

C-Spezialitäten

Im industriellen Umfeld existieren viele Anwendungen, die in „reinem“ C kodiert sind und gewartet werden müssen. Deshalb sollte man die Unterschiede zu C++ kennen.

Dieser Teil verdeutlicht die wesentlichen Unterschiede zwischen C und C++ bzgl. ihrer prozeduralen Eigenschaften.

Inhalt

1. Historie.....	3
2. Parameterübergabe	4
2.1. Wertübergabe (engl.: call by value)	4
2.2. Adressübergabe (Variablenübergabe, engl.: call by reference)	8
3. Strukturen und Funktionen.....	10
4. Kernighan & Ritchi Syntax für Funktionen	11
5. Blöcke und Deklarationen	12
6. Strings und Zeichenketten	15
7. Ein- und Ausgabe	21

7.1.	Lesen von Standardeingabe und Schreiben auf Standardausgabe	21
7.2.	Lesen von Dateien und Schreiben auf Dateien	29
7.3.	Formatiertes Lesen von und Schreiben auf Variablen	35
8.	Systemaufrufe	36
8.1.	Signale	37
8.2.	Dateiverwaltung.....	44
8.3.	Katalogverwaltung.....	50
8.4.	Zeitverwaltung.....	51

1. Historie

Die Programmiersprache C++ hat ihre Wurzeln zunächst in der Sprache C, die zu Beginn der 70er von **Dennis Ritchi** zur Implementierung des damals recht neuen Betriebssystems **Unix** „erfunden“ wurde.

Die Grundidee war, eine **prozedurale Hochsprache** zur Verfügung zu haben und gleichzeitig auch **hardwarenah**, aber hardwareunabhängig entwickeln zu können.

Zusammen mit **Kernighan** wurde das C Standardwerk „*The C Programming Language*“ veröffentlicht. Darin ist der so genannte **K&R Standard** definiert.

In der Folgezeit wurde wegen der enormen Beliebtheit von K&R C in Forschung und Industrie **ANSI C** eingeführt. Dabei wurde K&R C um einige wenige Eigenschaften erweitert.

Darauf aufbauend wurde (etwa 1980) von **Bjarn Stoustrup** „*C with Classes*“ vorgeschlagen, um C um objektorientierte Eigenschaften zu erweitern. Dies führte 1984 zum ersten C++ und schließlich 1995 zum heutigen Standard *ANSI C++*.

ca.	Name	Grundlegende Idee
1970	K&R C	Hochsprache zur hardwarenahen, aber portablen Programmierung
1980	ANSI C	Standardisierung
1984	C++	Besseres C und objektorientierte Programmierung
1995	ANSI C++	Standardisierung

2. Parameterübergabe

In Programmiersprachen werden u.a. folgende Arten der Parameterübergabe unterschieden:

2.1. Wertübergabe (engl.: call by value)

Mechanismus:

- Der Wert des aktuellen Parameters wird beim Aufruf des Unterprogramms ermittelt und in den formalen Parameter kopiert.

Folge:

- Als aktuelle Parameter sind auch Ausdrücke möglich.
- Änderungen am formalen Parameter im Unterprogramm beeinflussen den Wert des aktuellen Parameter nicht.
- Soll mit call by value eine Änderung durch das aufgerufene Unterprogramm erzielt werden, so müssen Zeiger übergeben werden und das Unterprogramm muss dereferenzieren. Dabei bleibt aber der Parameterwert (Zeiger) unverändert.

Programmiersprachen:

- Ada, ALGOL 60, C, C++, Pascal

Beispiel: Integerwert übergeben

Programm

```
$ cat incl.c
#include <stdio.h>
void incl(int p) {
    p++;
    printf("  in incl: p=%d\n", p);
}
main() {
    int x;
    x=1;
    printf(" vor incl: x=%d\n", x);
    incl(x);
    printf("nach incl: x=%d\n", x);
}
$
```

Aufruf

```
$ cc incl.c -o incl
$
$ incl
  vor incl: x=1
  in incl: p=2
nach incl: x=1
$
```

Beispiel: Zeiger übergeben

Programm

```
$ cat inc2.c
#include <stdio.h>
void inc2(int *p) {
    p++;
    printf("  in inc2:  p=%d\n", p);
}
main() {
    int x;
    int *pi;
    x=1;
    pi = &x;
    printf(" vor inc2: *pi=%d\n", *pi);
    printf(" vor inc2: pi=%ld\n", pi);
    inc2(pi);
    printf("nach inc2: *pi=%d\n", *pi);
    printf("nach inc2: pi=%ld\n", pi);
}
$
```

Aufruf

```
$ cc inc2.c -o inc2
$
$ inc2
 vor inc2: *pi=1
 vor inc2: pi=-1073744488
   in inc2: p=-1073744484
nach inc2: *pi=1
nach inc2: pi=-1073744488
$
```

Beispiel: Zeiger übergeben und im Unterprogramm Dereferenzieren.

