

# Syntaxhilfe zu ANSI-C und Sytemprogrammierung unter UNIX

Vollständigkeit und Korrektheit der Angaben sind nicht garantiert. Verbesserungsvorschläge sind jederzeit willkommen (mail an [talala@wirtschaftsphysik.de](mailto:talala@wirtschaftsphysik.de))! Seitenzahlen beziehen sich auf das „Systemnahe Software“ Skript.

## accept

```
#include<sys/types.h>
#include<sys/socket.h>
int accept(int sd, struct sockaddr *address, int addresslen)
S. 124
```

## Arrays

### Prototyp:

```
int vector[length]
char vector [length]
```

### Beschreibung:

Vektornamen sind Zeiger auf das erste Element

## atoi

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

### Prototyp:

```
int atoi(const *s);
```

### Returnwert:

Gibt den int-Wert der in der Zeichenkette gespeicherten Zahl zurück. Falls Auswertung nicht möglich, wird 0 zurückgegeben.

### Beschreibung:

Wandelt eine als Zeichenkette gespeicherte dezimale Ganzzahl in einen int-Wert um. Beim ersten nicht interpretierbaren Zeichen wird die Umwandlung abgebrochen. Ein Dezimalpunkt oder Exponent wird nicht als Teil der Zahl angesehen.

### Beispiel:

```
char *s= "+123 Biere";
printf("atoi(s) = %d\n",atoi(s));
/* Ausgabe: atoi(s) = 123 */
```

## Argumentverarbeitung

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

### Prototyp:

```
void main(int argc, char *argv[], char **env) //auch **argv möglich
```

### Beschreibung:

argc liefert die Anzahl der Argumente auf der Kommandoebene inklusive des Kommandonamens

argv ist ein Zeiger auf einen Vektor mit Zeichenketten, die die einzelnen Argumente enthalten, ein Argument pro Zeichenkette

env liefert die Umgebungsvariablen

argv[0] ist der Kommandoname

\*++argv / argv[i++] liefern beide das nächste Argument

argv[0][0] liefert den ersten Buchstaben des Kommandonamens

## bind

```
#include<sys/types.h>
#include<sys/socket.h>
int bind(int sd, struct sockaddr *address, int addresslen)
    sd:          Socket descriptor
    struct sockaddr:  Adresse
    addresslen:  Länge der Adresse
genauer: Skript S. 121
```

## close

```
#include <unistd.h>
```

### Prototyp:

```
int close(int fd /*filedescriptor*/)
```

### Rückgabewert:

Bei Erfolg 0 oder bei Fehler -1

### Beschreibung:

Beendet I/O Verbindung zwischen existierendem File und dem ausführenden Prozess

## connect

```
#include<sys/types.h>
#include<sys/socket.h>
int connect (int sd, struct sockaddr *address, int addresslen)
S. 123
```

## const

### Prototyp:

```
const int i 300
```

## dup

### Prototyp:

```
int dup (int fd)
```

### Rückgabewert:

dup liefert einen neuen Filedescriptor oder -1 bei einem Fehler

### Beschreibung:

dup verdoppelt einen bestehenden Filedeskriptor und liefert als Resultat einen neuen Filedeskriptor mit der kleinsten verfügbaren Nummer, der mit der gleichen Datei oder der gleichen Pipe verbunden ist. Beide FD haben den selben Positionszeiger. Damit kann z.B. ein FD mit der Nummer 0 erhalten werden, falls dieser vorher geschlossen (`close(0)`) wurde.

## Exec - Familie

### Prototyp:

```
execl(„/bin/echo“, „echo“, „the“, „lazy“, „dog“, (char*) 0);
```

### Beschreibung

`fflush(...)` nicht vergessen! `exec..` überlagert den Kontext des ausführenden Prozesses und bringt anschließend die neuen Instruktionen zur Ausführung. Deshalb im Zweifel immer nach einem `fork()` aufrufen.

```
int execl(char *path,                // Pfad des Programmes
          char *arg0, ...            // Argumente (1. Kommandoname, 2. ... dazugeh. Argumente
          (char*)0                  // NULL
          );
```

```
int execlp(char *filename,           // Name des Programmfiles (aktueller Pfad ????)
          char *arg0,1,...,n        // NULL
          (char*)0
          );
```

```
int execlp(char *path,              // Pfad des Programmes
          char *arg0, ...            // Argumente (1. Kommandoname, 2. ... dazugeh. Argumente
          (char*)0                  // NULL
          char **envp               // Zeiger auf die Umgebungsvariablen
          );
```

```
int execlp(char *path,              // Pfad des Programmes
          char *argv[];             // Zeiger auf einen Vektor mit Argumenten
          );
```

```
int execlp(char *filename,          // Name des Programmfiles
          char *argv[],             // Zeiger auf einen Vektor mit Argumenten
          );
```

```
int execlp(char *path,              // Pfad des Programmes
          char *argv[],             // Zeiger auf einen Vektor mit Argumenten
          char *envp[]              // Zeiger auf die Umgebungsvariablen
          );
```

## exit

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

### Prototyp:

```
void exit(int status);
```

### Rückgabewert:

keiner

### Beschreibung:

beendet ein Programm. Dateipuffer werden übertragen und Dateien werden geschlossen. **status** stellt den Exit-Code dar.

