Teil 10: Präprozessor und Ein-/Ausgabe

Gliederung

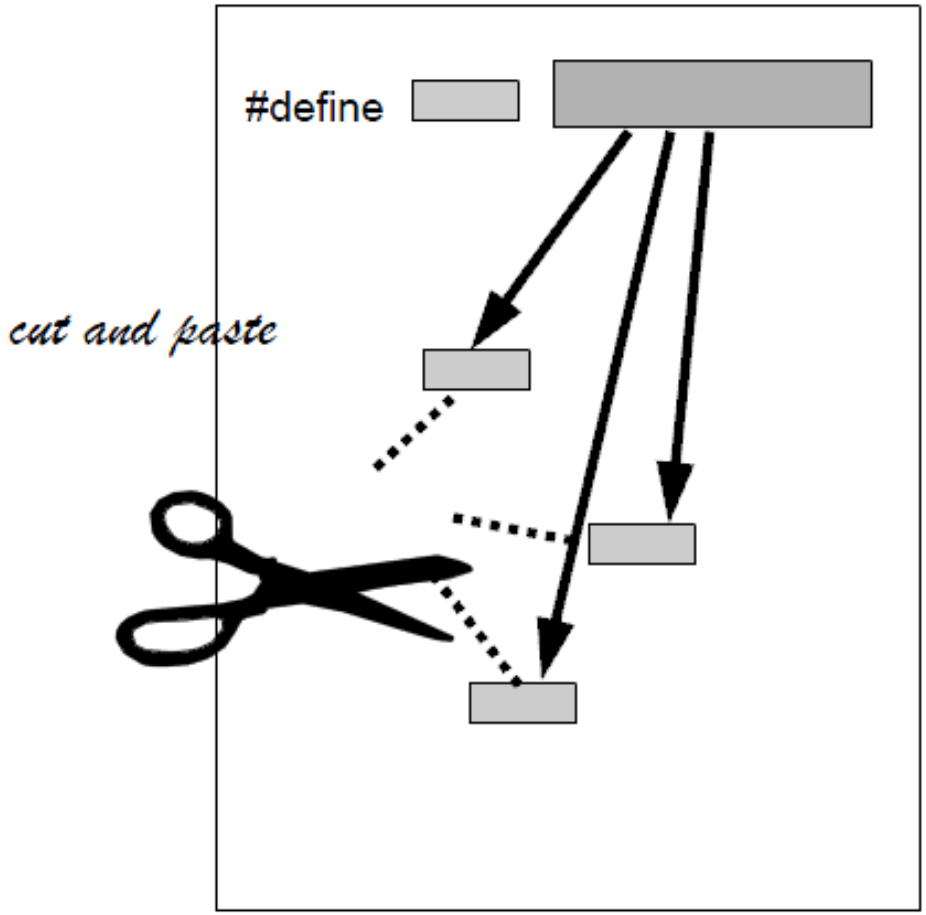
Präprozessor

Streams

Dateioperationen

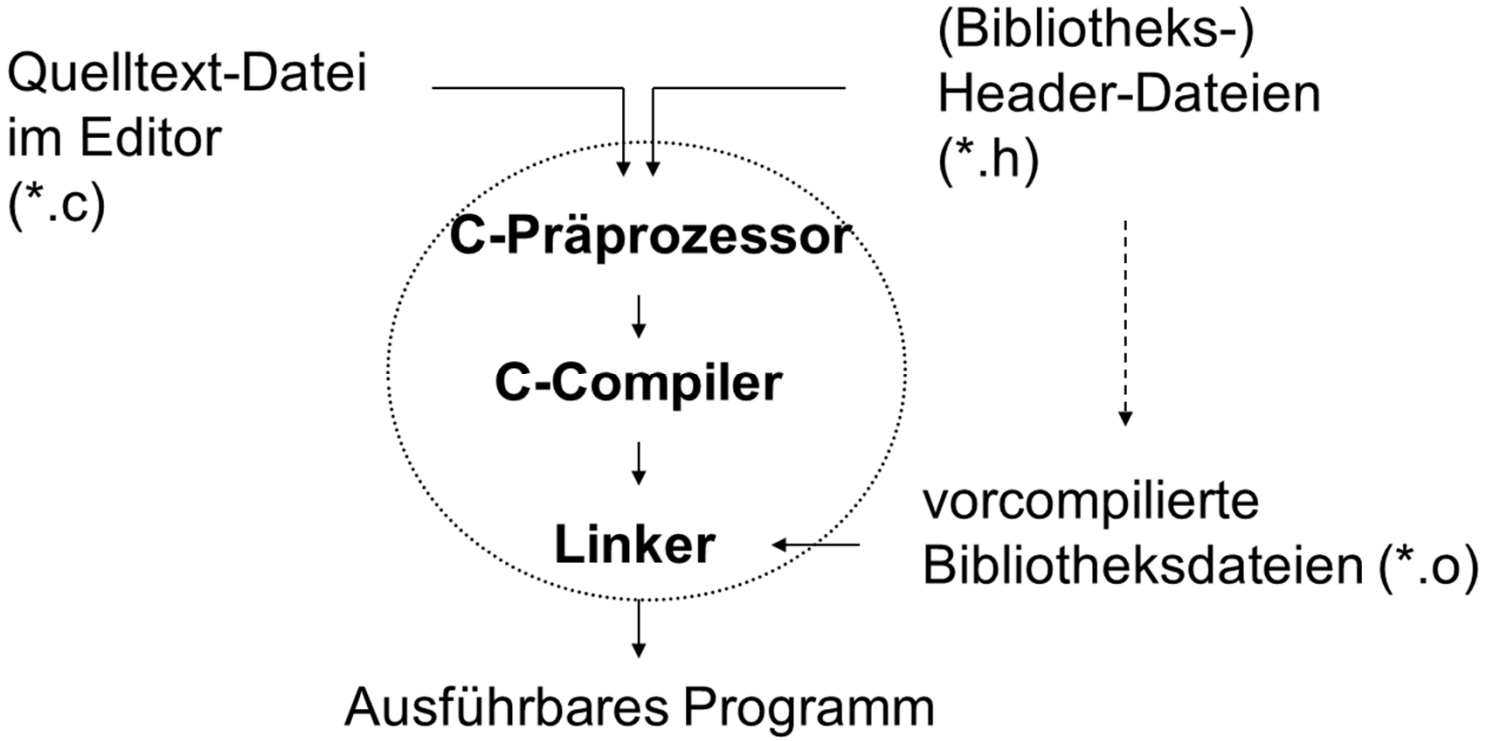
Kommandozeilenparameter

Präprozessor



Aufgaben des Präprozessors

- Einfügen von Dateien
- Ersetzen von Text
- bedingte Kompilierung



Präprozessor (Preprozessor)

- Präprozessoranweisungen = **Direktiven**
- Beginn mit Raute-Zeichen (**#**)
- zeilenorientiert, **kein** Semikolon zur Terminierung

