Teil 7: Ein-/Ausgabe und Präprozessor

Gliederung

Dateien

Streams

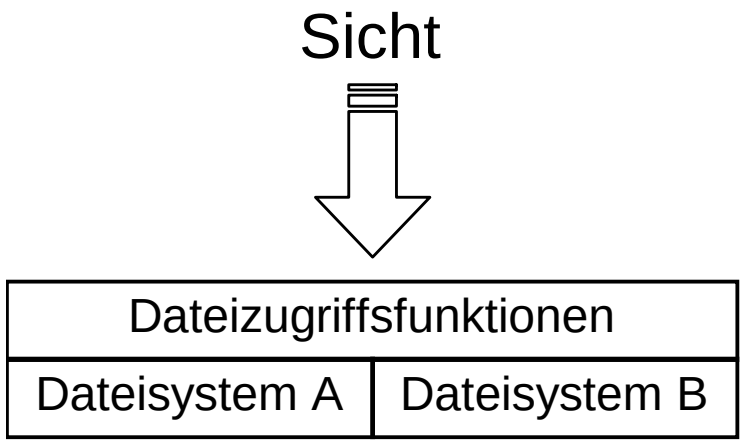
Dateioperationen

Präprozessor

Dateien

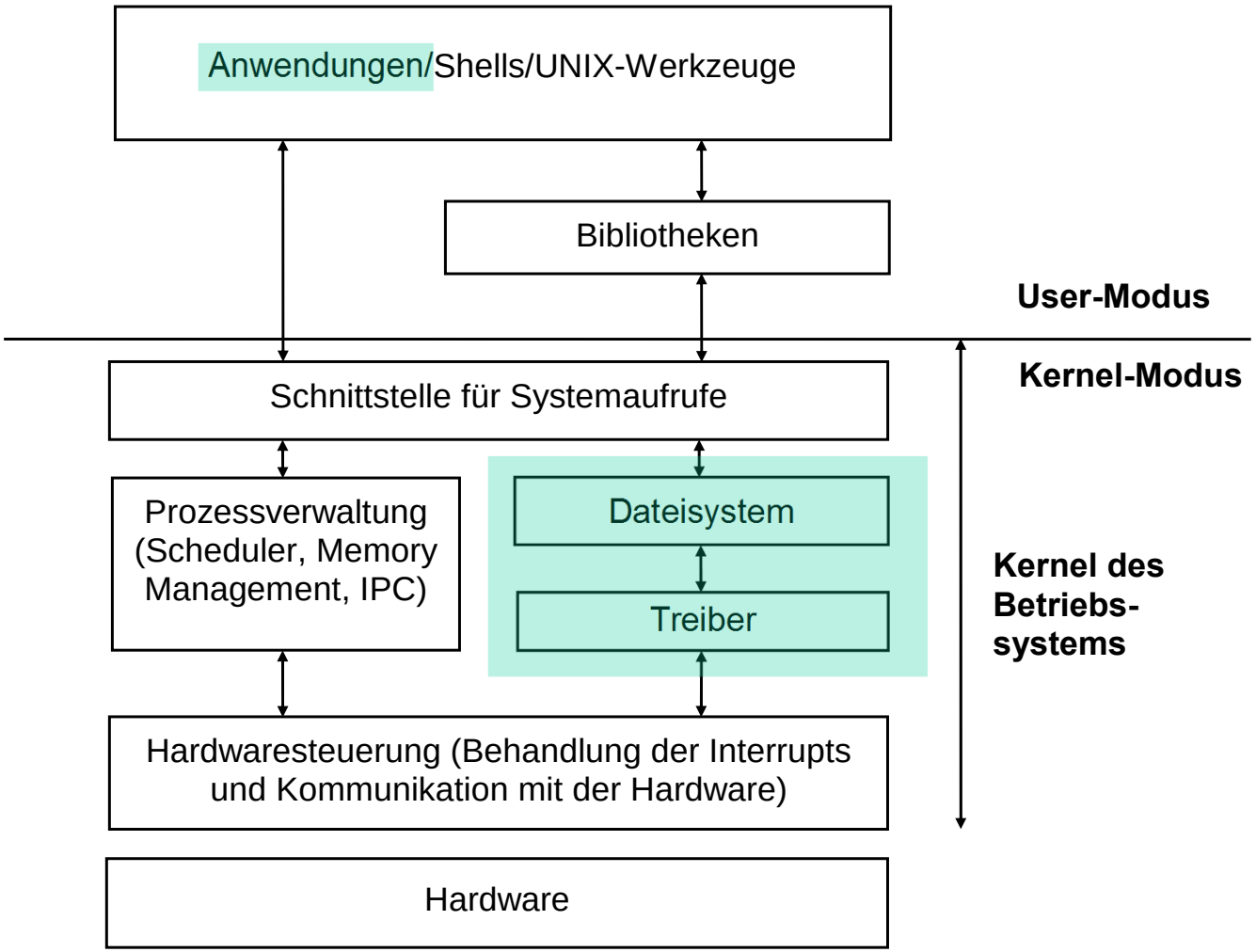
Dateien und Dateisysteme

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

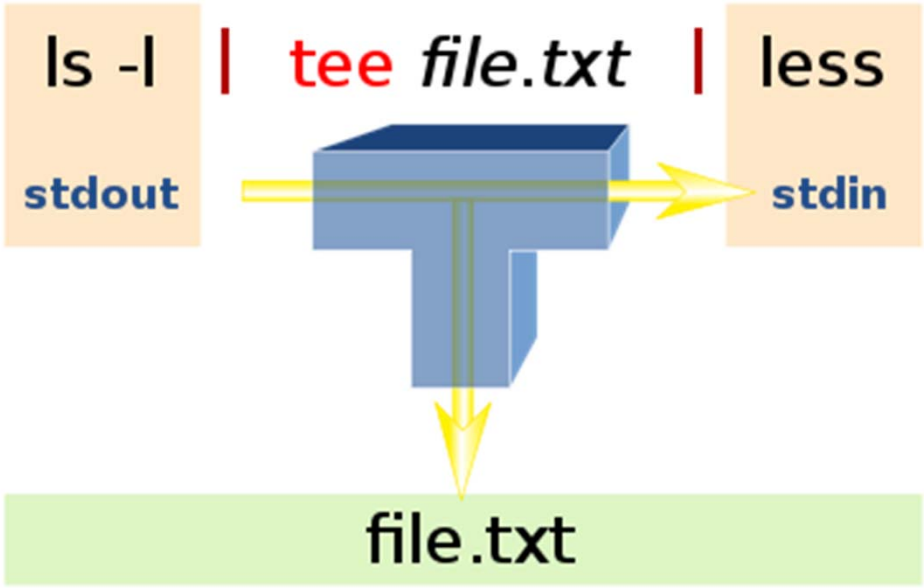


Schichtenmodell (Unix)