### Beispiel:

```
exit(1);
/* beendet das Programm und liefert den Exit-Code 1 zurück */
```

## fclose

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

### Prototyp:

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

**Rückgabewert:**

Datei geschlossen: 0; ungültiger File-Pointer: **EOF**

**Beschreibung:**

schließt die mit *fp* verbundene Datei überträgt den Inhalt des Dateipuffers. Der Speicherbereich des E/A-Puffers wird wieder freigegeben.

**Beispiel:**

```
if (fclose(fp) == 0)
printf("Datei wurde geschlossen\n");
```

**fflush**

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

**Prototyp:**

```
int fflush(FILE *fp);
```

**Rückgabewert:**

Fehlerfall: **EOF**; andernfalls: 0

**Beschreibung:**

überträgt den Inhalt des Dateipuffers:

Schreibmodus:

Dateipuffer wird geschrieben

Lesemodus:

Puffer wird geleert

Das Argument von fflush(arg) kann auch z.B. STDOUT sein

**Beispiel:**

```
if (fflush(fp) == EOF)
printf("Puffer nicht geschrieben\n");
```

**fgetc**

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

**Prototyp:**

```
int fgetc(FILE *fp);
```

**Rückgabewert:**

gelesenes Zeichen, Fehler oder Dateiende: **EOF (= - 1)**

**Beschreibung:**

liest aus der Datei das Zeichen von der aktuellen Position (Position wird 1 erhöht)

**Beispiel:**

```
c = fgetc(fp); /* weist c das Zeichen der aktuellen Position zu */
```

**fgets**

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

**Prototyp:**

```
char *fgets(char *string, int n, FILE *fp);
```

**Rückgabewert:**

Fehlerfall: **NULL**-Pointer; andernfalls: Zeiger *string*

**Beschreibung:**

liest Zeichenfolge aus der Datei und kopiert sie in einen durch string adressierten Speicherbereich; am Ende wird "\0" angefügt. Eingelesen wird bis n-1. Zeichen oder bis "\n".

**Beispiel:**

```
if (fgets(name,21,fp) != NULL) printf("1. Name: %s\n", name);
/* Ausgabe: 1. Name: xxxx */
```

**fopen**

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

**Prototyp:**

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

**Rückgabewert:**

File-Pointer

**Beschreibung:**

öffnet eine Datei *name* in der Zugriffsart *modus*. Durch Anhängen des Buchstabens "t" bzw. "b" an modus kann zwischen Binär- und Textmodus gewählt werden. **FILE** ist in **stdio.h** definiert. Liste der verschiedenen Modi:

"r"

Lesen

"r+"

Lesen / Schreiben

"w"

Schreiben (wenn Datei nicht existiert, wird sie angelegt)

"w+"

Lesen / Schreiben (wenn Datei nicht existiert, wird sie angelegt)

"a"

Schreiben ab Dateiende

"a+"

Schreiben ab Dateiende / Lesen

bei r und r+ muss die Datei existieren

**Beispiel:**

```
fp = fopen("bsp.txt","r");
/* öffnet die bereits existierende Datei bsp.txt zum Lesen (fp wird mit der Datei
verbunden) */
```

**for****Prototyp:**

```
for (init Ausdruck; term Ausdruck; increase Ausdruck)
    {Anweisung}
```

**Beschreibung:**

kann mit break oder return verlassen werden

**Beispiel:**

```
for (i = 0; i = 5; i++)
    {...}
```

**fork**

```
#include <unistd.h>    ????
```

**Prototyp:**

```
int fork()
switch (pid=fork()){}
```

**Rückgabewert:**

ProzessID des Sohnes an den Vaterprozess und 0 an den neu entstandenen Prozess (also im Erfolgsfall), -1 bei Fehler

**Beschreibung:**

Der neu erzeugt Prozess wird als Kind-Prozess, der aufrufende als Vater-Prozess bezeichnet.

fork wird einmal aufgerufen und kehrt im Erfolgsfall zweimal zurück: der Aufrufer erhält als Rückgabewert die PID des Sohnes, der Kindprozess erhält 0; im Fehlerfall (z.B. bereits zu viele Prozesse erzeugt) kehrt er einmal mit -1 zurück.