Programm	Aufruf
<pre>\$ cat inc3.c #include <stdio.h> void inc3(int *p) { (*p)++; printf(" in inc3: p=%d\n", p); printf(" in inc3: *p=%d\n", *p); } main() { int x; int *pi; x=1; pi = &x; printf(" vor inc3: *pi=%d\n", *pi); printf(" vor inc3: pi=%ld\n", pi); inc3(pi); printf("nach inc3: *pi=%d\n", *pi); printf("nach inc3: pi=%ld\n", pi); } \$</pre>	<pre>\$ cc inc3.c -o inc3 \$ \$ inc3 vor inc3: *pi=1 vor inc3: pi=-1073744488 in inc3: p=-1073744488 in inc3: *p=2 nach inc3: *pi=2 nach inc3: pi=-1073744488 \$</pre>

2.2.Adressübergabe (Variablenübergabe, engl.: call by reference)

Mechanismus:

- Die Speicheradresse des aktuellen Parameters wird ermittelt und in den formalen Parameter kopiert.

Folge:

- Ausdrücke haben keine Adressen, damit können sie nicht als Parameter übergeben werden.
- Änderungen des formalen Parameter sind beim Aufrufer sichtbar.

Programmiersprachen:

Ada, C++, Pascal

nicht in C!

Beispiel: Adressübergabe eines Integer in C++

Programm

```
$ cat inc4.c
#include <stdio.h>
void incl(int& p) {
    p++;
    printf("  in incl: p=%d\n", p);
}
main() {
    int x;
    x=1;
    printf(" vor incl: x=%d\n", x);
    incl(x);
    printf("nach incl: x=%d\n", x);
}
$
```

Aufruf

```
$ cc inc4.c -o inc4
inc4.c:2: parse error before `&'
$
$ g++ inc4.c -o inc4
$
$ inc4
 vor incl: x=1
  in incl: p=2
nach incl: x=2
$
```

3. Strukturen und Funktionen

In C existieren nur zwei Operationen auf Strukturen:

- mit & kann die Adresse einer Struktur ermittelt werden;
- auf eine Komponente einer Struktur kann zugegriffen werden.

Daraus folgt:

- Strukturen können **nicht** als Ganzes manipuliert werden, d.h. zugewiesen bzw. kopiert werden;
- Strukturen können **nicht** als Parameter an Funktionen übergeben werden;
- Eine Funktion kann eine Struktur **nicht** als Resultat-Typ besitzen.

Einige C-Implementierungen unterliegen diesen Einschränkungen nicht (Wie ist es in Ihrer Umgebung?).

Aber durch **Zeiger auf Strukturen** kann man die o.a. Einschränkungen umgehen.

4. Kernighan & Ritchi Syntax für Funktionen

Im K&R C haben Funktionsdefinitionen mit ihren formalen Parameter eine andere Syntax:

C++

K&R C

```
Typ Name (type1 Parameter1, ...)  
{  
    Anweisungen  
}
```

```
Typ Name (Parameter1, ...)  
type1 Parameter1, ...  
{  
    Anweisungen  
}
```

Beispiel:

```
#include <stdio.h>  
double add(x,y)  
double x;  
double y;  
{  
    return x+y;  
}  
int main()  
{  
    printf("%lf\n", add(3.1, 1));  
}
```

```
#include <stdio.h>  
double add(double x, double y)  
{  
    return x+y;  
}  
int main()  
{  
    printf("%lf\n", add(3.1, 1));  
}
```

5. Blöcke und Deklarationen

In C++ kann man Variable so spät als möglich in einem Block deklarieren. Dadurch sind Anweisungen und Deklarationen in beliebiger Reihenfolge möglich, natürlich muss eine Variable vor ihrem ersten Gebrauch deklariert worden sein.

In C hat ein Block einen festen syntaktischen Aufbau: zuerst kommen Deklarationen, dann nur noch Anweisungen – ein Mischen ist also **nicht** möglich.

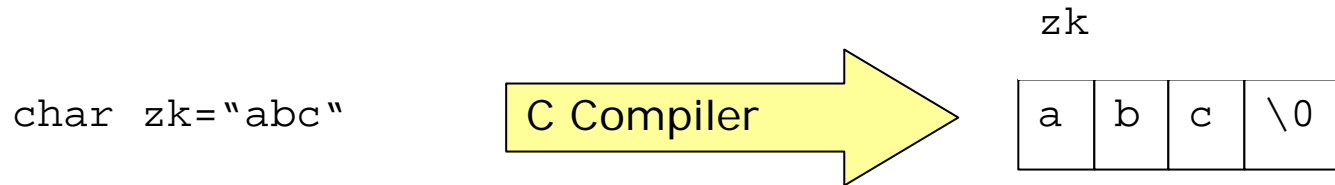
In C++ kann man Schleifenvariablen innerhalb eines for-Kopfes deklarieren, in C ist dies nicht möglich.

C++	C
<pre>\$ cat block02.cpp #include <stdio.h> double add(double x, double y) { return x+y; } int main() { double a=3.1; double b=1.0; printf("%lf\n", add(a,b)); double c=18; printf("%lf\n", add(a,c)); } \$</pre>	<pre>\$ cat block02.c #include <stdio.h> double add(double x, double y) { return x+y; } int main() { double a=3.1; double b=1.0; printf("%lf\n", add(a,b)); double c=18; printf("%lf\n", add(a,c)); } \$ gcc block02.c block02.c: In function `main': block02.c:13: parse error before `double' block02.c:14: `c' undeclared (first use in this function) \$</pre>

C++	C
<pre>\$ cat for.cpp #include <stdio.h> int main() { for (int i=1; i<=3; i++) printf("%d\n", i); } \$</pre>	<pre>\$ cat for.c #include <stdio.h> int main() { for (int i=1; i<=3; i++) printf("%d\n", i); } \$ gcc for.c for.c: In function `main': for.c:4: parse error before `int' for.c:4: `i' undeclared (first use in this function) for.c:4: for each function it appears in.) for.c:4: parse error before `)'</pre>

6. Strings und Zeichenketten

In C werden Strings als Zeichenvektoren abgebildet. Ein String endet dabei immer mit dem **Stringabschlusszeichen** `,\0'`. Dieser Abschluss wird bei Stringkonstanten durch den C-Compiler automatisch durchgeführt.