#include Direktive

- Einbinden von ANSI-C Standard-Bibliotheken:

```
#include <dateiname>
```

- Eigene Headerdatei:

```
#include "dateiname"
```

- Absolute Pfadangaben

```
#include "c:/myprog/header.h"
```

#define Direktive

- Symbolische Konstanten und Makros: "NAME" → "Ersatztext"

```
#define NAME Ersatztext
```

- Beispiel:

```
#define PI 3.14159

main()
{
   printf("Pi = %f", PI);
}
```

- Verschachtelung

```
#define PI_MAL_2 (PI + PI)
```

Vorteil: Berechnung der Ausdrücke schon bei der Kompilierung

Makros

- Name enthält Parameterliste, Argumente -> Ersetzung im Ersatztext

```
#define Makroname(Parameterliste) Ersatztext
```

- Beispiele:

```
#define SUM(n1, n2) (n1 + n2)
```

kann wie folgt benutzt werden:

```
n = SUM(17, 4);        // liefert dem Compiler n = 21;
```

häufig verwendet:

```
#define MAX(x,y) ((x<y) ? y : x)

#define TAUSCHE_INT(x,y)  {                    \
                            int j;             \
                            j=x; x=y; y=j;     \
                          }
```

Makros: Fehlerquellen

```
#include <stdio.h>
#define PRODUKT(a, b) a * b

main()
{
  int x = 2, y = 3, z1, z2;

  z1 = PRODUKT(x, y);
  z2 = PRODUKT(x + 1, y - 1);

  printf("%d %d", z1, z2);
}
```

Makros: Fehlerquellen

```
#include <stdio.h>
#define PRODUKT(a, b) a * b

main()
{
  int x = 2, y = 3, z1, z2;

  z1 = PRODUKT(x, y);
  z2 = PRODUKT(x + 1, y - 1);

  printf("%d %d", z1, z2);
}
```

Makroauflösung liefert

```
z1 =  x * y ;
z2 =  x + 1 * y - 1 ;          // korrekt: (x + 1) * (y - 1)

-> Ausgabe: 6 4                // korrekt: 6 6
```

Makros: Fehlerquellen

Abschluss des Makros mit einem Semikolon:

```
#define SUM(n1, n2) ((n1) + (n2));
```

Leerzeichen zwischen Makronamen und Parameterliste:

```
#define SUM (n1, n2) ((n1) + (n2))
            ^
```

Hier darf **kein** Leerzeichen stehen

Fazit: Makros Aufgrund von Seiteneffekten Makros mit Vorsicht einsetzen!

Vordefinierte Makros nach ANSI-C

Makro	Bedeutung
`__LINE__`	Zeilennummer innerhalb der aktuellen Quellcodedatei
`__FILE__`	Name der aktuellen Quellcodedatei
`__DATE__`	Datum, wann das Programm compiliert wurde (als Zeichenkette)
`__TIME__`	Uhrzeit, wann das Programm compiliert wurde (als Zeichenkette)
`__STDC__`	Liefert eine 1, wenn sich der Compiler nach dem Standard-C richtet.
`__STDC_VERSION__`	Liefert die Zahl 199409L, wenn sich der Compiler nach dem C95-Standard richtet; die Zahl 199901L, wenn sich der Compiler nach dem C99-Standard richtet. Ansonsten ist dieses Makro nicht definiert (z.B. für C89-Standard)

Beispiel: Makros

```
#include <stdio.h>

main()
{
   printf("Programm wurde compiliert am ");
   printf("%s um %s.\n", __DATE__, __TIME__);

   printf("Diese Programmzeile steht in Zeile ");
   printf("%d in der Datei %s.\n", __LINE__, __FILE__);

#ifdef __STDC__
   printf("Standard-C-Compiler!\n");
#else
   printf("Kein Standard-C-Compiler!\n");
#endif
}
```

Bedingte Compilierung

- Kompilierung nur bestimmter Teile des Quelltextes
- Auswertungen wieder **vor** der Kompilierung, nicht zur Laufzeit!
  - Nicht verwechseln mit C-Kontrollstrukturen!
- Anwendung:
  - Pflege mehrerer Programmversionen
  - hardware- oder betriebssystemspezifische Unterschiede
  - Fehlersuche
- verwendbare Direktiven: **#define**
  #undef
  #if
  #elif
  #ifdef
  #if defined
  #ifndef
  #else
  #endif

Bedingte Compilierung

#define definiert symbolische Konstante
#undef hebt Definitionen auf

```
#define NAME
                // NAME ist nun ein gueltiges Praeprozessor-Symbol

#undef NAME
               // NAME ist nun kein gueltiges Praeprozessor-Symbol mehr
```

Beispiel 1	Beispiel 2	Beispiel 3

`#ifdef NAME`
`  Programmteil 1`
`#endif`
 	
`...`
 	
`#ifndef NAME`
`  Programmteil 2`
`#endif`
	`#ifdef NAME`
	`  Programmteil 1`
	
	`#else`
	`  Programmteil 2`
	
	`#endif`
		`#if ConstAusdruck1`
		`  Programmteil 1`
		
		`#elif ConstAusdruck2`
		`  Programmteil 2`
		
		`#else`
		`  Programmteil 3`
		
		`#endif`

Beispiel Fehlersuche

```
#include <stdio.h>

int main()
{
   int a, b, c;

   scanf("%d %d", &a, &b);
   c = a * b;

   printf("Variableninhalte:");
   printf("a = %d b = %d c = %d\n", a, b, c);

   printf(“Ergebnis von %d mal %d ist %d\n", a, b, c);
   return 0;
}
```

Beispiel Fehlersuche

```
#include <stdio.h>
#define TEST

int main()
{
   int a, b, c;

   scanf("%d %d", &a, &b);
   c = a * b;

#ifdef TEST
   printf(“Variableninhalte:");
   printf(“a = %d b = %d c = %d\n", a, b, c);
#endif

   printf(“Ergebnis von %d mal %d ist %d\n", a, b, c);
   return 0;
}
```

Beispiel Hardware/Betriebssystem

```
#include <stdio.h>

int main()
{
#ifdef WINDOWS
   printf("Programmteil für Windows OS\n");
   printf("...\n");
#else
   printf("Programmteil fuer andere OS\n");
   printf("...\n");
#endif

#if CPU == AMD
   printf("Optimierter Programmteil für AMD Prozessoren\n");
   printf("...\n");
#elif CPU == INTEL
   printf("Optimierter Programmteil für Intel Prozessoren\n");
   printf("...\n");
#else
   printf("Programmteil für alle anderen Prozessoren\n");
   printf("...\n");
#endif
   return 0;
}
```