- Geräteabhängigkeit ist durch das Dateisystem verborgen

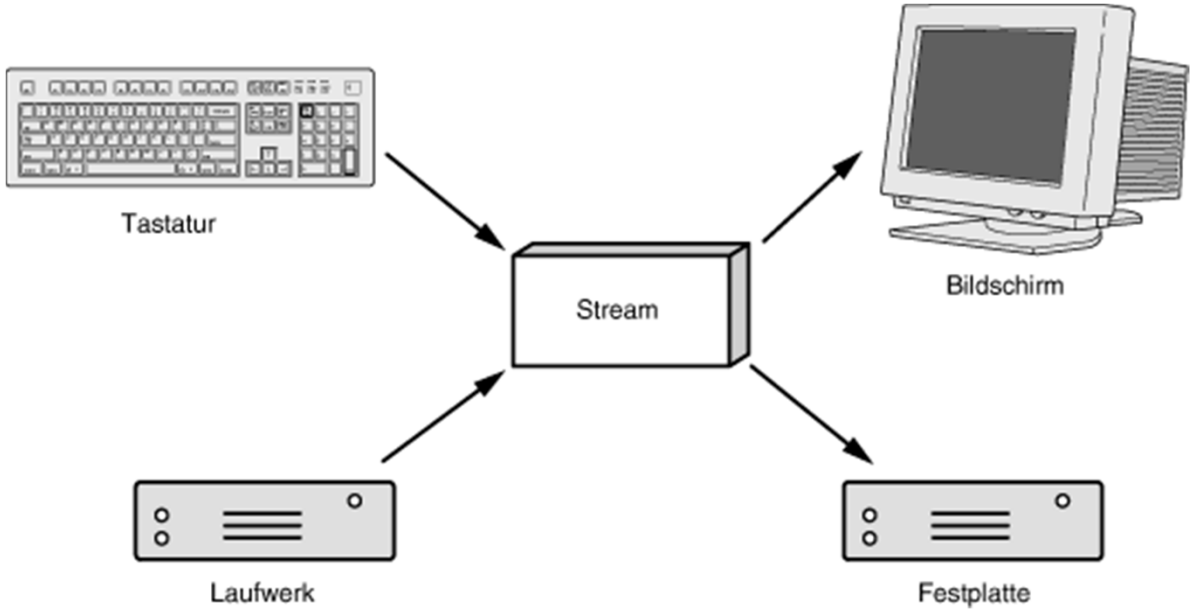


Streams



Stream-Konzept

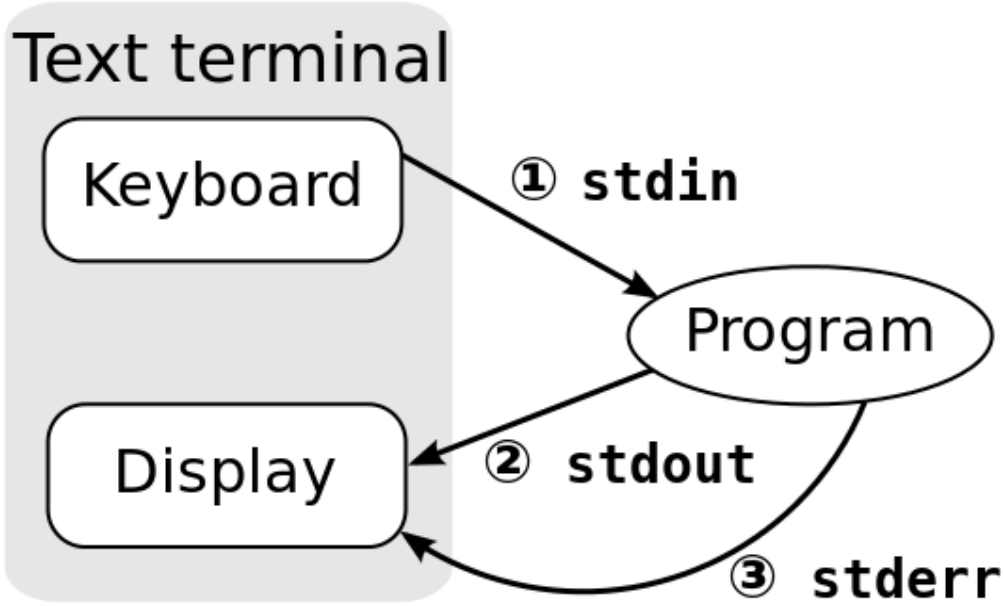
- Eingabe – Verarbeitung – Ausgabe (EVA-Prinzip)
- **Streams**: abstraktes Modell von Datenströmen
  - geordnete Folge von Bytes
- **Datenquelle → Datenstrom → Datenziel**
  - Quelle und Ziel: Zuordnung zu Datei oder Gerät



