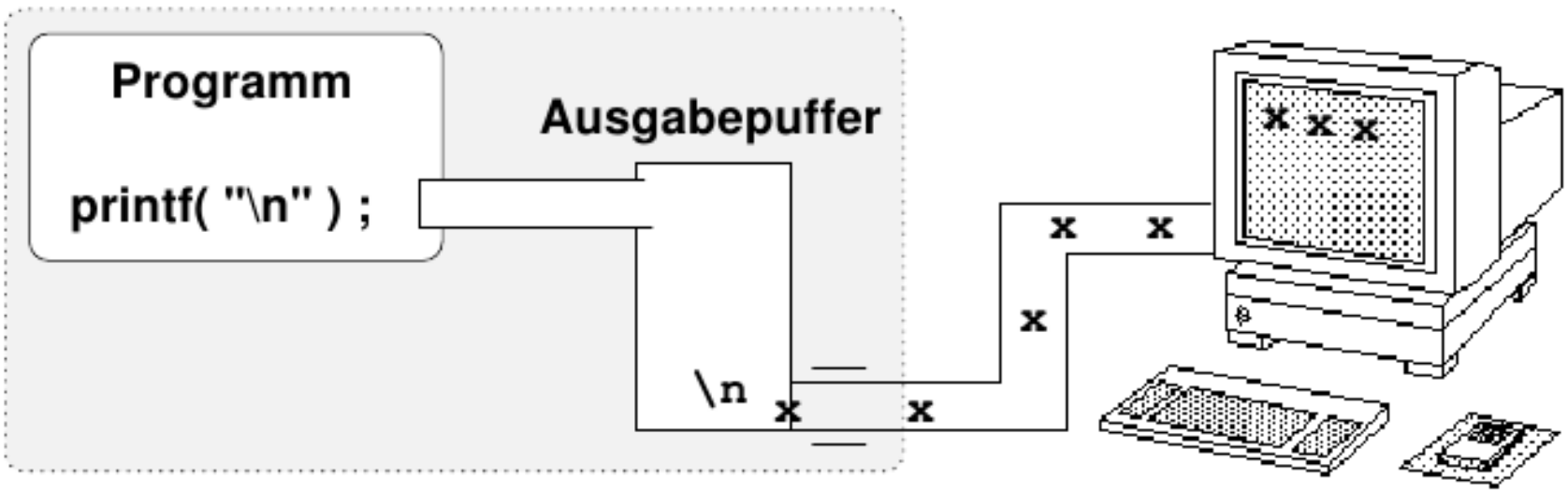
1. Zeichen werden im Ausgabepuffer gesammelt



Pufferung

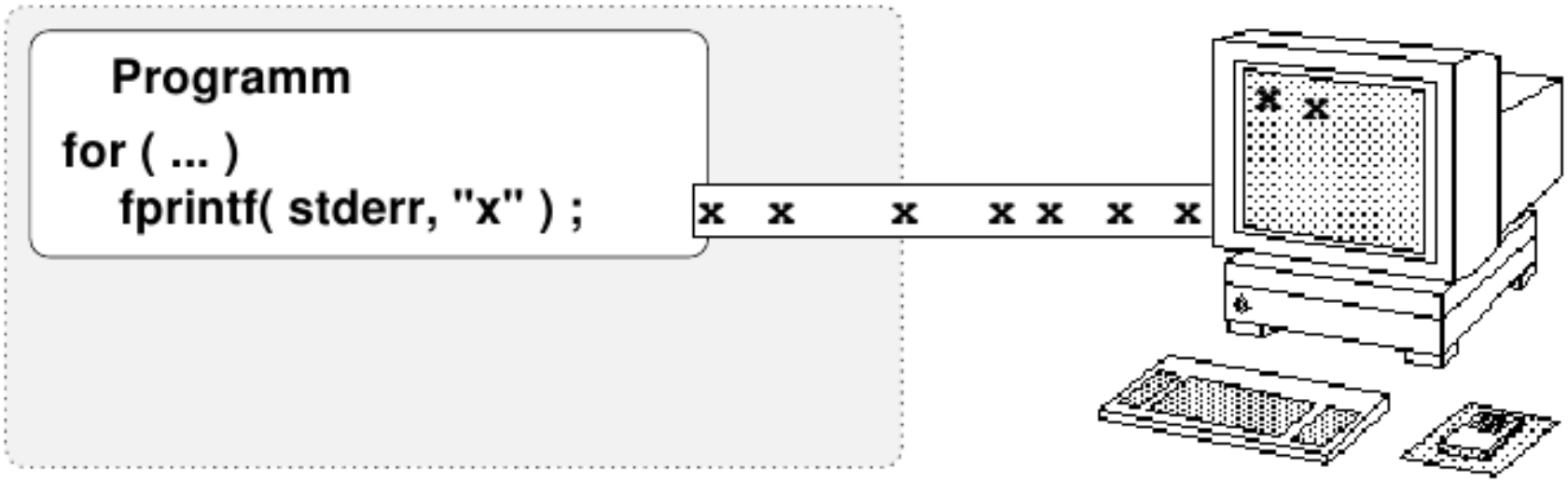
Gepufferte Ausgabe:

2. Ausgabepuffer wird geleert, z.B. bei Zeilenende oder `fflush(stdout)`



Pufferung

Ungepufferte Ausgabe (systemabhängig, z.B. `stderr`, ...):



Wozu Pufferung?

- Effzienter beim Lesen/Schreiben, z.B. auf Platten-Dateien
- Zeilenpufferung der Eingabe ermöglicht Korrekturen, bevor Zeile mit `<Return>` an das lesende Programm "abgeschickt" wird

Datenströme und Dateideskriptoren

Die Kommunikation mit einem Datenstrom erfolgt in C uber eine Zeigervariable auf den speziellen Datentypen `FILE`. Dieser ist in `<stdio.h>` deklariert. Diese Zeigervariable heißt auch Dateideskriptor.

Um je einen Datenstrom zu einer Ein- und einer Ausgabedatei bereitzuhalten, dient z.B. die folgende Deklaration:

```
FILE *eingdat, *ausgdat;
```

Operationen mit Dateien

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

- öffnet die Datei name
  Rückgabewert: Zeiger auf Datei bei Erfolg, `NULL` bei Fehler
- Mögliche Operationen auf dem stream hängen vom `modus` ab:

`r` (read)	Textdatei lesen.
`w` (write)	Textdatei schreiben. Evtl. vorhandene Datei wird gelöscht
`a` (append)	an Textdatei anhängen. Falls Datei name schon vorhanden, bleibt ihr Inhalt erhalten, und es wird am Dateiende geschrieben. Ansonsten wird die Datei name neu angelegt

  Andern (Lesen und Schreiben) bei Textdateien:

`r+`	Textdatei lesen/schreiben.
`w+`	Textdatei lesen/schreiben. Evtl. vorhandene Datei wird gelöscht
`a+`	Textdatei lesen/anhängen. Geschrieben wird immer am gerade aktuellen Dateiende

- Binärdateien: "b" (binary) dazu, also `rb`, `wb`, `ab`, `rb+`, `wb+`, `ab+` als modus

Operationen mit Dateien

```
int fclose( FILE *stream )
```

- schließt den angegebenen stream
  - Zuvor werden eventuell noch gepufferte Daten herausgeschrieben
  - Liefert Wert $\neq 0$ bei Fehler

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

- wie `printf()` bzw. `scanf()`, jedoch Ausgabe auf bzw. Eingabe von stream

```
int fputc( int c, FILE *stream )
```

- gibt Zeichen c auf stream aus
- Liefert `EOF` bei Fehler

```
int fgetc( FILE *stream )
```

- holt nächstes Zeichen von stream
- Liefert `EOF` bei Fehler

Operationen mit Dateien

`int fputs( const char *s, FILE *stream )`

- gibt String `s` auf stream aus

`char *fgets( char *s, int n, FILE *stream )`

- liest von stream, bis Zeilenende erreicht ist – jedoch höchstens $n - 1$ Zeichen
- Zeilenende-Zeichen wird mit in `s` abgelegt
- Null-Abschluss von `s` garantiert
- Ergebnis: `s` bei Erfolg, sonst `NULL`