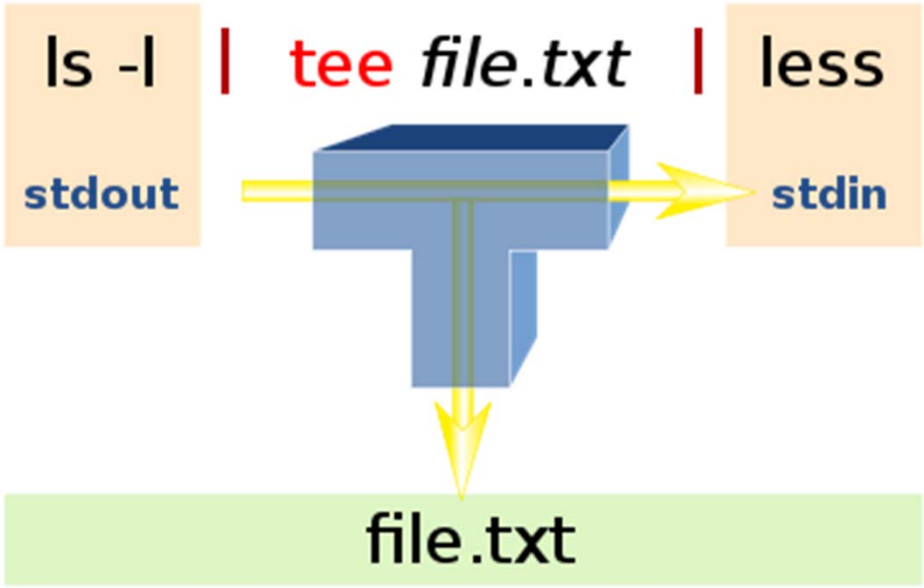
```
#include <stdio.h>
#define WINDOWS
#define CPU INTEL

int main()
{
#ifdef WINDOWS
   printf("Programmteil für Windows OS\n");
   printf("...\n");
#else
   printf("Programmteil fuer andere OS\n");
   printf("...\n");
#endif

#if CPU == AMD
   printf("Optimierter Programmteil für AMD Prozessoren\n");
   printf("...\n");
#elif CPU == INTEL
   printf("Optimierter Programmteil für Intel Prozessoren\n");
   printf("...\n");
#else
   printf("Programmteil für alle anderen Prozessoren\n");
   printf("...\n");
#endif
   return 0;
}
```

```
#include <stdio.h>
#define WINDOWS
#define CPU INTEL

int main()
{
#ifdef WINDOWS
   printf("Programmteil für Windows OS\n");
   printf("...\n");
#else
   printf("Programmteil fuer andere OS\n");
   printf("...\n");
#endif

#if CPU == AMD
   printf("Optimierter Programmteil für AMD Prozessoren\n");
   printf("...\n");
#elif CPU == INTEL
   printf("Optimierter Programmteil für Intel Prozessoren\n");
   printf("...\n");
#else
   printf("Programmteil für alle anderen Prozessoren\n");
   printf("...\n");
#endif
   return 0;
}
```

- Resultat für den nächsten Schritt der Kompilierung:

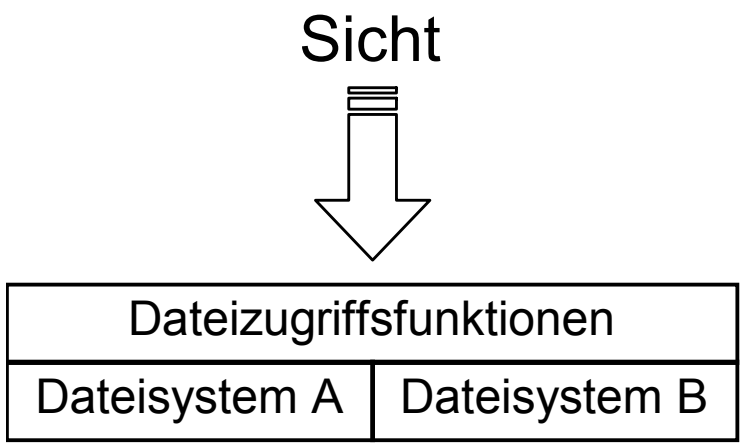
```
#include <stdio.h>

int main()
{
   printf("Programmteil für Windows OS\n");
   printf("...\n");

   printf(" Optimierter Programmteil für Intel Prozessoren\n ");
   printf("...\n");
   return 0;
}
```

Streams



Dateien und Dateisysteme

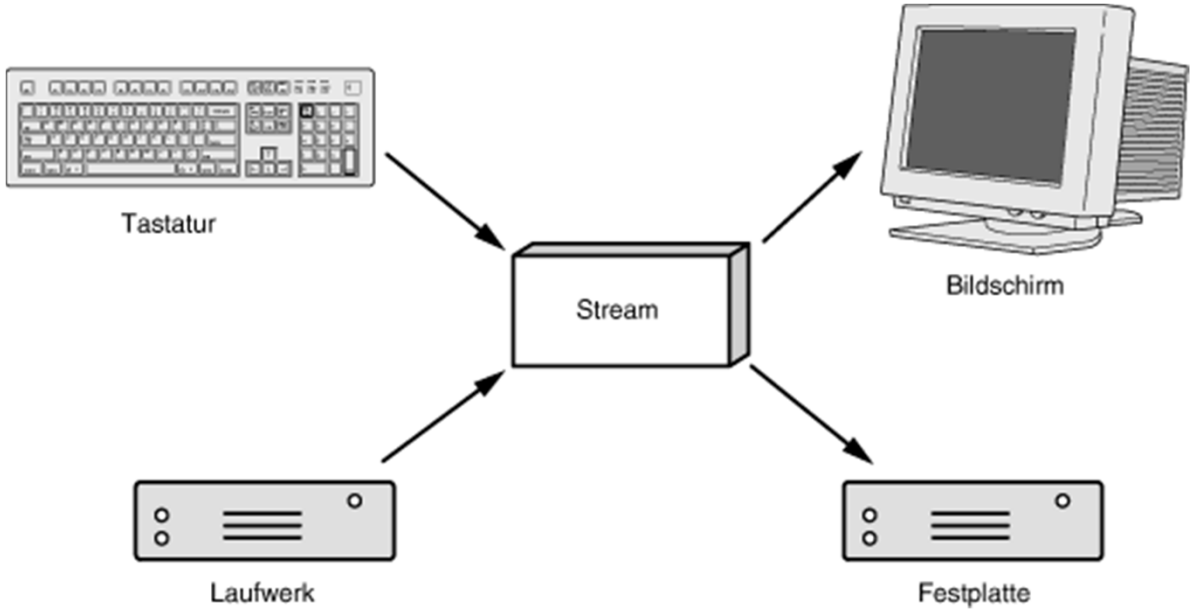
- Definition **Datei:** mit Namen versehener Datensatz beliebiger Länge
- Programme, die über Dateien kommunizieren, müssen sich über das Format verständigen
- Dateisystem als Struktureinheit kennt nur Bytes
- Zugriffs durch entsprechende (Bibliotheks-) Funktionen