Init Prozess erbt den Sohn wenn der Vater vor dem Sohn terminiert und nicht wartet

Beispiel:

```
switch (pid=fork()){
    case -1:    // Fehler
    case 0:    // Sohn
    default:   // Papi
```

**fprintf**

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

**Prototyp:**

```
int fprintf(FILE *fp, const char formatstring, ... );
```

**Rückgabewert:**

Anzahl der ausgegebenen Zeichen; Fehlerfall: **EOF (= - 1)**

**Beschreibung:**

schreibt Formatstring in die mit *fp* (*Filepointer*) verbundene Datei (Formatierung zulässig)

**Beispiel:**

```
fprintf(fp, "2 * 2 = %d \n", 2*2);
/* Ausgabe von: 2 * 2 = 4 in die Datei, die mit fp verbunden ist */
```

**fputc**

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

**Prototyp:**

```
int fputc(int c, FILE *fp);
```

**Rückgabewert:**

Fehlerfall: **EOF (= - 1)**; andernfalls: ausgegebenes Zeichen

**Beschreibung:**

schreibt das Zeichen c auf die aktuelle Dateiposition (wirkungsgleich mit Makro **putc**)

**Beispiel:**

```
fputc(c, fp);
/* schreibt das Zeichen c an die aktuelle Position in die Datei */
```

**fputs**

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

**Prototyp:**

```
int fputs(const char *string, FILE *fp);
```

**Rückgabewert:**

Fehlerfall: **EOF (= - 1)**; andernfalls: von **EOF (= - 1)** verschiedener Wert

**Beschreibung:**

schreibt die Zeichenkette *string* in Datei. Das Stringende-Zeichen '\0' wird nicht in die Datei geschrieben.

**Beispiel:**

```
fputs(string, fp);
/* schreibt string ohne Stringende-Zeichen in die mit fp verbundene Datei */
```

**fscanf**

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

**Prototyp:**

```
int fscanf(FILE *fp, const char, *formatstring[,arg1,arg2,..]);
```

**Rückgabewert:**

Anzahl der tatsächlich gelesenen Datenfelder; Dateiende erreicht: **EOF**

**Beschreibung:**

liest Elemente aus der Datei, interpretiert sie unter Kontrolle des formatstrings und legt sie in den durch *arg1, arg2, ...* adressierten Speicherplätzen ab. *formatstring* wird in der bei **scanf** beschriebenen Weise gebildet *arg1, ...* ist ein Zeiger auf eine Variable, deren Typ mit der entsprechenden Typangabe im Format-String übereinstimmen muss.

**Beispiel:**

```
fscanf(fp, "%10s %ld", name, &number);
printf("name = %s number = %ld\n", name, number);
/* Ausgabe des Strings "name" und der long-Zahl "nummer" */
```

## getchar

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

**Prototyp:**

```
int getchar(void);
```

**Rückgabewert:**

eingelenes Zeichen (als Integer), Fehler oder Dateiende: **EOF (= - 1)**

**Beschreibung:**

Der Makro liest das nächste Zeichen von **stdin**

**Beispiel:**

```
c = getchar();
/* weist c das nächste Zeichen von stdin zu */
```

## getenv

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

**Prototyp:**

```
char *getenv(const char *name);
```

**Rückgabewert:**

Zeiger auf den entsprechenden Eintrag; kein Name: **NULL**

**Beschreibung:**

sucht die Tabelle der Umgebungsvariablen nach dem zu *name* gehörenden Eintrag ab

**Beispiel:**

```
ptr = getenv("PATH");
/* weist ptr den Inhalt des "PATH"-Eintrages in der Umgebung zu */
```

## gets

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

**Prototyp:**

```
char *gets(char *buffer);
```

**Rückgabewert:**

Funktion liefert ihr Argument als Rückgabe, Fehler oder Dateiende: **NULL-Zeiger**

**Beschreibung:**

liest eine Textzeile bis zum Newline-Zeichen "\n" von der **stdin** und schreibt sie in den durch *buffer* adressierten Speicherbereich. "\n" wird durch "\0" ersetzt