Diese Konvention wird in vielen Funktionen der Standardbibliothek verwendet. Zu beachten ist, dass man diese Konvention beim Programmieren eigener Funktion einhalten muss.

In C kann man Zeichenketten nicht direkt zuweisen, man benötigt eine Funktion dafür. Entweder man schreibt selbst eine, besser man verwendet die Funktionen aus der Standardbibliothek.

Beispiel (Kopieren von Zeichenketten):

```
$ cat copy.c
#include <stdio.h>
#include <string.h>

void copy(char s[], char d[]) {           // s nach d kopieren
    int i=0;
    while ((d[i] = s[i]) != '\0')
        i++;
}

int main() {
    char zk1[]="abc";
    char zk2[80];
    char zk3[80];

    copy(zk1, zk2);
    printf("%s\n", zk2);

    strcpy(zk3, zk1);                     // strcpy(ZIEL, QUELLE) aus Bibliothek
    printf("%s\n", zk3);
}
```

Auch der Vergleich von Zeichenketten ist **nicht direkt**, sondern nur über (Bibliotheks)-Funktionen möglich:

```
$ cat cmp.c
#include <stdio.h>
#include <string.h>
int main()
{
    char zk1[]="abcd";
    char zk2[]="xyz";

    if (strcmp(zk1, zk2) == 0)
        printf("%s == %s\n", zk1, zk2);
    else if (strcmp(zk1, zk2) < 0)
        printf("%s < %s\n", zk1, zk2);
    else printf("%s > %s\n", zk1, zk2);
}
```

Zur Beschreibung der Bibliotheksfunktion kann man das man-Kommando verwenden (Unix).

```
$ man strcmp
```

```
STRCMP(3)
```

```
Bibliotheksfunktionen
```

```
STRCMP(3)
```

```
BEZEICHNUNG
```

```
strcmp, strncmp - vergleicht zwei Strings miteinander
```

```
ÜBERSICHT
```

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

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

```
int strncmp(const char *s1, const char *s2, size_t n);
```

```
BESCHREIBUNG
```

Die Funktion strcmp() vergleicht die beiden Strings s1 und s2 miteinander. Sie gibt eine ganze Zahl kleiner, gleich oder größer als Null zurück, wenn s1 gefunden wurde und kleiner, gleich oder größer als s2 ist.

Die Funktion strncmp() funktioniert genauso mit der Ausnahme, daß nur die ersten n Buchstaben von s1 verglichen werden.

```
RÜCKGABEWERT
```

Die Funktionen strcmp() und strncmp() geben eine ganze Zahl zurück, die kleiner, gleich oder größer als Null ist, wenn s1 gefunden wurde und kleiner, gleich oder größer als s2 ist (bzw die ersten n Buchstaben von s1).

```
KONFORM ZU
```

```
SVID 3, POSIX, BSD 4.3, ISO 9899
```

```
SIEHE AUCH
```

```
bcmp(3), memcmp(3), strcasecmp(3), casecmp(3), strcoll(3).
```

```
$
```

```
$ man strcpy
```

BEZEICHNUNG

strcpy, strncpy - kopiert eine Zeichenkette

ÜBERSICHT

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

```
char *strcpy(char *dest, const char *src);
```

```
char *strncpy(char *dest, const char *src, size_t n);
```

BESCHREIBUNG

Die Funktion `strcpy()` kopiert die Zeichenkette, auf die der Zeiger `src` zeigt, **inklusive** des Endezeichens `'\0'` an die Stelle, auf die `dest` zeigt. Die Zeichenketten dürfen sich nicht überlappen und **dest muß groß genug sein**.

Die Funktion `strncpy()` tut dasselbe mit dem Unterschied, daß nur die ersten `n` Byte von `src` kopiert werden. Ist kein `'\0'` innerhalb der ersten `n` Bytes, so wird das Ergebnis nicht durch `'\0'` abgeschlossen.

Ist die Länge von `src` kleiner als `n` Bytes, so wird **dest mit Nullen aufgefüllt**.

RÜCKGABEWERT

Die Funktionen `strcpy()` und `strncpy()` geben einen Zeiger auf `dest` zurück.

KONFORM ZU

SVID 3, POSIX, BSD 4.3, ISO 9899

SIEHE AUCH

`bcopy(3)`, `memccpy(3)`, `memcpy(3)`, `memmove(3)`.

\$

Hörsaalübung:

Realisieren Sie eine Funktion, die durch folgende Spezifikation definiert ist inklusive Testumgebung.

BEZEICHNUNG

`strLen` - berechnet die Länge einer Zeichenkette

ÜBERSICHT

```
long strLen(const char *s);
```

BESCHREIBUNG

Die Funktion `strLen()` berechnet die Länge der Zeichenkette `s` und berücksichtigt dabei nicht das Zeichen `'\0'`.