Stream-Konzept

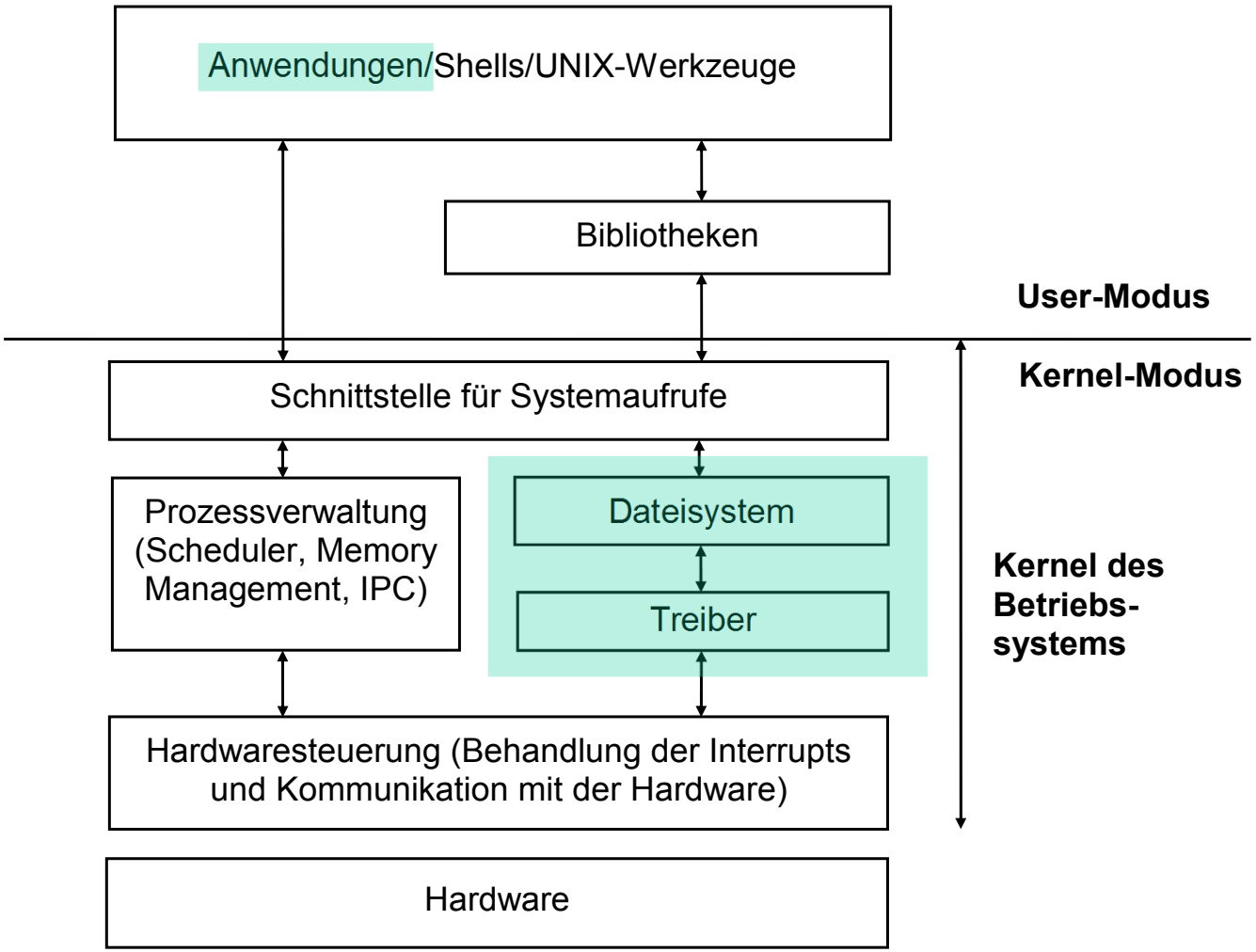
- "Eingabe/Verarbeitung/Ausgabe" (EVA-Prinzip)
- Konzept von Datenströmen: **streams**
  - abstraktes Modell: geordnete Folge von Bytes

    Datenquelle → Datenstrom → Datenziel
  - Quelle/Ziel: Zuordnung zu Datei oder Gerät



Schichtenmodell (Unix)

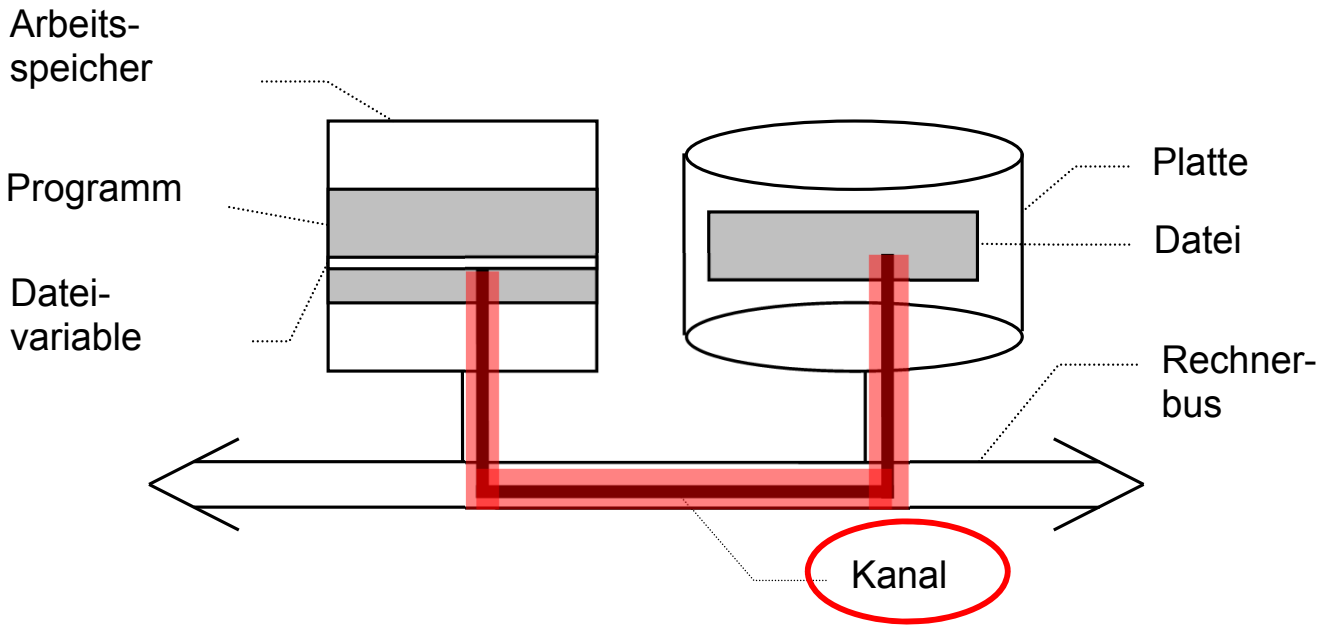
- Geräteabhängigkeit ist durch das Dateisystem verborgen