**Beispiel:**

```
char buffer[100];
gets(buffer);
/* liest Textzeile in buffer ein */
```

## if

**Prototyp:**

```
if (Abfrage) {Anweisung}
```

**Rückgabewert:**

**Beispiel:**

```
if (a > b)
{
    Anweisung 1
else if (a < b) {Anweisung 2}
else {Anweisung 3}
    puts("else");
}
```

## kill

```
#include <signal.h>
```

**Prototyp:**

```
int kill(pid, int sig);
```

**Beschreibung:**

pid=0: alle Prozesse der selben Prozessgruppe

Signal sig (aus signal.h) wird an Prozess mit PID pid gesendet (auch an aufrufenden Prozess möglich). Senden ist nur an Prozesse möglich, die einem selbst gehören (Ausnahme: Superuser).

ctrl-c: SIGINT

ctrl-\: SIGQUIT

## Makros

### Beschreibung:

Wenn Makro über mehrere Zeilen geht, jede Zeile mit \ abschließen  
Vorteile: eine Typdeklaration  
schneller als Funktionen

### Beispiel:

```
#define grenze 100
```

Makro: für Grenze wird im Programm 100 eingesetzt; nur 1 Zeile  
**Achtung** nach define **kein ;**  
Die Makro Definition wird wieder entfernt

```
undef grenze  
#define quadrat(a) (a)*(a)  
i = quadrat(3+5)
```

wird vom Preprozessor als (3+5) \* (3+5) interpretiert

## listen

```
#include<sys/socket.h>  
int listen(int sd, int backlog)  
    backlog: spezifiziert die Länge einer Warteschlange des passiven Socket  
S. 124
```

## open

```
#include <fcntl.h>  
Prototyp:  
int open(char *path, int oflag, mode_t mode)  
Rückgabewert:  
Fildescriptor oder bei Fehler -1  
Beschreibung:  
Stellt I/O Verbindung zwischen existierendem File und dem ausführenden Prozess her  
Schlägt fehl, wenn Prozess Rechte nicht besitzt  
oflag siehe Skript 3 Seite 130 (O_RDONLY, O_WRONLY, O_RDWR)  
mode siehe Skript 3 Seite 131 (O_CREAT, O_TRUNC, O_APPEND, O_EXCL, O_NONBLOCK, O_NDELAY,  
O_SYNC)  
Beispiel:  
open (path, O_WRONLY | O_CREAT | O_TRUNC, perms)  
/* zum schreiben öffnen, wenn noch nicht existiert anlegen, sonst Länge auf 0 */  
/* perms z.B. 0662; führende 0 wichtig -> Hexadezimal)*/
```

## pclose

```
Prototyp:  
int pclose (FILE *fp)  
Rückgabewert:  
Exitstatus von cmd oder -1 bei Fehler  
Beschreibung  
Schließt die Pipe zu cmd.  
S. 67
```

## pipe

```
Prototyp:  
int pipe (int pipefd[2])  
Rückgabewert:  
0 bei Erfolg, -1 bei Fehler  
Beschreibung:  
Eine Pipe ist ein unidirektionaler Datenkanal. Zwei FD repräsentieren die Pipe im User Prozess. Der Systemcall pipe() kreiert die Pipe, er liefert die beiden Enden als FD über sein Vektorargument an den Prozess. Dabei ist pipefd[1] das Schreibende, pipefd[0] das Lesende.  
Realisierung Skript ab S. 59
```

## popen

```
Prototyp:  
FILE *popen (char *cmd, char *mode)  
    cmd: auszuführendes Kommando  
    mode: Schreiben oder Lesen über eine Pipe mit cmd  
Rückgabewert  
Filepointer bei Erfolg oder NULL bei Fehler.  
Beschreibung:  
kreiert unidirektionale PIPE und einen Prozess, der aus der Pipe liest, oder in die Pipe schreibt  
S. 67  
Beispiel:  
popen („ls“, „r“)  
/* führt ls als eigenen Prozess aus und liest von ls/*
```

## printf

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

### Prototyp:

```
int printf(const char *formatstring, ..., [arg1, ..., argn]);
```

### Rückgabewert:

Anzahl der ausgegebenen Zeichen; im Fehlerfall: **EOF** (= - 1)

### Beschreibung:

gibt *formatstring* auf **stdout** aus. Formatelemente werden durch Werte aus der Argumentenliste ersetzt

## putc

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

### Prototyp:

```
int putc(int c, FILE *fp);
```

### Rückgabewert:

Fehlerfall: **EOF** (= - 1); andernfalls: ausgegebenes Zeichen

### Beschreibung:

Der Makro schreibt das Zeichen *c* auf die aktuelle Position der Datei

### Beispiel:

```
putc(c, fp);  
/* schreibt das Zeichen c an die aktuelle Position der Datei, die mit fp verbunden  
ist */
```

## putchar

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

### Prototyp:

```
int putchar(int c);
```

### Rückgabewert:

Fehlerfall: **EOF** (= - 1); andernfalls: ausgegebenes Zeichen

### Beschreibung:

Der Makro schreibt das Zeichen *c* auf die Standardausgabe. (**putc(c,stdout)** ist wirkungsgleich zu **putchar(c)**).