RÜCKGABEWERT

Die Funktion `strLen()` liefert die Anzahl der Buchstaben der Zeichenketten `s` zurück.

7. Ein- und Ausgabe

sscanf, fscanf, Scanf, printf, File IO

Die Ein- und Ausgabe, bzw. die dazu nötigen Funktionen sind nicht Bestandteil der Sprache C selbst. Jedoch existieren in allen C-Implementierungen Bibliotheken mit speziellen E/A-Funktionen.

7.1. Lesen von Standardeingabe und Schreiben auf Standardausgabe

Zum unformatierten Lesen und Schreiben existieren folgende Funktionen:

```
NAME
    getc, getchar, gets - Eingabe von Zeichen          und Strings
ÜBERSICHT
    #include <stdio.h>

    int getc(FILE *stream);
    int getchar(void);
    char *gets(char *s);
```

BESCHREIBUNG

`getc()` ist äquivalent zu `fgetc()`, außer daß es als ein Makro implementiert sein darf, der stream mehr als einmal ausgewertet.

`getchar()` ist äquivalent zu `getc(stdin)`.

`gets()` liest eine Zeile von `stdin` in den Puffer, auf den `s` zeigt, bis entweder ein abschließender Zeilenvorschub oder EOF auftritt, welche durch `'\0'` ersetzt werden. Es wird keine Prüfung auf Pufferüberlauf durchgeführt (siehe BUGS unten).

RÜCKGABEWERTE

`getc()` and `getchar()` geben das gelesene Zeichen als ein `unsigned char` gecastet in einem `int` zurück, oder EOF bei Dateiende oder Fehler.

`gets()` gibt `s` zurück bei Erfolg, und NULL im Fehlerfall oder wenn Dateiende auftritt ohne daß Zeichen gelesen wurden.

KONFORM ZU

ANSI - C, POSIX.1

BUGS

Da es unmöglich ist zu sagen, wie viele Zeichen `gets()` lesen wird, ohne die Daten vorher zu kennen und da `gets()` fortfährt und Daten über das Ende des Puffers hinaus speichert, ist es sehr gefährlich, diese Funktion zu

benutzen. Sie wurde benutzt um in Rechner einzubrechen.

Es ist nicht ratsam, Aufrufe von Funktionen der Bibliothek `stdio` mit low-level-Aufrufen von `read()` für den Dateideskriptor zu mischen, der mit demselben Eingabestream verbunden ist; die Ergebnisse sind undefiniert und sehr wahrscheinlich nicht erwünscht.

SIEHE AUCH

`read(2)`, `write(2)`, `fopen(3)`, `fread(3)`, `scanf(3)`, `puts(3)`,
`fseek(3)`, `ferror(3)`.

BEZEICHNUNG

putc, putchar, puts - Ausgabe von Zeichen und Zeichenketten

ÜBERSICHT

```
#include <stdio.h>
int putc(int c, FILE *stream);
int putchar(int c);
int puts(char *s);
```

BESCHREIBUNG

putc() entspricht fputc() außer, daß es als ein Makro implementiert den stream mehr als einmal auswertet.

putchar(c) entspricht putc(c, stdout).

puts() schreibt die Zeichenkette s und einen nachfolgenden Zeilenumbruch in die Standardausgabe stdout.

RÜCKGABEWERTE

putc() und putchar() geben das als ein unsigned char geschriebene und in ein int umgesetzte Zeichen zurück, oder EOF im Fehlerfall.

puts() geben bei Erfolg eine nichtnegative Zahl zurück, oder EOF im Fehlerfall.

KONFORM ZU

ANSI - C, POSIX.1

BUGS

Es ist nicht ratsam, Aufrufe von Ausgabefunktionen der Bibliothek stdio mit lowlevel-Aufrufen von write() zu ver-

mischen, wenn der Dateideskriptor denselben Ausgabekanal bezeichnet. Die Ergebnisse sind undefiniert und sehr wahrscheinlich nicht die gewünschten.

SIEHE AUCH

write(2), fopen(3), fwrite(3), scanf(3), gets(3),
fseek(3), error(3).

Beispiel (simpleCat):

```
$ cat simpleCat.c
#include <stdio.h>
main() {
    int c;
    while ((c=getchar()) != EOF)
        putchar(c);
    putchar( '\n' );
}
$
```



zeichenweise

Beispiel (simpleCat):

```
$ cat simpleCat02.c
```

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

```
main() {
```

```
    char line[80];
```

```
    while (gets(line) != NULL)
```

```
        puts(line);
```

```
    putchar( '\n' );
```

```
}
```



zeilenweise

Zur formatierten Ein- und Ausgabe existieren die scanf - und printf -Funktionen.

```
printf(Kontrollstring, Argument1, Argument2, ...)
```

Unter der Kontrolle der Zeichenkette `Kontrollstring` werden die Argumente umgewandelt und auf Standardausgabe ausgegeben. Im `Kontrollstring` sind gewöhnliche Zeichen – sie werden einfach ausgegeben – und Formatelemente, die die Ausgabe des korrespondierenden Arguments bestimmen.

Beispiel (ASCII Tabelle):

```
$ cat ascii.c
#include <stdio.h>
main() {
    int i;
    printf("ASCII Tabelle\n");
    for (i=32; i<=127; i++)
        printf("%d\t%c\n", i, i);
    printf("\n");
}
```

```
ASCII Tabelle
32
33      !
34      "
35      #
36      $
37      %
38      &
...
$
```

`scanf(Kontrollstring, Argument1, Argument2, ...)`

`scanf` liest Zeichen von der Standardeingabe und wandelt sie gemäß des Kontrollstrings um und speichert sie an den Adressen, die durch die Argumente (Zeiger!!!) spezifiziert sind.