Ein-Ausgabe in C

- Bibliothek <stdio.h>
- Zugriff auf Dateien über Dateivariable (**File-Pointer**)
- Verknüpfung von Dateivariable und physikalischer Datei zur Laufzeit

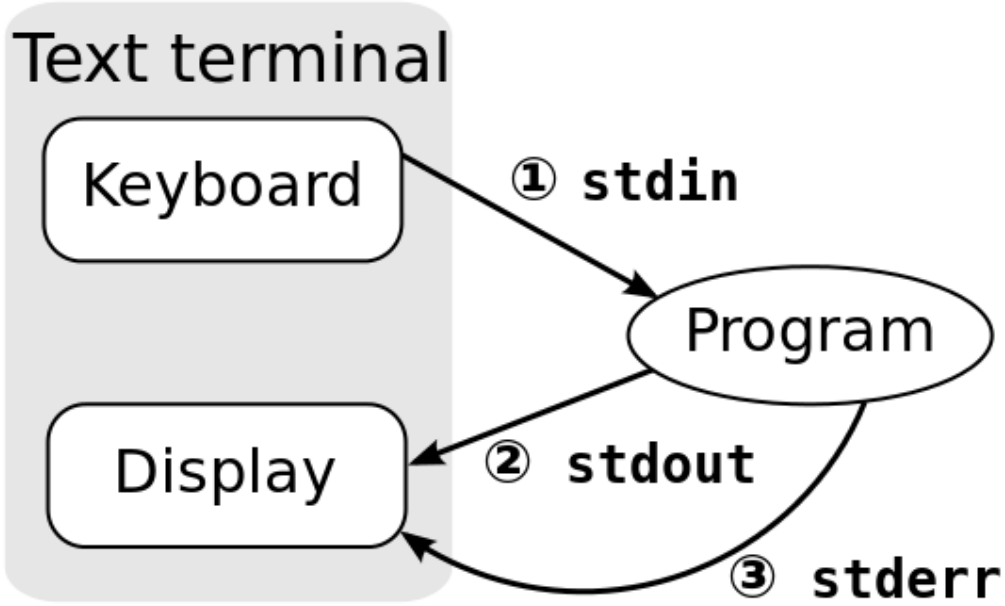
```
fp = fopen("TEST.DAT", "w");
```



Standardkanäle

- in jedem Programm vorhandene **Standardkanäle**

  (1) stdin (Standardeingabe, Voreinstellung Tastatur)
  (2) stdout (Standardausgabe, Voreinstellung Konsole)
  (3) stderr (Standardfehlerausgabe, Voreinstellung Konsole)



Umleitung durch das Betriebssystem

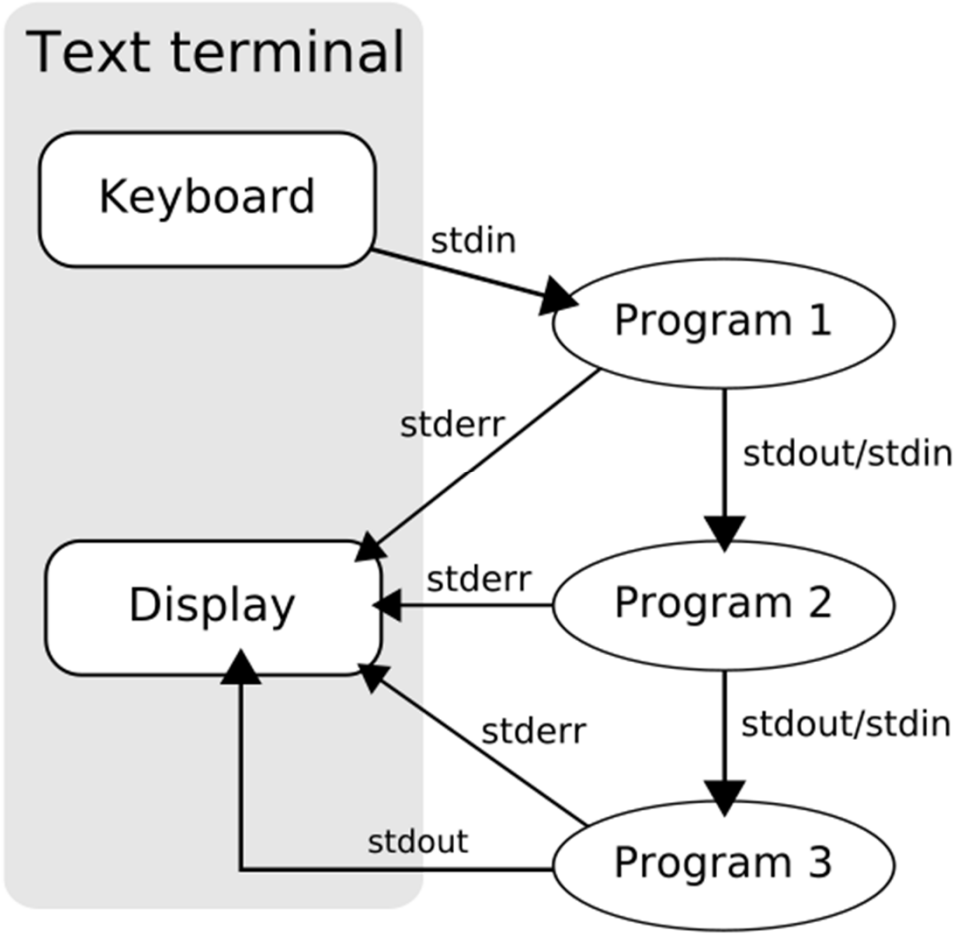
- **Umlenkung** (Umleitung) der Ein- und Ausgabe

```
myprog > test.out
```

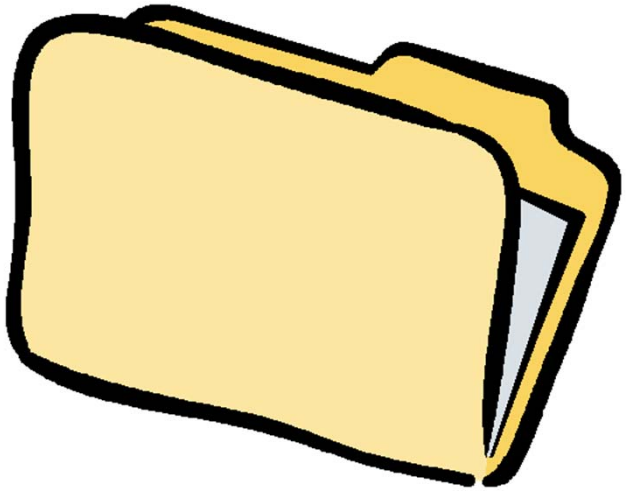
- **Pipelining**

```
prog1 | prog2 | prog3
```