### Beispiel:

```
putchar(c);  
/* gibt das Zeichen c über stdout aus */
```

## puts

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

### Prototyp:

```
int puts(const char *string);
```

### Rückgabewert:

nicht negativer Wert; im Fehlerfall: **EOF** (= - 1)

### Beschreibung:

Schreibt den String *string* auf die Standardausgabe **stdout**, abschließend wird ein Newline-Zeichen '\n' ausgegeben.

### Beispiel:

```
puts("Ausgabe");  
/* gibt 'Ausgabe' mit abschließendem Zeilenumbruch auf stdout aus */
```

## rand

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

### Prototyp:

```
int rand(void);
```

### Rückgabewert:

Zufallszahl

### Beschreibung:

liefert eine Pseudo-Zufallszahl, die zwischen **0** und **RAND\_MAX** liegt. **RAND\_MAX** ist in **stdlib.h** definiert und ist mindestens **32767**. Die Initialisierung erfolgt mit **srand**. Im Allgemeinen ist eine **MODULO**-Division (%) sinnvoll, z.B. wenn man eine Zahl zwischen 0 und 3 möchte: `rand() % 4`.

## scanf

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

### Prototyp:

```
int scanf(const char *formatstring, ..., [arg1, ..., argn]);
```

### Rückgabewert:

Anzahl tatsächlich eingelesener und konvertierter Eingabefelder; bei Lesefehler oder Dateieinde: **EOF** (= - 1)

### Beschreibung:

liest Zeichen in formatierter Form von der Standardeingabe **stdin** ein und legt sie konvertiert auf den Argumenten *arg1* bis *argn* ab. Die Interpretation der eingelesenen Zeichen erfolgt durch den Format-String  
Mit & wird Zeiger auf folgende Variable konstruiert

## sigaction()

```
#include <signal.h>
```

### Prototyp

```
int sigaction(int sig, const struct sigaction *newact, struct sigaction *oldact);
struct sigaction newact, oldact;
```

**Die sigaction Struktur enthält:**

```
void (*sa_handler)();
void (*sa_sigaction)(int, siginfo_t *, void *);
sigset_t sa_mask;
int sa_flags;
```

**Beschreibung**

sig bezeichnet das Signal, bei dem sigaction eintritt (außer SIGKILL und SIGSTOP). newact zeigt auf eine Struktur die enthält was passieren soll wenn sig eintritt. oldact enthält die Default-Werte für sig und wird wiederhergestellt, wenn sigaction beendet wird.

**signal**

```
#include <signal.h>
```

**Prototyp:**

```
int (*signal (int sig, void (*func)(int)))(int);
```

**Rückgabewert:**

Fehlerfall : SIG\_ERR; erster Aufruf : SIG\_DFL; andernfalls: Wert von func aus dem vorhergehenden Aufruf

**Beschreibung:**

legt fest, wie das Programm auf Signale der Signalnummer sig reagiert. Möglichkeiten:

SIGINT durch Drücken von CTRL + C

SIGQUIT durch Drücken von CTRL + \

SIGUSR1

SIGUSR2

Das Argument func legt die Art der Reaktion fest. Folgende Möglichkeiten sind vorhanden:

SIG\_DFL Standardroutine für die Signalnummer wird aufgerufen; danach im allgemeinen Programmende

SIG\_IGN Signal wird ignoriert, das Programm fortgesetzt

**Beispiel:**

```
signal(SIGINT,mysig) == SIGERR) /*führt bei CTRL + C mysig aus*/
signal(SIGINT,SIG_IGN); /*Ignoriert CTRL + C*/
signal(SIGINT,SIG_DFL); /*Setzt Reaktion auf CTRL + C auf Standard zurück*/
```

**sizeof**

**Prototyp:**

```
int sizeof(char *buf)
```

**Beschreibung:**

liefert die Anzahl der Zeichen von buf

**socket**

```
#include<sys/types.h>
```

```
#include<sys/socket.h>
```

**Prototyp**

```
int socket(int domain, int type, int protocol)
```

domain: Die zu verwendende Domain (AF\_UNIX, AF\_INET; AF=address family)

type: zu verwendende Kommunikationssemantik (SOCK\_STREAM, SOCK\_DGRAM, SOCK\_RAW)

protocol: Kommunikationsprotokoll. bei 0: System wählt geeignetes Protokoll (abh. von domain und type)

**Rückgabewert:**

Deskriptor, der den erzeugten Socket referenziert und in allen weiteren darauf operierenden Funktionen benutzt wird.