`int feof( FILE *stream )`

- liefert einen Wert $\neq 0$ bei Dateiende von stream

`int fflush( FILE *stream )`

- erzwingt ein Leeren des Puffers von stream
  - Datei bleibt geöffnet
  - Liefert 0 bei Erfolg, `EOF` sonst

```c
#include<stdio.h>
int main() {
  float a, b ;
  FILE *eingdat , *ausgdat;
  eingdat = fopen( "input.dat","r");
  assert(eingdat!=NULL);
  ausgdat = fopen("output.dat","w");
  assert(ausgdat!=NULL);
  fscanf(eingdat, "a = %f\nb = %f" ,&a ,&b);
  fprintf(ausgdat, "Summe aus %f und %f ist %f\n",
                    a, b, a+b);
  fprintf( stdout, "Summe aus %f und %f ist %f\n",
                    a, b, a+b);
  fclose(eingdat);
  fclose(ausgdat);
  return 0;
}
```

Der Inhalt von `input.dat` **muss** die folgende Struktur haben:

```
a = 1.0
b = 2.0
```

```c
#include <stdio.h>
#include <string.h> // <-- strlen
int main() {
   FILE *stream ;  /* Zeiger auf Eingabestream */
   char zeile[1000];
   int zeilen = 0, zeichen = 0 ;
   stream = fopen("data.txt","r");
   /* Eingabedatei zeilenweise lesen */
   while( fgets(zeile, 1000, stream) != NULL ) {
       zeilen++;
       zeichen += strlen(zeile) ;
   }
   fclose(stream) ; /* Eingabedatei schliessen */

   /* Zaehlung ausgeben */
   printf( "%d Zeilen mit insgesamt %d Zeichen\n",
           zeilen, zeichen ) ;
   return 0;
}
```

Binäre Ein- und Ausgabe

Hier werden die Daten direkt als Bytefolgen in die Dateien geschrieben und aus ihnen gelesen. Die Größe hängt nur vom Datentyp ab (und nicht etwa von der Anzahl der ausgegebenen Stellen).

```
size_t fwrite(const void *zgr, size_t size, size_t anzahl,
                                                FILE *Dskr);
```

- Die Funktion schreibt von der Speicherposition, auf die `zgr` zeigt, anzahl Datenobjekte der Größe `size` in den Datenstrom `Dskr`.
- Der Rückgabewert ist die Anzahl der erfolgreich geschriebenen Datenobjekte und kann zum Test auf Ausgabefehler verwendet werden.
- Hier bedeutet `const void *`, dass nicht festgelegt ist, auf welchen Datentyp `zgr` zeigt und dass das referenzierte Datenobjekt geschützt ist.

Binäre Ein- und Ausgabe

```
size_t fread(void *zgr, size_t size, size_t anzahl, FILE *Dskr);
```

- Die Funktion liest `anzahl` Datenobjekte der Größe `size` aus dem Datenstrom `Dskr` und speichert sie (sequenziell) ab der Position, auf die `zgr` zeigt.
- Der Rückgabewert ist die Anzahl der erfolgreich gelesenen Datenobjekte.

```c
#include<stdio.h>
#include<assert.h>
void vector_speichern(char* fname, double *vec, int N) {
  FILE *out;
  int m;
  out=fopen(fname,"wb");
  m=fwrite(&N,sizeof(int),1,out);
  assert(m==1);
  m=fwrite(vec,sizeof(double),N,out);
  assert(m==N);
  fclose(out);
}
```

```c
#include<stdio.h>
#include<assert.h>
int vector_einlesen(char* fname, double **vec) {
  FILE *inp;
  int m,N;
  inp=fopen(fname,"rb");
  m=fread(&N,sizeof(int),1,inp);
  assert(m==1);
  assert(N>0);
  (*vec)=malloc(N*sizeof(double));
  m=fread(*vec,sizeof(double),N,inp);
  assert(m==N);
  fclose(inp);
  return N;
}
```