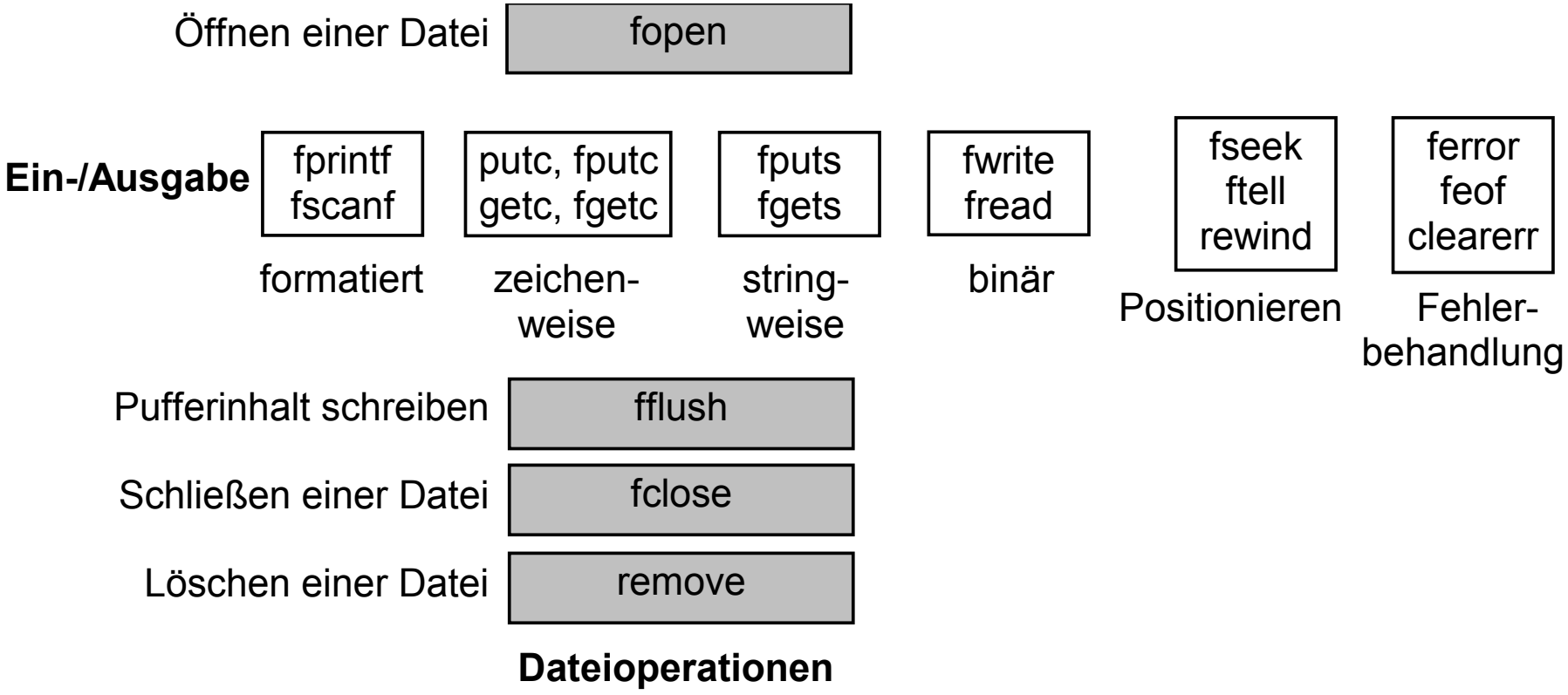
- eine Umlenkung bzw. Pipelining ist für das Programm transparent



Dateioperationen



Dateizugriffsfunktionen in C



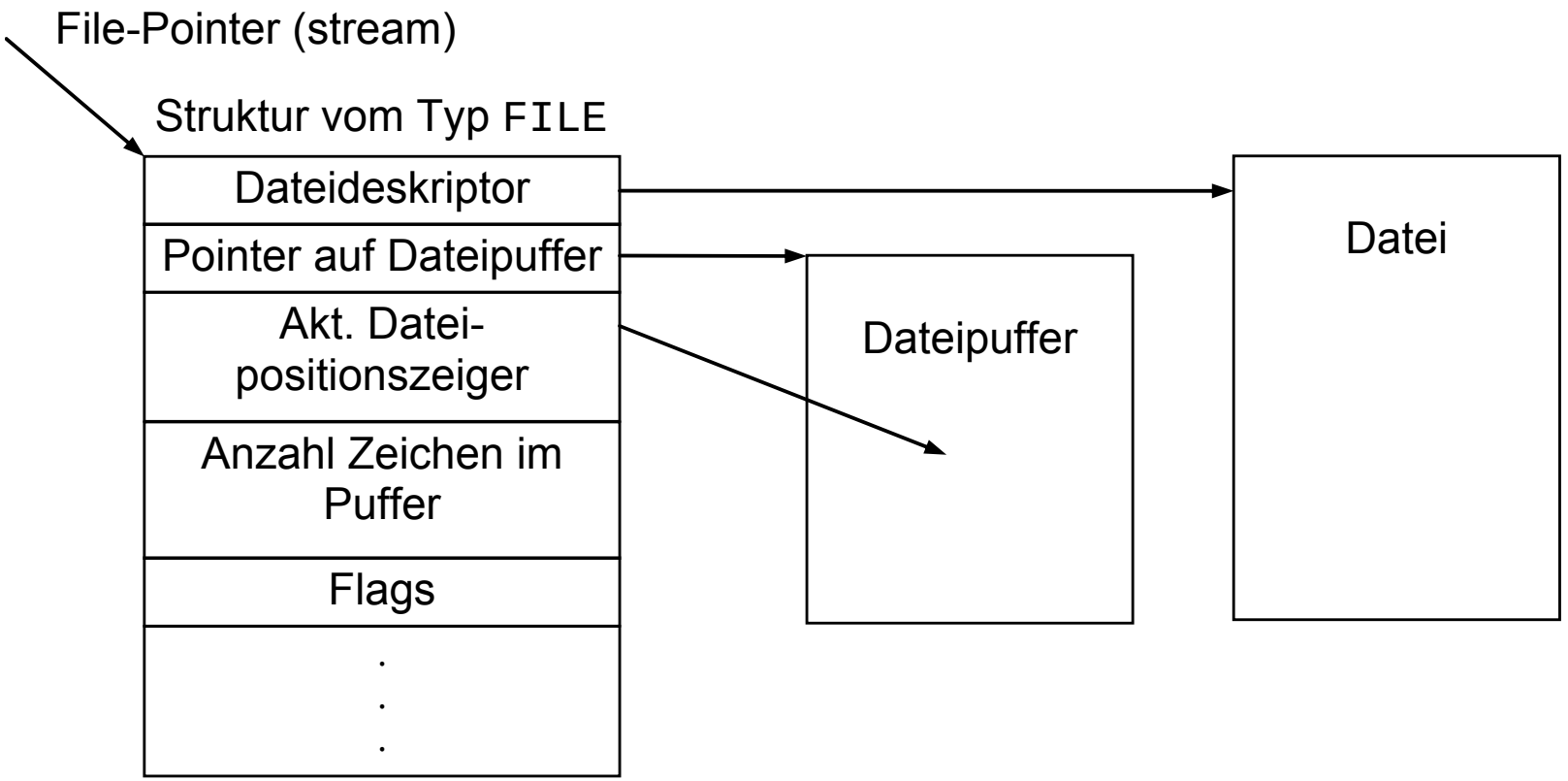
Öffnen einer Datei

```
FILE * fopen(const char *name, const char *mode);
```