**Beispiel:**

```
int sd = socket(AF_INET, SOCK_STREAM, 0);
```

Damit wird ein Stream-Socket der Internet-Domäne erzeugt, welches das voreingestellte Transportprotokoll (z.B. TCP) als das dem Socket zugrundeliegende Kommunikationsprotokoll verwendet.

**sprintf**

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

**Prototyp:**

```
int sprintf(char *string,const char *formatstring,... [arg1,...,argn]);
```

**Rückgabewert:**

Anzahl der Zeichen, die in den String kopiert wurden (ohne Stringende-Zeichen)

**Beschreibung:**

kopiert formatstring mit den Werten aus der Argumentliste formatiert in den String string (Stringende-Zeichen wird angehängt)

**Beispiel:**

```
sprintf(text,"%15s",name);
```

```
/* kopiert den 15 Zeichen langen String name in den Speicher, der durch text adressiert wird */
```

**srand**

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

**Prototyp:**

```
void srand(unsigned n);
```

**Rückgabewert:**

keiner

**Beschreibung:**

initialisiert den Zufallszahlengenerator mit **n**; anschließend erzeugt [rand](#) eine neue Folge von Zufallszahlen

**Beispiel:**

```
srand(5); /* initialisiert den Generator mit 5 */
```

## sscanf

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

**Prototyp:**

```
int sscanf(char *string,const char *formatstring,... [arg1,...,argn]);
```

**Rückgabewert:**

Anzahl der tatsächlich übertragenen Eingabefelder; Stringende-Zeichen : **EOF**

**Beschreibung:**

liest und konvertiert Daten aus dem String *string* und schreibt sie in die durch die Argumentliste bezeichneten Variablen

**Beispiel:**

```
sscanf(adresse,"%[^,],%[^,],%d",ort,straße, &hausnr);  
/* liest aus dem String adresse 1.Teilstring (bis zu einem Komma)  
in die Variable ort, 2. Teilstring (bis zu einem Komma)  
in straÙe und den Rest (eine Zahl) in hausnr */
```

## strings

**Beschreibung:**

Stringkonstante immer in doppelten Anführungszeichen

Immer durch Nullbyte (\0) abgeschlossen

**Beispiel:**

```
char *str ="ein Text";  
char s[] = „Ein weiterer Text“;  
while(*s != \0) s++;  
/* Geht durch den gesamten String s durch */  
while(*s++);  
/* etwas kürzere Möglichkeit */
```

## strcmp

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

**Prototyp:**

```
int strcmp(const char *s1,const char *s2);
```

**Rückgabewert:**

< 0 falls s1 < s2;

= 0 falls s1 = s2

> 0 falls s1 > s2

**Beschreibung:**

vergleicht die Strings *s1* und *s2* lexikographisch

**Beispiel:**

```
strcmp("HALLO","hallo");  
/* vergleicht die beiden Strings 'HALLO' und'hallo' miteinander ('hallo' ist größer  
als 'HALLO',da im ASCII die Kleinbuchstaben höherwertig sind) */
```

## struct

**Prototyp:**

```
struct pers_dat {  
    char *p_name, *p_nr;  
    short geb;  
};  
struct pers_dat pers  
/*Variable pers*/
```

**Beschreibung:**

Die Variable pers steht nun für die gesamte Struktur

**Beispiel:**

```
Initialisierung:      pers_dat person = {Otto Huber","SAI", "15}  
Zugriff:              if(person.geb == 15) ...  
Zeiger auf Struktur:  pers_dat *point;  
                      if ((point -> geb == 15) || ((*point).geb == 15))...
```

## strcat

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

**Prototyp:**

```
char *strcat(char *s1,const char *s2);
```

**Rückgabewert:**

erstes Argument

**Beschreibung:**

kettet den String *s2* an das Ende des Strings *s1* (Stringende-Zeichen von *s1* wird überschrieben). *s1* muss *s2* aufnehmen können.

strncat kettet n Zeichen von *s2* an *s1*

**Beispiel:**

```
strcat(s1,s2); /* hängt s2 an s1 an */
```

## strchr

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

**Prototyp:**

```
char *strchr(const char *s,int c);
```

**Rückgabewert:**

Zeiger auf Stelle des 1. Auftretens von *c* im String *s*; nicht vorhanden: **NULL**-Zeiger