Standardkanäle (bzw. Standard-Streams)

- In jedem C-Programm sind 3 **Standardkanäle** vorhanden:

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



Umleitung durch das Betriebssystem

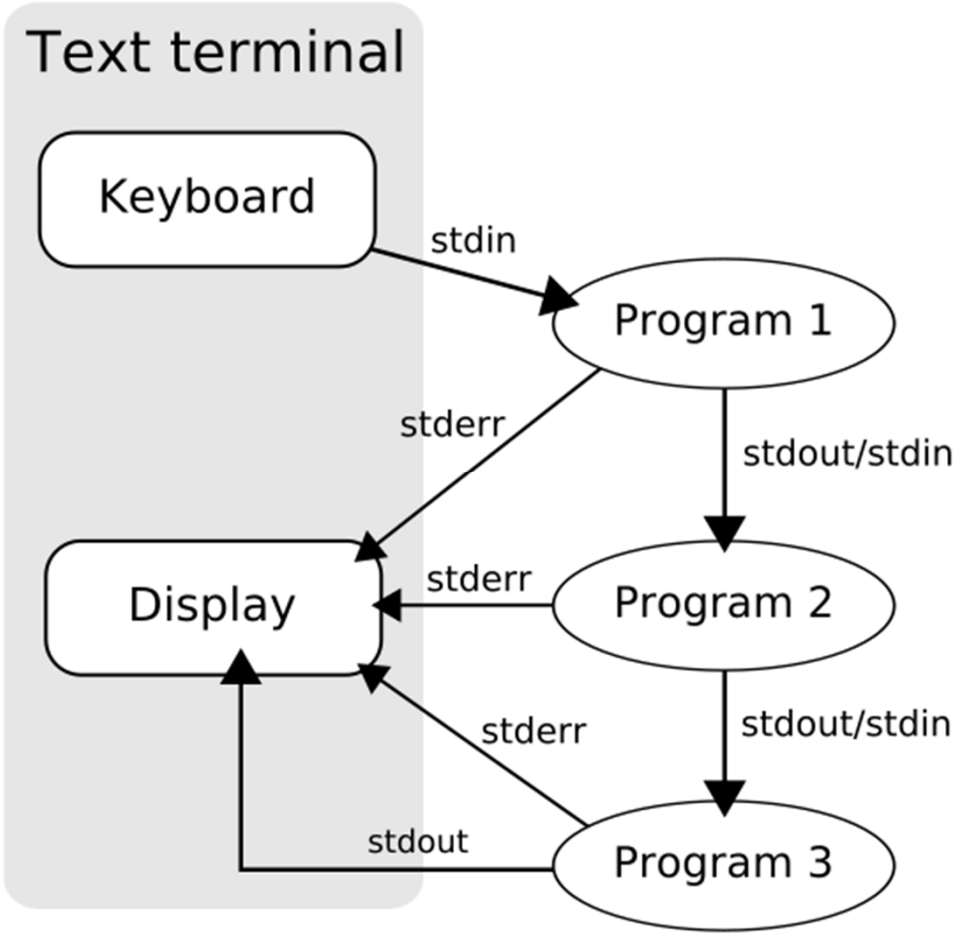
- **Umlenkung** (Umleitung) der Ein- und Ausgabe

```
myprog > test.out
```