`*name`	Dateiname, z.B. `"test.dat"` oder `"/tmp/test.dat"` oder auch `"C:\\temp\\test.dat"`
`*mode`	Zugriffsmodus
`r`	nur zum Lesen (Datei muß existieren)
`w`	nur zum Schreiben (wenn Datei existiert wird sie gelöscht)
`a`	nur zum anhängen (ggf. Datei anlegen)
`r+ w+ a+`	wie oben, wird jedoch immer zum Lesen **und** Schreiben
`b`	Binärmodus, Datenaustausch ohne Interpretation
`t`	Textmodus (Standard), '\n' wird interpretiert

Rückgabewert:
- Erfolg: Filepointer (`!= NULL`)
- Fehler: `NULL`

Informationen im Filepointer



Schließen einer Datei

```
int fclose(FILE *pDatei);
```

- Schreibpuffer wird zwingend auf Medium geschrieben
- Verzeichniseintrag wird aktualisiert
    - siehe auch `fflush(FILE *pDatei);`
- Filepointer freigeben
- Automatisches Schließen am Programmende
- Empfehlung: **sofort** nach Abschluss der Dateibearbeitung schließen
    - verhindert evtl. Datenverlust bei einem späterem Programmabsturz
    - Anzahl gleichzeitig geöffneter Dateien bleibt begrenzt

Ein-/Ausgabe-Funktionen in C

Verwendet Standard-Stream	Erfordert einen Stream-Namen	Beschreibung
`printf()`	`fprintf()`	Formatierte Ausgabe
`puts()`	`fputs()`	String-Ausgabe
`putchar()`	`putc(), fputc()`	Zeichenausgabe
`scanf()`	`fscanf()`	Formatierte Eingabe
`gets()`	`fgets()`	String-Eingabe
`getchar()`	`getc(), fgetc()`	Zeicheneingabe
`perror()`		String-Ausgabe an stderr

```c
#include <stdio.h>
#define STR_LEN 80

int main()
{
   char str[STR_LEN];
   FILE *fp;
   const char *const filename = "bsp.txt";

   /* Datei oeffnen und eine Zeile anhaengen: */
   if ((fp = fopen (filename, "a")) == NULL)
   {
      /* Fehlerbehandlung: */
      fprintf (stderr, "Fehler beim Öffnen der Datei '%s'\n",
               filename);
      return -1;
   }
   fprintf (fp, "Noch eine Zeile...\n");

   fclose (fp);
```

```
    ...

    /* Datei wieder oeffnen, alle Zeilen ausgeben: */
    if ((fp = fopen (filename, "r")) == NULL)
    {
       /* Fehlerbehandlung: */
       fprintf (stderr, "Fehler beim Öffnen der Datei '%s'\n",
                filename);
       return -1;
    }
    while (fgets (str, STR_LEN, fp))
       printf (str);
    fclose (fp);

    return 0;
}
```

Formatierte Ein-/Ausgabe: fscanf() und fprintf()

```
int fscanf (pDatei, "Formatstring", ...);
```

Rückgabewert: Anzahl ausgelesener und abgespeicherter Parameter (Erfolg)
EOF (Fehler)

```
int fprintf (pDatei, "Formatstring", ...);
```

Rückgabewert: Anzahl der geschriebenen Bytes (Erfolg)
EOF (Fehler)

Strings lesen/schreiben: fputs() und fgets()

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

int main()
{
   FILE *pdatei;
   char string[] = "Das ist ein Test";
   char puffer[20];

   pdatei = fopen("TEST.TXT", "w+");
   fputs(string, pdatei);

   fseek(pdatei, 0, SEEK_SET); // Positionszeiger zurücksetzen

   fgets(puffer, strlen(string)+1, pdatei);
   printf("%s", puffer);
   fclose(pdatei);

   return 0;
}
```

Zeichenweise Ein-/Ausgabe: getc() und putc()

```
FILE *quelle, *ziel;
int c;
char quelle[255], ziel[255];

printf("Name Quelldatei: ");
scanf("%s", quelle);
quelle = fopen(quelle, "rb");
if (!quelle)
{
   printf("Konnte %s nicht finden bzw. oeffnen!\n", quelle);
   return -1;
}
printf("Name Zieldatei: ");
scanf("%s", ziel);
ziel = fopen(ziel,"w+b");
if (ziel == NULL)
{
   printf("Konnte Zieldatei nicht erzeugen!\n");
   return -1;
}

while ((c = getc(quelle)) != EOF)
   putc(c, ziel);
```

Binärmodus: fread() und fwrite()

```
size_t fread(void *puffer, size_t blockgroesse,
                  size_t blockanz, FILE *stream);
```

`blockanz` Blöcke der Größe `blockgroesse` werden aus `stream` gelesen und in `puffer` abgelegt

Rückgabewert: Anzahl gelesener Blöcke

```
size_t fwrite(const void *puffer, size_t blockgroesse,
                   size_t blockanz, FILE *pdatei);
```

`blockanz` Blöcke der Größe `blockgroesse` werden aus `puffer` gelesen und nach `stream` geschrieben

Rückgabewert: Anzahl geschriebener Blöcke

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

typedef struct { int day, month, year; } DATE;

int main(void)
{
    DATE dat1 = {3, 02, 2014}, dat2;
    FILE *fp;
    char fname = "EXAMPE.BIN";

    fp = fopen(fname, "w+b");
    if (fp == NULL)
    {
        printf("Datei %s kann nicht geoeffnet werden.", fname);
        return -1;
    }
    fwrite(&dat1, sizeof(DATE), 1, fp);
    rewind(fp); /* rewind to the beginning */
    fread(&dat2, sizeof(DATE), 1, fp);
    printf("%d.%d.%d", dat2.day, dat2.month, dat2.year);
    fclose(fp);
    return 0;
}
```

Wahlfreier Zugriff

```
void rewind(FILE *stream);
```

Dateipositions-Zeiger auf Stream-Anfang setzen

```
long ftell(FILE *stream);
```

ermittelt aktuelle Dateiposition (in Bytes bezogen auf Dateianfang)

```
int  fseek(FILE *stream, long offset, int whence);
```

springt an beliebige Dateiposition, Markierung für nächste Operation

`offset` Zielposition, `whence` bestimmt Interpretation von `offset`:

```
#define SEEK_SET 0    offset bzgl. Dateianfang
#define SEEK_CUR 1    offset relativ zur aktuellen Dateiposition
#define SEEK_END 2    offset bzgl. Dateiende
```

```
int  feof(FILE *stream);
```