Beispiel (ASCII Tabelle):

```
$ cat ascii02.c
#include <stdio.h>
main() {
    int i;
    int von, bis;
    printf("Intervall von bis angeben: ");
    i = scanf("%d %d", &von, &bis);
    if (i != 2) {
        printf("Intervall nicht korrekt angegeben!\n");
        exit(1);
    }

    printf("ASCII Tabelle\n");
    for (i=von; i<=bis; i++)
        printf("%d\t%c\n", i, i);
    printf("\n");
}
```

```
$ ascii02
Intervall von bis angeben: 63 70
ASCII Tabelle
63      ?
64      @
65      A
66      B
67      C
68      D
69      E
70      F

$
```

Die Manual Pages bzw. die Online Hilfe verraten die Formatumwandlungen.

7.2. Lesen von Dateien und Schreiben auf Dateien

Zum Öffnen von Dateien kann man in C die Funktion `fopen()` verwenden.

NAME

fopen, fdopen, freopen - Funktionen zum Öffnen von Streams

ÜBERSICHT

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

```
FILE *fopen( char *path, char *mode);
```

```
FILE *freopen( char *path, char *mode, FILE *stream);
```

BESCHREIBUNG

Die Funktion fopen öffnet die Datei, dessen Name der String ist, auf den path zeigt, und verbindet einen Stream damit.

Das Argument mode zeigt auf einen String, der mit einer der folgenden Sequenzen beginnt (Zusätzliche Zeichen dürfen diesen Sequenzen folgen.):

r Öffne die Textdatei zum Lesen. Der Stream wird auf den Dateianfang positioniert.

r+ Öffne die Textdatei zum Lesen und Schreiben. Der Stream wird auf den Dateianfang positioniert.

w Verkürze die Datei auf die Länge Null oder erzeuge eine Textdatei zum Schreiben. Der Stream wird auf den Dateianfang positioniert.

w+ Öffne die Datei zum Lesen und Schreiben. Die Datei wird erzeugt, wenn sie nicht existiert, ansonsten abgeschnitten. Der Stream wird auf den Dateianfang positioniert.

a Öffne die Datei zum Schreiben. Die Datei wird

erzeugt, wenn sie nicht existiert. Der Stream wird auf das Dateiende positioniert.

a+ Öffne zum Lesen und Schreiben. Die Datei wird erzeugt, wenn sie nicht existiert. Der Stream wird auf das Dateiende positioniert.

Der String mode kann auch das Zeichen ``b'' enthalten, entweder als ein drittes Zeichen oder als ein Zeichen in einem der oben beschriebenen Zwei-Zeichen-Strings. Dies ist ausschließlich Aus Kompatibilitätsgründen zu ANSI C3.159-1989 (``ANSI C'') und hat keinen Effekt; das ``b'' wird ignoriert. Linux verhält sich evtl. nicht so.

Jede erzeugte Datei hat den Modus S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH (0666), modifiziert durch den umask-Werk des Prozesses (siehe umask(2)).

Lese- und Schreibzugriffe dürfen in Schreib-/lese - Streams in jeder Reihenfolge gemischt verwendet werden und benötigen kein zwischenzeitliches seek wie in früheren Versionen von stdio. Dieses Verhalten ist nicht portabel mit anderen Systemen, und muß unter Linux nicht funktionieren (irgendjemand sollte das testen und diese Manpage berichtigen); ANSI C verlangt, daß eine Dateizeiger-Positionierfunktion zwischen Aus- und Eingabe aufgerufen wird, solange nicht eine Eingabeoperation ein Dateiende vorfindet.

Die Funktion `freopen` öffnet eine Datei, deren Name der String ist, auf den `path` zeigt, und verbindet den Stream, auf den `stream` zeigt, damit. Der originale Stream (wenn er existiert) wird geschlossen. Das Argument `mode` wird genauso wie in der Funktion `fopen` benutzt. Der primäre Nutzen der Funktion `freopen` ist es, die Datei zu ändern, die mit einem standard Text-Stream (`stderr`, `stdin`, oder `stdout`) verbunden ist.

RÜCKGABEWERT

Bei erfolgreicher Beendigung geben `fopen` und `freopen` einen Dateideszeiger `FILE` zurück. Anderenfalls wird `NULL` zurückgegeben und die globale Variable `errno` gesetzt um den Fehler anzuzeigen.

Eine geöffnete Datei kann dann durch die `fputc`, `fgetc`, `getc`, `fscanf` und `fprintf` Funktionen bearbeitet werden. Diese Funktionen arbeiten wie `putc`, `getc`, `scanf` und `printf`, man gibt einfach einen Zeiger auf die zu bearbeitende Datei mit.