**Beschreibung:**

sucht nach dem 1. Auftreten des Zeichens *c* im String *s*

**Beispiel:**

```
strchr(text,zeichen);
```

```
/* sucht im String text nach dem 1. Auftreten des Zeichens zeichen */
```

## strcpy

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

**Prototyp:**

```
char *strcpy(char *s1,const char *s2);
```

„Rückgabewert“ über das Argument:

*s1*

**Beschreibung:**

kopiert *s2* in den von *s1* adressierten Speicher

**Beispiel:**

```
strcpy(s1,s2);
```

```
/* kopiert s2 in den Speicher von s1 (s1 geht verloren) */
```

## strlen

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

**Prototyp:**

```
size_t strlen(const char *s);
```

**Rückgabewert:** Länge des Strings

**Beschreibung:**

liefert die Länge des Strings *s* (ohne Stringende-Zeichen). **size\_t** ist in **string.h** als **unsigned int** definiert

**Beispiel:**

```
strlen("Beispiel");
```

```
/* bestimmt die Länge des Strings 'Beispiel' */
```

## strtok

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

**Prototyp:**

```
char *strtok(char *s1,const char *s2);
```

**Rückgabewert:**

Zeiger auf das nächste Token; kein weiteres Token: **NULL**-Zeiger

**Beschreibung:**

zerlegt String *s1* in 'Token', die durch Zeichen aus *s2* voneinander getrennt sind. Beim 1. Aufruf muss *s1* angegeben werden, bei jedem weiteren dagegen 0. An jedes Token wird das Stringende-Zeichen '\0' angehängt (*s1* wird verändert!). *s2* kann bei jedem Aufruf verändert werden

**Beispiel:**

```
zeiger = strtok("Hallo, wie geht es"," ");
```

```
zeiger1 = strtok(0," ");
```

```
/* zeiger zeigt auf 'Hallo,' und zeiger1 auf 'wie' */
```

## switch

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

**Prototyp:**

```
switch(i);
```

**Beschreibung:**

*i* muß vom Typ int, short, long, char sein

Mehrfachverzweigung, dient zur Fallunterscheidung mit case

**Beispiel:**

```
switch(i)
{
    case 0: ....
            break;
    default: ...
            break;
}
```

## system

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

**Prototyp:**

```
int system(const char *s);
```

**Rückgabewert:**

NULL-Zeiger übergeben:

- Kommandointerpreter nicht verfügbar: 0

- sonst:

Wert ungleich 0

Argument kein NULL-Zeiger:

- compilerabhängig, in der Regel 0 bei korrekter Ausführung

Wert ungleich 0 bei einem Fehler

**Beschreibung:**

schickt den String *s* an den Kommandointerpreter des Betriebssystems. Die Bearbeitung erfolgt wie bei am Prompt eingegebenen Befehlen.

**Beispiel:**

```
system("CLEAR");
```

```
/* löscht über den Kommandointerpreter des Betriebssystems den Bildschirm */
```

**typedef****Prototyp:**

```
typedef int LENGTH;
```

**Beschreibung:**

Nun ist LENGTH synonym zu int

Es wird kein neuer Datentyp erzeugt, sondern nur ein anderer Name eingeführt

**Beispiel:**

```
#define PT int*
```

```
PT p1, p2;          /*p1 wird Pointer, p2 Integer*/
```

```
typedef int* PT
```

```
PT p1, p2;          /*p1 und p2 werden zu Pointern*/
```

**ungetc**

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

**Prototyp:**

```
int ungetc(int c, FILE *fp);
```

**Rückgabewert:**

Zeichen *c*; Fehlerfall : EOF (= - 1)

**Beschreibung:**

schreibt das Zeichen *c* wieder in den Dateipuffer zurück. Sollte ein EOF (= - 1)-Flag gesetzt sein, wird es gelöscht. Die nachfolgende Leseoperation beginnt mit dem Zeichen *c*. Die Datei muss zum Lesen geöffnet sein.

**Beispiel:**

```
ungetc(c, fp);
```

```
/* schreibt das Zeichen c in den Dateipuffer der mit fp verbundenen Datei */
```

**wait**

```
#include <sys/wait.h>
```

**Prototyp:**

```
int wait (int *exitstatus);
```

**Rückgabewert:**

PID des Sohnes oder -1 bei einem Fehler.

**Prototyp:**

Hat der Aufrufer von wait() keinen Kind-Prozess erzeugt, kehrt wait() mit -1 zurück.

Ansonsten blockiert wait(), bis einer der erzeugten Kinder terminiert;wait liefert dann die PID dieses Kindes.

Das Argument von wait() liefert Informationen über den terminierten Kind-Prozess. Wird kein wait eingesetzt, so wird das Kind zum Zombie terminiert.