- **Pipelining**

```
prog1 | prog2 | prog3
```

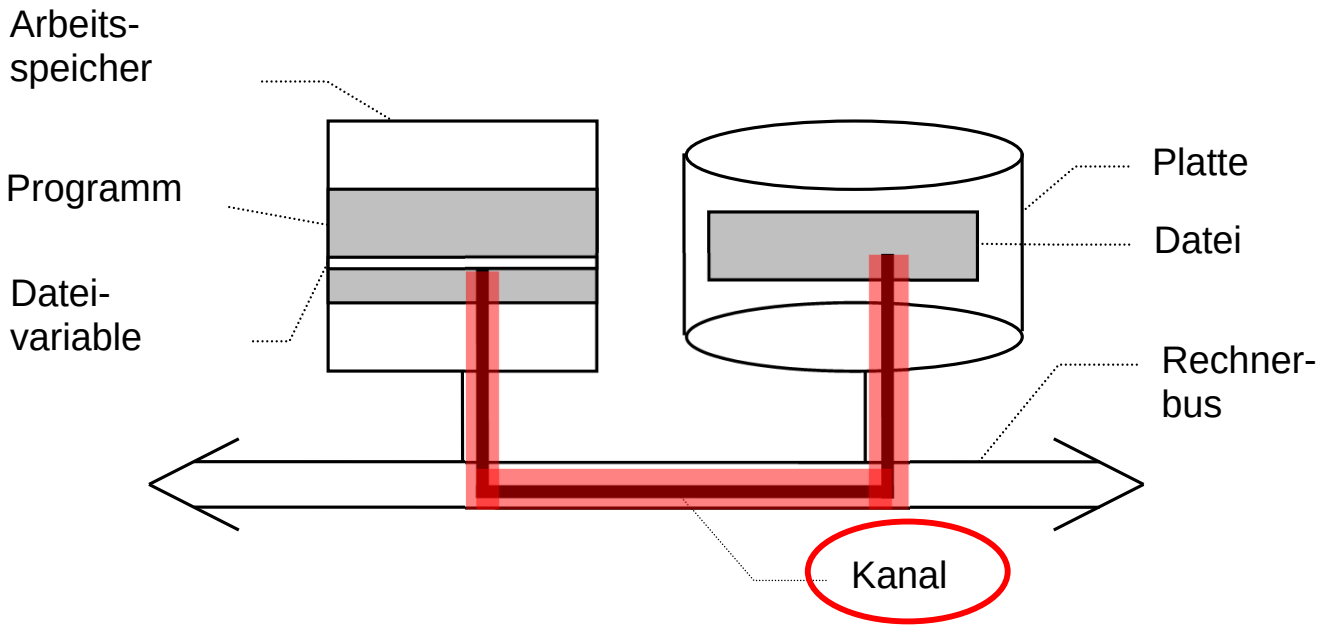
- Umlenkung / Pipelining sind für das Programm transparent



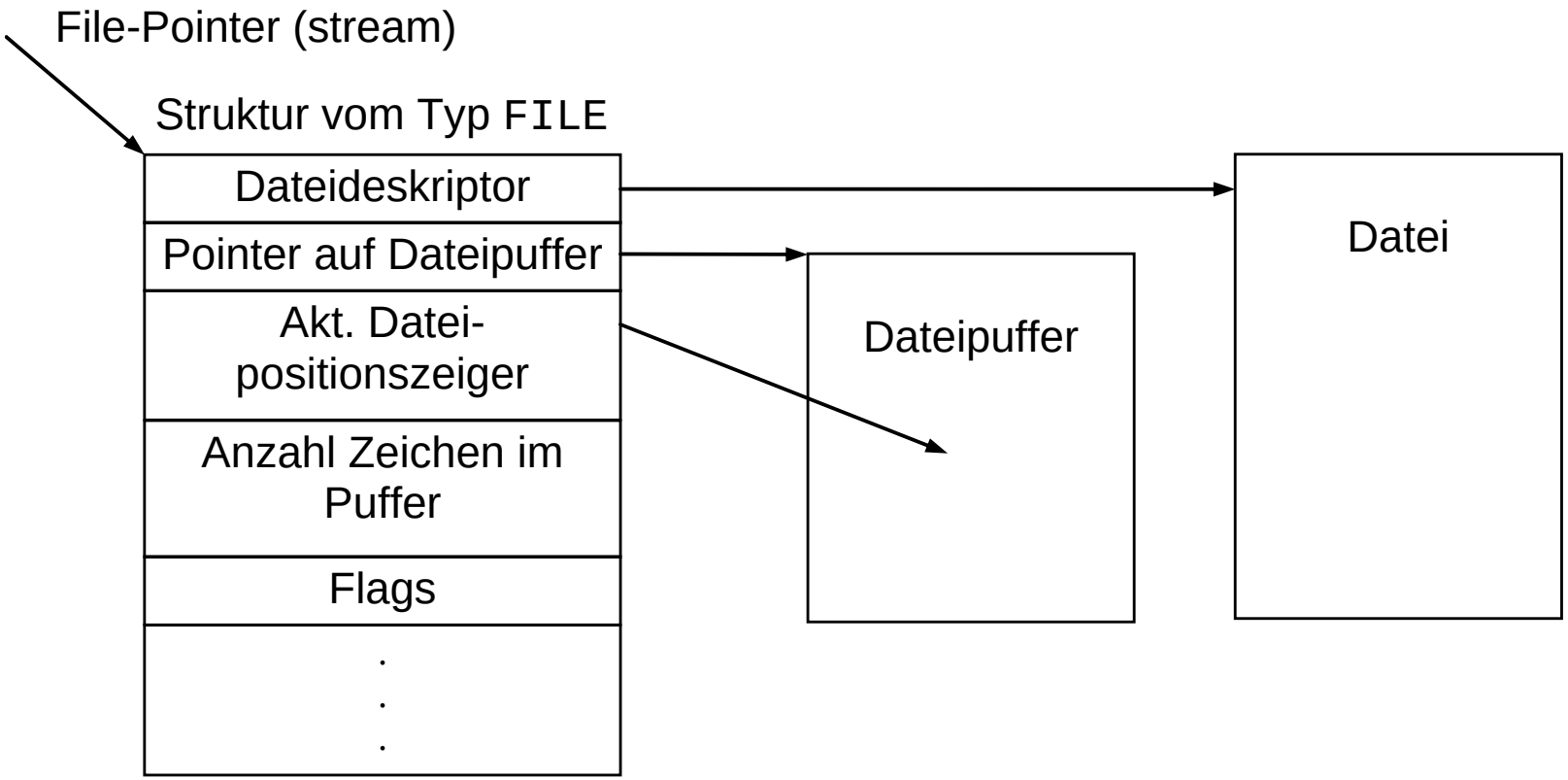
Eigene Streams (Dateien) in C-Programmen