Abfrage auf Dateiende (1 ≙ Dateiende erreicht, 0 sonst)

Weitere Funktionen

```
int remove(const char *dateiname);
int rename(const char *altname, const char *neuname);
int fflush(FILE *stream);
int ungetc(int c, FILE *stream);
```

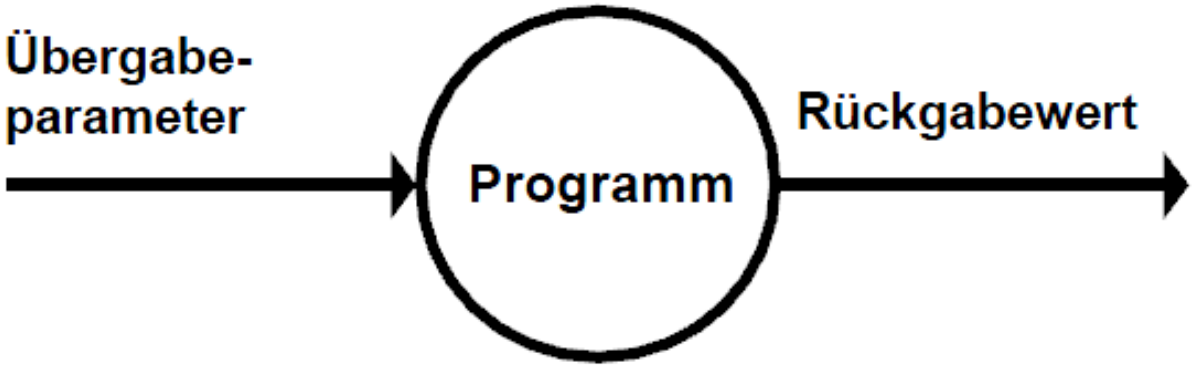
- Fehlerbehandlung:

```
extern int errno;                    // in errno.h
char * strerror(int errno);          // in string.h
void perror(const char *message); // in stdio.h
```

- `errno` von Bibliotheksfunktionen im Fehlerfall gesetzt
- `strerror()` liefert Fehlerstring zur Fehlernummer `errno`
- `perror()` liefert eine Fehlermeldung an `stderr`, (bezogen auf die letzte fehlgeschlagene Bibliotheksfunktion)

Ausgabeformat: `<meldung>:<Fehlerstring>`

Kommandozeilenparameter



Parameter der main()-Funktion

- Funktionskopf der main()-Funktion ohne Parameter:

```
int main(void);
```

- Kommandozeilenparameter am Beispiel des Shell-Kommandos "cp":

```
cp * ~\myfiles
```

`cp`	ist der Name des Programms
`*`	ist der 1. Kommandozeilenparameter
`~\myfiles`	ist der 2. Kommandozeilenparameter

→ Verarbeitung der Parameter erfordert modifizierte `main()`-Funktion:

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

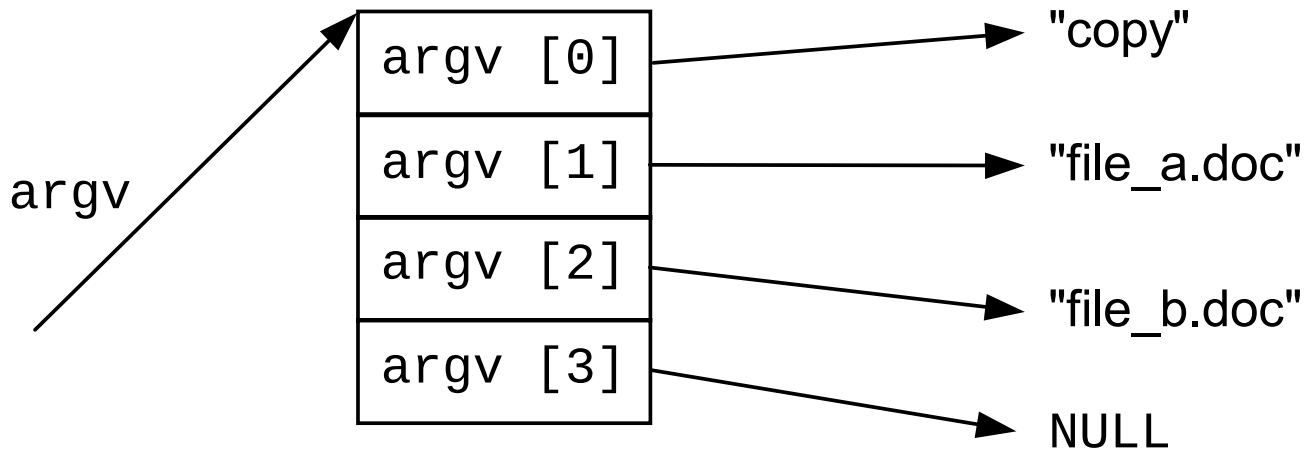
`argc` (argument count) Anzahl der Kommandozeilenparameter + 1

`argv` (argument value) referenzieren der Reihe nach den Programmnamen bzw. Kommandozeilenparameter

Parameter der main()-Funktion

- Beispiel:

```
C:\>copy file_a.doc file_b.doc
```



- Alternativen:

```
char *argv[]        oder  char **argv
```

exit() und atexit()

```
#include <stdlib.h>

void exit(int status);
```

Programm "geordnet" abbrechen und Statuswert liefern (wie `return`)

`return` aktueller Funktionsaufruf wird beendet
`exit()` das Programm selbst wird abgebrochen

```
int atexit (void (*func)(void));
```

registriert eine Funktion, die bei einem normalen Programmabbruch aufgerufen wird (bei `exit()` oder `return` aus `main()`)

Registrierte Funktionen werden dann in umgekehrter Reihenfolge ihrer Registrierung aufgerufen

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

void final_1(void)
{
   printf("\nfinal_1 ist dran");
}

void final_2(void)
{
   printf("\nfinal_2 ist dran");
}

int main(void)
{
   atexit(final_1);
   atexit(final_2);
   printf("\nmain ist dran");
   exit(EXIT_SUCCESS);  /* aequivalent zu return EXIT_SUCCESS  */
}
```

Exit-Code

- ist der von `main()` mittels `return` zurückgelieferte `int`-Wert
- Wert wird an Betriebssystem (bzw. Shell) zurückgegeben
- Abfrage in der Shell:

```
echo $?
```

- Windows-Betriebsysteme:

```
echo %errorlevel%
```

Hinweis: Kommandozeilenparameter können in einer IDE simuliert werden, um die Arbeit mit dem Debugger zu vereinfachen. (Bsp.: Visual Studio "Projekt/Einstellungen/Debug/Programmargumente")