**Beispiel:**

```
pid=wait(&stat) //pid ist SohnID, stat der Exitstatus des Sohnes
```

**waitpid**

```
#include <wait.h>
```

**Prototyp:**

```
int waitpid(int pid, int *stat, int opt);
```

**Prototyp:**

pid = -1: alle Sohnprozesse

stat= Exit-Status

opt= WNOHANG (Vater wartet nicht), ...

**while****Prototyp:**

```
while (1 > 0) {...}
```

**Beschreibung:**

Schleife, hier sogar Endlosschleife, die z.B. durch eine exit() verlassen werden kann

## write

```
#include <unistd.h>
```

### Prototyp:

```
int write(int fd, char *buf, int nbytes)
```

### Rückgabewert:

Anzahl der Bytes oder bei Fehler -1

### Beschreibung:

Schreibt die Datenmenge `nbytes` aus `buf` in den Filedescriptor `fd`

## Zeiger

### Beschreibung:

Durchgriff auf referenziertes Objekt durch Vorstellen von `*` vor die Zeigervariable

für `NULL` muß `<stdlib.h>` importiert werden

Der Wert einer Zeigervariable ist die Adresse einer Speicherfläche, deren Inhalt von dem Typ sein muß, auf den der Zeiger zeigt

### Beispiel:

```
int *p, i = 10, arr[] = {1,2,3,4,5};  
p = &i;      → printf(„%d“, *p) gibt 10 aus  
p = arr;    → printf(„%d“, *p) gibt 1 aus  
*p = 100    → printf(„%d“, *p) gibt 100 aus
```

## Anweisungen

<code>break</code>	Verlassen das case-Falles in der Switch-Anweisung
<code>continue</code>	Springt an Schleifenanfang
<code>return</code>	Beendet Funktion
<code>sleep(1)</code>	Pause in Sekunden
<code>system("clear")</code>	Befehl an Shell
<code>fflush(stdout)</code>	Puffer leeren

## Operatoren

<code>&amp;p</code>	Adressoperator
---------------------	----------------

\*p Dereferenzierungsoperator: p Zeigerwert → \*p Objekt, auf das p zeigt  
 ! logisches Komplement; Ergebnis ist eins, falls Operand 0, sonst 0  
 sizeof(T) Zahl der Bytes, die der Datentyp T belegt

### Zahlentypen:

int  
 short int  
 long int  
 unsigned int  
 unsigned short int  
 unsigned long int  
 float  
 char  
 double

### Ersatzdarstellungen für Datentyp char

\b backspace  
 \n newline  
 \r Wagenrücklauf  
 \0 Nullbyte

### Logische Ausdrücke

logisches und &&  
 logisches oder ||

### Sonstiges

- Bei Prozeduren (Funktionen ohne Resultat) sollte void als Funktionstyp angegeben werden
- „=" ist der Zuweisungsoperator
- >test 2>fehler  
lenkt stdin auf test und stderr auf fehler um
- echo \$? gibt den Exit Status eines Programms zurück
- Funktionen immer Global -> nicht geschachtelt.
- Parameterübergabe immer call-by-value
- top liefert Rechenintensive Prozesse
- 

### Programmname: name.c

Compiler: gcc -o prog name.c

### Programmaufbau:

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

```
const unsigned int konstante = 100 , abc = 5;  
char name[50]
```

```
int func(int zahl) {...}
```

```
int main () {
```

```
    int zahl = 4;
```

```
    while (1 > 0) {
```

```
        zahl++;
```

```
        printf("Gebe Zahl %d aus", zahl);
```

```
    }
```

```
    return 0;
```

```
}
```

0 Ordnungsgemäß beendet - 1 nicht normal beendet

```
puts("hallo");
```

```
printf("text %d text %d", zahl , zahl2);
```

Ausgabe eines Strings auf stdout

Ausgabe (Anzahl der ausgegebenen Zeichen)

%4.2d	Integer ( min. Ausgabelänge 4 ; Führende Nullen 2)
%6.5f	Float (min. 6 Ausgabezeichen ; 5 Nachkommastellen)
%c	Char
%5.0s	String (max. Ausgabelänge " s")
%x	Hexadezimal mit a - f
%X	Hexadezimal mit A - F
%e	Reele Zahlen in Exponentialform
l,h	Zusätzlich: l: long; h: short

send, recv  
S. 126

sendto, recvfrom  
S. 127

sendmsg, recvmsg  
S. 127

hier können noch einige Socket-Funktionen hinzugefügt werden (ab S. 128). Ich bin jetzt zu müde um das noch zu verstehen.