- Bibliothek `<stdio.h>`
- Erzeugung: Erstellung einer Dateivariable (**File-Pointer**)
- Verknüpfung von Dateivariable und physikalischer Datei

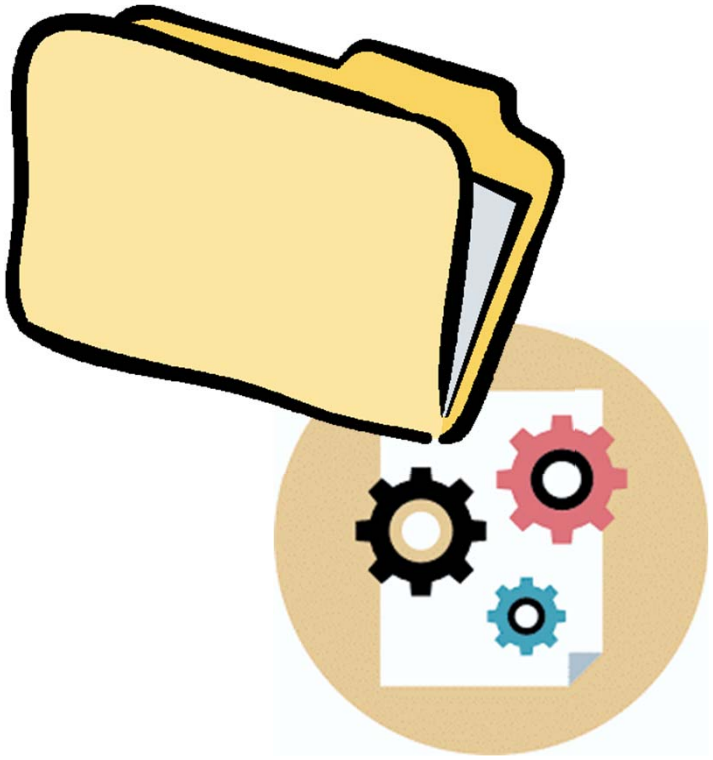
```
fp = fopen("TEST.DAT", "w");
```



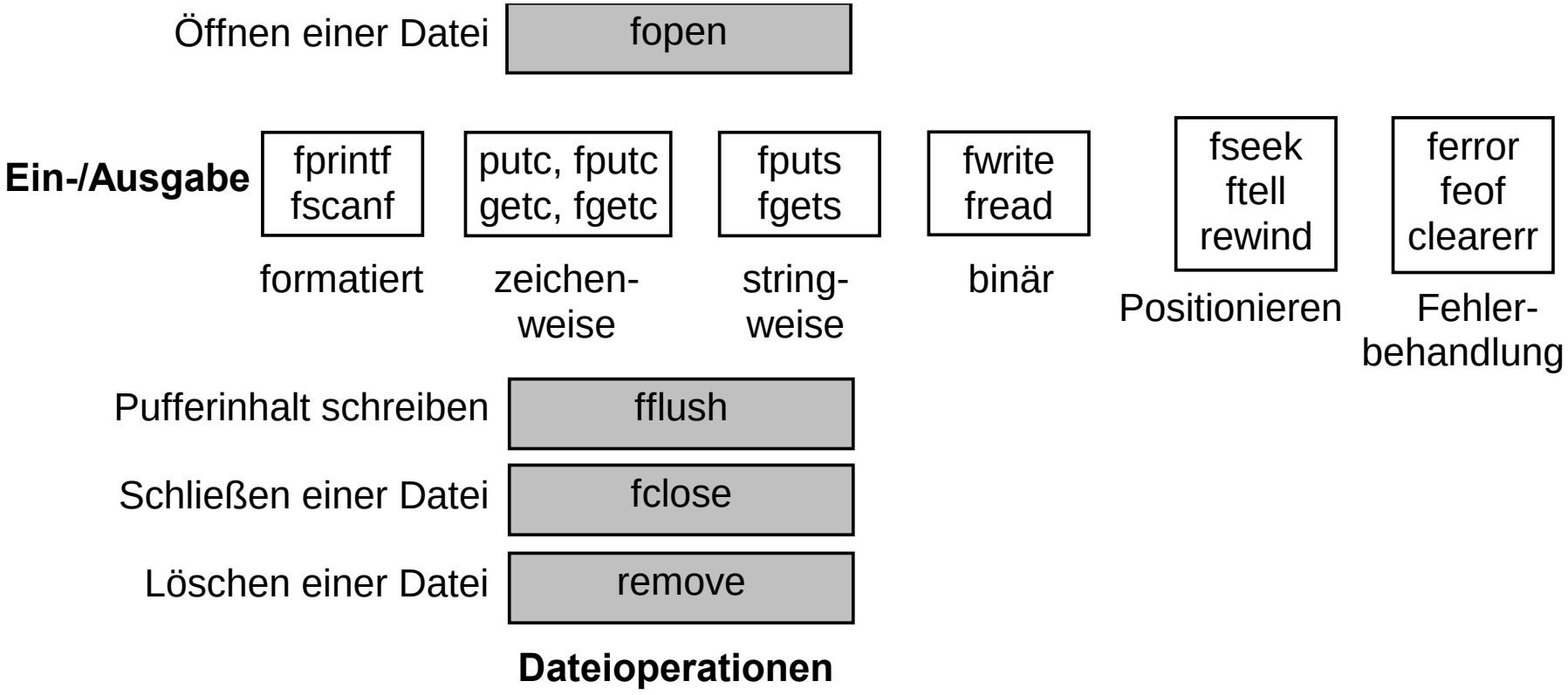
Informationen im File-Pointer



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 (Datei wird überschrieben)
`a`	nur zum anhängen (ggf. Datei anlegen)
`r+ w+ a+`	wie oben, jedoch immer zum Lesen **und** Schreiben
`b`	Binärmodus, Datenaustausch ohne Interpretation
`t`	Textmodus (Standard), '\n' wird interpretiert