Beispiel (simpleCat)

```
$ cat simpleCat03.c
#include <stdio.h>
int main(int argc, char *argv[]) {
    FILE *fp; // file pointer
    char line[80];

    if (argc != 2) { // Verwendung prüfen
        fprintf(stdout, "Verwendung: %s Dateiname\n", argv[0]);
        exit(1);
    }
    if ((fp=fopen(argv[1], "r")) == NULL) { // Datei öffnen
        fprintf(stdout, "Datei %s kann nicht geöffnet werden\n", argv[1]);
        exit(1);
    }

    while (fgets(line, 80, fp) != NULL) // zeilenweise von Datei lesen
        fputs(line, stdout); // auf stdout schreiben
    putchar('\n');
}
```

Die folgenden Funktionen sind die am meisten gebräuchlichsten:

```
#include <stdio.h>

int fputc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
int puts(char *s);

int fgetc(FILE *stream);
char *fgets(char *s, int size, FILE *stream);
int getc(FILE *stream);
int getchar(void);
char *gets(char *s);

int printf( const char *format, ...);
int fprintf( FILE *stream, const char *format, ...);
int sprintf( char *str, const char *format, ...);

int scanf( const char *format, ...);
int fscanf( FILE *stream, const char *format, ...);
int sscanf( const char *str, const char *format, ...);
```

7.3. Formatiertes Lesen von und Schreiben auf Variablen

Die o.a. Funktionen `sscanf` und `sprintf` verwenden anstelle einer Datei eine Zeichenkettenvariable, um von ihr zu lesen oder in sie zu schreiben.

Beispiel (`sscanf`):

```
$ cat sscanf.c
#include <stdio.h>
main() {
    char eingabe[20];
    char ausgabe[80];
    printf("Eingabe: ");
    gets(eingabe);
    sprintf(ausgabe, "Die Eingabe war: '%s'.\n", eingabe);
    puts(ausgabe);
}
$
```

```
$ sscanf
Eingabe: 111
Die Eingabe war: '111'.

$
```

8. Systemaufrufe

Systemaufrufe bilden die Schnittstelle zur Hardware, auf dem das Betriebssystem abläuft. Deshalb sind große Teile eines Systemaufrufs in Assembler programmiert. Um sie für Programmierer nutzbar zu machen, wird oft eine **C-Bibliothek** bereitgestellt.

Ein einfacher Systemaufruf zum Lesen einer Datei ist „**read**“. Mit drei Parametern von „read“ wird beschrieben, welche Datei gelesen werden soll, wohin die Leseoperation das Ergebnis ablegen soll und wie viele Bytes aus der Datei gelesen werden sollen. Der Aufruf in einem C-Programm hat folgendes Aussehen:

```
count = read(file, buffer, nbytes);
```

Durch den Systemaufruf werden wird die durch „file“ angegebene Datei gelesen und „count“ Bytes in die Variable „buffer“ gespeichert. Normalerweise ist „count“=“nbytes“, aber wenn das Dateiende erreicht wird, sind u.U. weniger als „nbytes“ Bytes zum Lesen da. Wenn der Systemaufruf nicht ausgeführt werden kann (Plattenfehler oder Datei nicht lesbar), wird „count“ auf -1 gesetzt und die Fehlernummer wird in einer globalen Variablen „errno“ abgelegt.

Ein vollständiges C-Programm (simpleCat.c), das eine Datei liest und auf dem Bildschirm ausgibt, ist nachfolgend gezeigt:

```
#define BUFSIZE 512
main()
{
    char buf[BUFSIZE];
    int n;
    while ((n = read(0, buf, BUFSIZE)) > 0)
        write(1, buf, n);
}
```

Im Folgenden werden nun die wichtigsten Systemaufrufe kurz vorgestellt. Dabei werden sie gemäß folgender Gruppen diskutiert:

- Signale
- Dateiverwaltung
- Katalog- und Dateisystemverwaltung
- Zeitverwaltung

8.1. Signale

Signale sind das Äquivalent im Bereich Software zu Interrupts im Bereich Hardware. Programme, die als Prozess innerhalb des Betriebssystems ablaufen, müssen unterbrechbar sein, damit z.B. ungewollte Aktionen nicht ausgeführt werden. So kann die Ausgabe (durch das Kommando `cat`) einer großen Datei mit „CTR-C“ abgebrochen werden. Durch „CTR-C“ wird dem Ausgabeprozess ein Signal gesendet. Als Reaktion beendet der Prozess die Ausgabe.

Wenn ein **Signal** zu einem Prozess gesendet wird und der Prozess das Signal nicht annimmt, dann wird der Prozess vom Betriebssystem **automatisch entfernt**. Um sich vor diesem Automatismus schützen zu können, kann sich ein Prozess durch den Systemaufruf „signal“ auf das Eintreffen von Signalen vorbereiten. Dazu muss er eine **Signalbehandlungs-Routine** bereitstellen.

Ein Prozess, der eine Signalbehandlungs-Routine bereitgestellt hat, wird bei Eintreffen eines Signals angehalten, der Prozesszustand auf des Stack geschrieben und die Signalaroutine wird aufgerufen. Diese Routine darf selbst beliebige Systemaufrufe veranlassen. Ist sie beendet, wird der Prozess da weiter ablaufen, wo er vorher unterbrochen wurde (der Zustand wird vom Stack restauriert).