Rückgabewert

Erfolg:	gültiger File Pointer `FILE *`
Fehler:	`NULL`

Schließen einer Datei

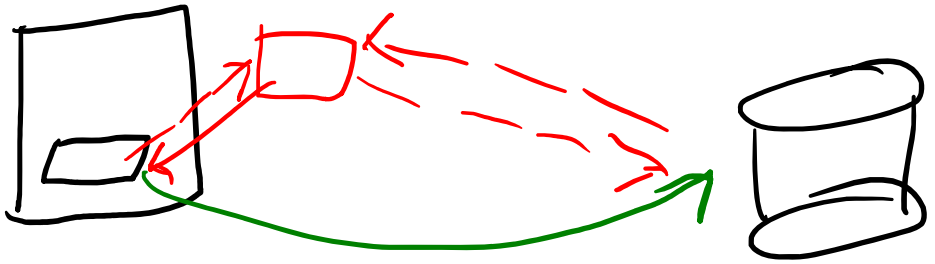
```
int fclose(FILE *pDatei);
```

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

Ein-/Ausgabe Operationen

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

Datei-Operationen



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

int main()
{
   FILE *fp;
   char *filename = "bsp.txt";

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

   // Datei schliessen
   fclose(fp);
```

```
    ...

    // Datei wieder oeffnen und alle Zeilen ausgeben
    if ((fp = fopen (filename, "r")) == NULL)
    {
       // Fehlerbehandlung
       fprintf(stderr, "Fehler beim Öffnen der Datei '%s'\n",
               filename);
       return EXIT_FAILURE;
    }

    char str[STR_LEN];
    while(fscanf(fp, "%s", str) != EOF)
       printf ("%s", str);

    // Datei schliessen
    fclose (fp);

    return EXIT_SUCCESS;
}
```

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

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

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

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

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

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

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

int main()
{
   FILE *pdatei;
   char testString[] = "Das ist ein Teststring";
   char puffer[STR_LEN];

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

   fputs(testString, pdatei);

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

   fgets(puffer, STR_LEN, pdatei);

   printf("%s\n", puffer);

   fclose(pdatei);
   return EXIT_SUCCESS;
}
```

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

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

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

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

Lesen/Schreiben im 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()
{
    DATE datum1 = {27, 1, 2023};
    DATE datum2;
    FILE *fp;
    char *filename = "EXAMPE.DAT";

    fp = fopen(filename, "w+b");
    if (fp == NULL)
    {
        fprintf(stderr, "Fehler beim \x99ffnen der Datei %s.", filename);
        return EXIT_FAILURE;
    }

    // Datum in Datei schreiben
    fwrite(&datum1, sizeof(DATE), 1, fp);
...
```

```
...

    // Datei-Posistionszeiger wieder an den Anfang setzen
    rewind(fp);

    // Datum aus Datei lessen und ausgeben
    fread(&datum2, sizeof(DATE), 1, fp);
    printf("%d.%d.%d", datum2.day, datum2.month, datum2.year);

    fclose(fp);

    return EXIT_SUCCESS;
}
```

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` ist Zielposition, bezogen auf den Wert von `whence`:

```
#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 Dateifunktionen

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

→ (putc , getc)

- Fehlerbehandlung:

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

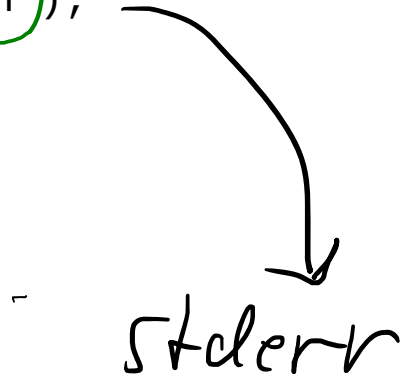
- `errno` von Bibliotheksfunktionen im Fehlerfall gesetzt
- `strerror()` liefert Fehlerstring zur Fehlernummer `errno`

- `perror()` liefert eine Fehlermeldung an `stderr`, (bezogen auf die letzte fehlgeschlagene Bibliotheksfunktion) im folgenden Ausgabeformat:

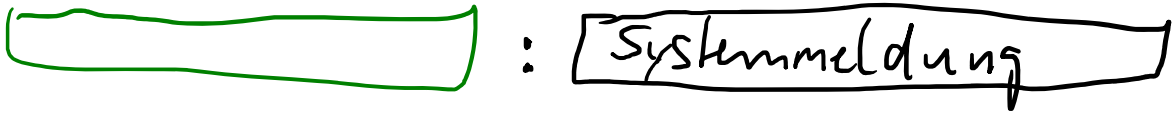
`<message>`: `<Fehlerstring der Bibliothek>`

Da ist was schief gegangen : Access denied ...

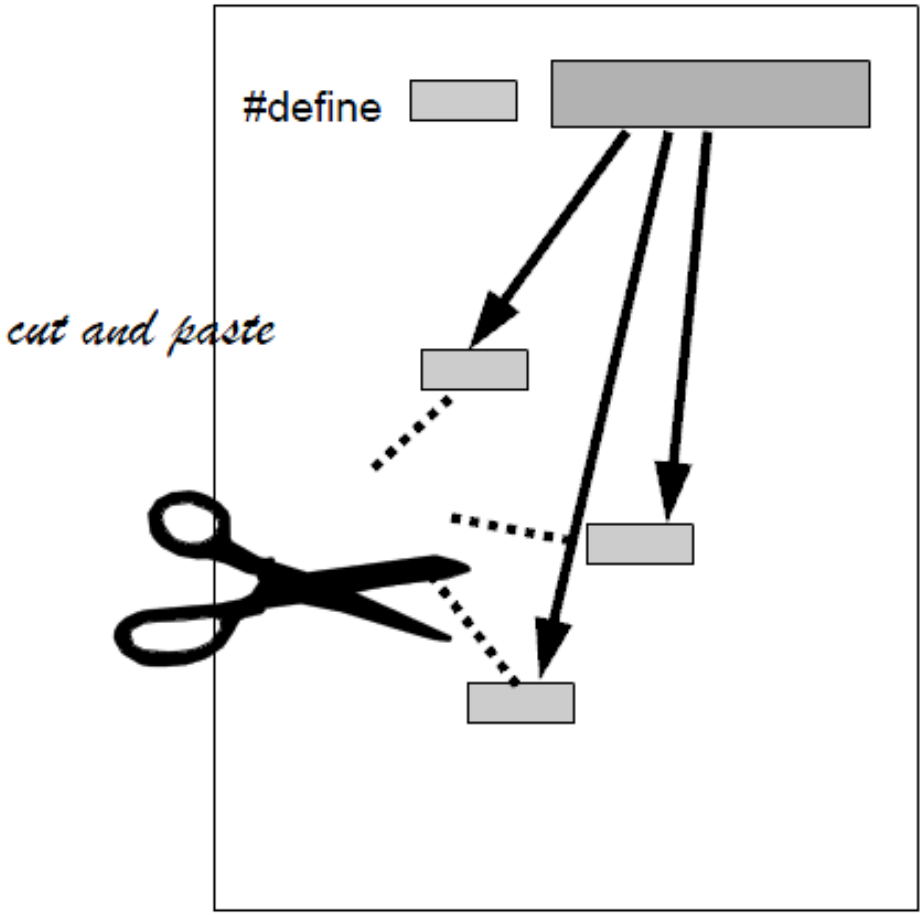
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
   FILE *fp;

   fp = fopen("fehlendeDatei.dat", "r");
   if (fp == NULL)
   {
      perror("FEHLER! Kann nicht aus Datei lesen");
      return EXIT_FAILURE;
   }
   return EXIT_SUCCESS;
}
```




```
FEHLER! Kann nicht aus Datei lesen: No such file or directory
```

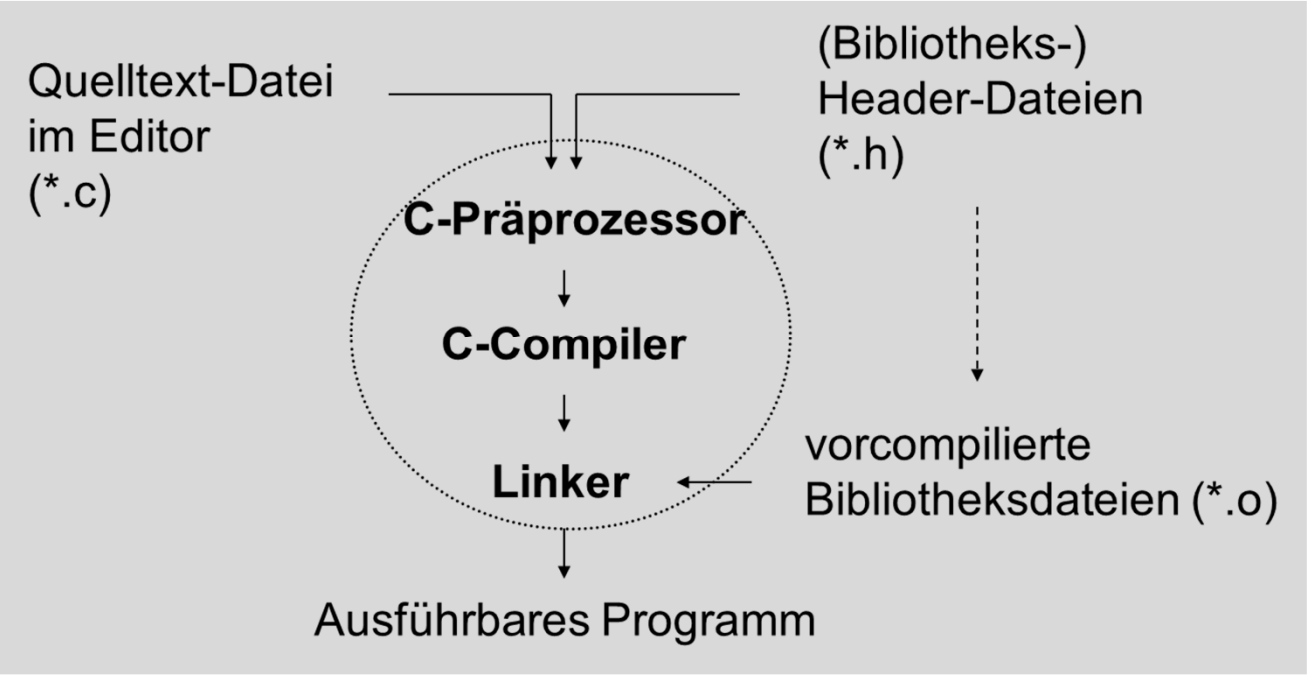


Präprozessor



Aufgaben des Präprozessors

- Einfügen von Bibliotheken und Modulen (`#include <…>`)



- Ersetzen von Text (Symbolische Konstanten, Makros)
- bedingte Kompilierung

Direktiven

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

#include Direktive

- Einbinden von ANSI-C Standard-Bibliotheken:

```
#include <dateiname>
```

- Eigene Headerdateien:

```
#include "dateiname"
```

- Absolute Pfadangaben sind Plattformabhängig:

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

#define Direktive

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

```
#define NAME Ersatztext
```

- Beispiel:

```
#define PI 3.14159

int main()
{
   printf("Pi hat den Wert: %f", PI);
   ...
}
```

- Verschachtelung möglich

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

Vorteil: Auswertung des Ausdrucks ggf. schon bei der Kompilierung

Makros

- Name enthält Parameterliste, Argumente werden im Ersatztext ebenfalls ersetzt

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

- Beispiele:

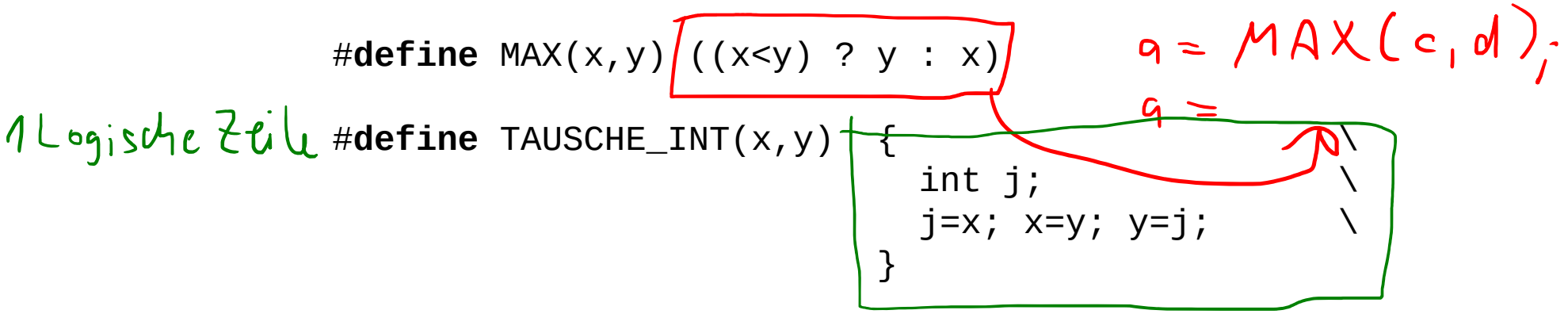
```
#define SUM(n1, n2) (n1 + n2)
```

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 (I)

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

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

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

Makroauflösung durch Präprozessor liefert:

```
z1 =  x * y;
z2 = x + 1 * y - 1;
```

korrekt wäre jedoch:

```
z2 = (x + 1) * (y - 1);
```

Makros: Fehlerquellen (II)

Abschluss des Makros mit einem Semikolon:

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

Hier darf **kein** Semikolon stehen

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 für erweitere Fehlerausgabe nutzen

```
#include <stdio.h>

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

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

#ifdef __STDC__
   fprintf(stderr, "Standard-C-Compiler!\n");
#else
   fprintf(stderr, "Kein Standard-C-Compiler!\n");
#endif

   return 0;
}
```

Bedingte Compilierung

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

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("Summe von a und b: %d\n", c);

   return 0;
}
```

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("Summe von a und b: %d\n", c);

   return 0;
}
```

Beispiel: Fehlersuche

```c
#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("Summe von a und b: %d\n", 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("Summe von a und b: %d\n", c);

   return 0;
}
```

Beispiel: Hardware / Betriebssystem Spezifika

```
#include <stdio.h>

int main()
{
#ifdef WINDOWS
  printf("Programmteil fuer 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 fuer Intel Prozessoren\n");
  printf("...\n");
#else
  printf("Programmteil fuer alle anderen Prozessoren\n");
  printf("...\n");
#endif
   return 0;
}
```

```c
#include <stdio.h>
#define LINUX
#define CPU AMD

int main()
{
#ifdef WINDOWS
   printf("Programmteil fuer 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 fuer Intel Prozessoren\n");
   printf("...\n");
#else
   printf("Programmteil fuer alle anderen Prozessoren\n");
   printf("...\n");
#endif
   return 0;
}
```

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

int main()
{
#ifdef WINDOWS
   printf("Programmteil fuer 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 fuer Intel Prozessoren\n");
   printf("...\n");
#else
   printf("Programmteil fuer alle anderen Prozessoren\n");
   printf("...\n");
#endif
   return 0;
}
```

- resultierender Quellcode für die eigentliche Kompilierung:

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

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

- Weitere Anwendung z.B. als "Include Guard"
  https://en.wikipedia.org/wiki/Include_guard