In einem Betriebssystem gibt es mehrere Signalarten. Die meisten Signale werden durch Ereignisse, die von der Hardware ausgelöst werden erzeugt. Die nachfolgende Tabelle listet die wichtigsten Signalarten:

Nummer	Bezeichnung	Bedeutung
1	SIGHUP	Hang up, Modemunterbrechung
2	SIGINT	DEL Taste
3	SIGQUIT	Quit Signal von Tastatur
4	SIGILL	Nicht erlaubte Instruktion
5	SIGTRAP	Unterbrechung für Testzwecke
8	SIGFPE	Gleitkommaüberlauf
9	SIGKILL	Abbruch
10	SIBBUS	Busfehler
11	SIGSEGV	Segmentfehler

12	SIGSYS	Ungültige Argumente bei Systemaufruf
13	SIGPIPE	Ausgabe auf Pipe ohne Leser
14	SIGALARM	Alarm
15	SIGTERM	Softwareerzeugtes Endesignal
16	frei	

Die Signalbehandlungs-Routine sollte als erstes selbst wieder den Mechanismus zur Signalbehandlung aufrufen, damit ein erneut eintretendes Signal ebenfalls abgefangen wird.

Das folgende Beispiel zeigt, wie in C eine Signalbehandlungs-Routine realisiert wird. Dabei wird das Programm (eine Unendlichschleife) auf das Eintreffen vom Signal „SIGINT“, ausgelöst durch „CTR-C“ reagieren, indem es einen Text auf die Ausgabe schreibt.

```

$ cat signal.c
#include <stdio.h>
#include <signal.h>
void handler()
{
    signal(SIGINT, handler);
    printf("handler\n");
    return;
}

void main()
{
    signal(SIGINT, handler); /* CTR-C handled */
    while (1) {
        printf("main\n");
        sleep(2);
    }
}
$

```

Anstelle eines selbst programmierten Handlers, kann man vordefinierte Handler verwenden. Sie werden durch die Konstante `SIG_IGN` (ignoriere Signal) und `SIG_DFL` (reagiere per Default Aktion) beim Systemaufruf „signal“ verwendet.

In Unix gibt es das Kommando „**kill**“ zum Beenden von Prozessen, die im Hintergrund laufen. Wenn ein Programm so geschrieben ist, dass es alle Signale ignoriert, könnte es nie abgebro-

chen werden. Deshalb gibt es das Signal „SIGKILL“. Dieses Signal kann **nicht** per Signalhandler abgefangen werden.

```
$ ps
2864 pts/1      00:00:18 bash
4423 pts/1      00:00:01 signal
4525 pts/1      00:00:00 ps
$
$ kill -9 4423
killed signal
$
```

Im Bereich Echtzeitanwendungen muss ein Betriebssystem in der Lage sein, Prozesse nach einer gewissen Zeit zu informieren, dass bestimmte Dinge zu erledigen sind. In Kernkraftwerken muss die Temperatur des Reaktors regelmäßig überprüft werden. Deshalb wird dem Prozess, der die Überwachung bewerkstelligt regelmäßig ein Signal gesendet, wodurch er die Temperatur prüft. Der Systemaufruf „**alarm**“ hat einen Parameter, der die Anzahl Sekunden angibt, nach denen das Signal „SIGALARM“ erzeugt werden soll. Im Umfeld der Netzprogrammierung wird der Systemaufruf z.B. verwendet, um das ping Kommando zu realisieren. Dabei wird jede Sekunde ein Datenpaket zu einem Rechner gesendet und gewartet, ob es zurück gesendet wird.

Wenn Programme ablaufen, bei denen eine gewisse Zeit gewartet werden soll, kann dies programmtechnisch realisiert werden, indem man eine Schleife verwendet, die stets nichts tut, als testet, ob die Zeit schon um ist. Diese Lösung ist CPU intensiv. Besser wäre es, man könnte per Systemaufruf sagen, dass eine gewisse Zeit pausiert werden soll. Dazu ist der System-

aufruf „pause“ verfügbar. Der Systemaufruf „**pause**“ veranlasst den aufrufenden Prozess zu warten, bis er das entsprechende Signal zum Weitermachen erhält.

Das folgende Programm „timer.c“ verwendet den Systemaufruf „alarm“, um einen Timer zu setzen. Wenn innerhalb von 5 Sekunden keine Benutzereingabe erfolgt, wird das Programm beendet.

```
$ cat timer.c
#include <stdio.h>
#include <signal.h>
void handler()
{
    printf("By\n");
    exit(0);
}

main()
{
    char str[80];
    signal(SIGALRM, handler);

    while (1) {
        alarm(5);          /* start timer */
        printf("> ");
        gets(str);
        printf("%s\n", str);
    }
}
$
```

Ein Aufruf von „timer“ bewirkt etwa:

```
$ timer
> 12345
12345          5 Sekunden warten
> By
$
```

8.2.Dateiverwaltung

Das nachfolgende Beispielprogramm „simpeTouch.c“ demonstriert den Systemaufruf „creat“. Das Programm legt die als Aufrufparameter anzugebende Datei mit den Zugriffsrechten „0640“ an.

```
#include <stdio.h>
main(int argc, char **argv)
{
    int fd;
    if (argc != 2) {
        fprintf(stderr, "usage %s file\n", argv[0]);
        exit(1);
    }
    if ((fd=creat(argv[1],0640)) < 0) {
        fprintf(stderr, "create error\n");
        exit(2);
    }
}
```

Bevor eine Datei bearbeitet werden kann, muss sie geöffnet werden. Dazu existiert der Systemaufruf „open“, der neben dem Namen noch die Art des Zugriffs (Lesen, Schreiben oder beides) benötigt.

Das folgende Programm („cat.c“) liest die als Parameter anzugebende Datei und schreibt den Inhalt auf die Standardausgabe.

```

#include <stdio.h>
#include <fcntl.h>
#define BUFSIZE 512
main(int argc, char **argv)
{
    int fd;
    int n;
    char buf[BUFSIZE];

    /* check usage */
    if (argc != 2) {
        fprintf(stderr, "usage %s file\n", argv[0]);
        exit(1);
    }
    /* open file */
    if ((fd=open(argv[1], O_RDWR)) < 0) {
        fprintf(stderr, "open error\n");
        exit(2);
    }
    /* read and write file */
    while ((n = read(fd, buf, BUFSIZE)) > 0)
        write(1, buf, n);
}

```

Der **wahlfreie Zugriff** auf Dateien wird durch den Systemaufruf „lseek“ realisiert. lseek hat drei Parameter:

1. einen Filedescriptor, der die zu bearbeitende Datei definiert,
2. den Offset , der die Position des Lese/Schreibkopfes in Byte relativ zum Ausgangspunkt definiert und
3. den Ausgangspunkt (SEEK_SET=Dateianfang, SEEK_END=Dateiende, SEEK_CUR=aktuelle Position) für die Positionierung des Lese/Schreibkopfes

Damit kann man ein Programm („revCat.c“), das eine Datei von hinten nach vorne liest einfach schreiben:

```

#include <stdio.h>
#include <fcntl.h>
main(int argc, char **argv)
{
    int fd;
    int pos;
    char buf[1];
    if (argc != 2) { /* check usage */
        fprintf(stderr, "usage %s file\n", argv[0]);
        exit(1);
    }
    if ((fd=open(argv[1], O_RDWR)) < 0) { /* open file */
        fprintf(stderr, "open error\n");
        exit(2);
    }
    /* set position for reading at end of file */
    if ((pos=lseek(fd, -1, SEEK_END)) == -1) {
        fprintf(stderr, "lseek error\n");
        exit(1);
    }
    while (pos>=0) { /* read and write file */
        read(fd, buf, 1);
        write(1, buf, 1);
        lseek(fd, --pos, SEEK_SET);
    }
    printf("\n");
}

```

Im folgenden Programm „hole.c“ wird der Systemaufruf „lseek“ verwendet, um ein Loch in einer Datei zu erzeugen.

```
#include <stdio.h>
int main() {
    int fd;
    char buf1[] = "abcdefghijklmnop";
    char buf2[] = "ABCDEFGHIJKLMNOP";

    if ((fd=creat("file.hole",0640)) < 0) {
        fprintf(stderr, "create error\n");
        exit(1);
    }
    if ((write(fd, buf1, 16)) != 16) { /* offset now 16 */
        fprintf(stderr, "buf1 write error\n");
        exit(2);
    }
    if ((lseek(fd, 32, SEEK_SET)) == -1) { /* offset now 32 */
        fprintf(stderr, "lseek error\n");
        exit(3);
    }
    if ((write(fd, buf2, 16)) != 16) { /* offset now 48 */
        fprintf(stderr, "buf2 write error\n");
        exit(4);
    }
    exit(0);
}
```

Ein Aufruf von „hole“ bewirkt etwa:

```
$ ls -l fil*
-rw-r----- 1 as users 48 Nov 1 19:11 file.hole
$ od -c file.hole
0000000  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p
0000020  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 Loch
0000040  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P
0000060
$
```

8.3. Katalogverwaltung

Hier werden einige Systemaufrufe vorgestellt, die auf Verzeichnisse wirken.

Um vom aktuellen Verzeichnis in ein anderes Verzeichnis zu wechseln, existiert der Systemaufruf „**chdir**“. Nachdem das aktuelle Verzeichnis mit dem Systemaufruf

```
chdir( „/usr/schuetzte/tmp“ )
```

geändert wurde, werden die folgenden Aufrufe (create, open), bei denen kein voller Pfadname angegeben ist, immer im tmp-Verzeichnis Dateien anlegen, bzw. öffnen.

Ein Systemaufruf, mit dem das Root-Verzeichnis bestimmt werden kann ist „**chroot**“. Er wird verwendet, um etwa auf Web Servern virtuelle Root-Verzeichnisse für unterschiedliche Benutzergruppen definieren zu können.

8.4. Zeitverwaltung

In Betriebssystemen muss die Verwaltung von Datum und Uhrzeit durch Systemaufrufe unterstützt werden. Neben dem Abfragen der Systemzeit muss es Aufrufe für das Setzen der Zeit und für die Umstellung von Sommerzeit zu Winterzeit und umgekehrt geben.

In Unix ist Datum und Uhrzeit als Anzahl Sekunden, die seit dem „Unix Urknall“ (1.1.1970 00:00:00) vergangen sind, abgelegt. Alle Datumsangaben, werden so gespeichert und erst in der Anzeige in lesbare Form gebracht. Dazu existiert der Systemaufruf „time“, der die Sekundenanzahl liefert und Routinen, um diese Intergerzahl in ein lesbares Format zu konvertieren.

Das folgende C-Programm („now.c“) liefert das aktuelle Datum und die Uhrzeit.

```
#include <stdio.h>
#include <time.h>
main()
{
    time_t now;
    now = time(NULL); /* now as no. secs since ZERO */
    printf("%s\n", ctime(&now)); /* convert into readable form */
    exit(0);
}
```

```
$ now
```

```
Thu Nov 15 16
```