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 30
7.3. Formatiertes Lesen von und Schreiben auf Variablen 36
8. Systemaufrufe 37
8.1. Signale 39
8.2. Dateiverwaltung 45
8.3. Katalogverwaltung 51
8.4. Zeitverwaltung 52
8.5. Prozessverwaltung 56
9. Terminaleigenschaften verändern 70

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	Aufruf

`$ cat inc1.c`
`#include <stdio.h>`
`void inc1(int p) {`
`        p++;`
`        printf("  in inc1: p=%d\n", p);`
`}`
`main() {`
`        int x;`
`        x=1;`
`        printf(" vor inc1: x=%d\n", x);`
`        inc1(x);`
`        printf("nach inc1: x=%d\n", x);`
`}`
`$`
	`$ cc inc1.c -o inc1`
	`$`
	`$ inc1`
	` vor inc1: x=1`
	`  in inc1: p=2`
	`nach inc1: x=1`
	`$`

Beispiel: Zeiger übergeben

Programm	Aufruf

`$ 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);`
`}`
`$`
	`$ 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 K&R C!

Beispiel: Adressübergabe eines Integer in C++

Programm	Aufruf

$ cat inc4.c
#include <stdio.h>
void inc1(**int&** p) {
        p++;
        printf(" in inc1: p=%d\n", p);
}
main() {
        int x;
        x=1;
        printf(" vor inc1: x=%d\n", x);
        inc1(x);
        printf("nach inc1: x=%d\n", x);
}
$
	
	$ **cc** inc4.c -o inc4
	inc4.c:2: **parse error** before `&'
	$
	$ **g++** inc4.c -o inc4
	$
	$ inc4
	 vor inc1: x=1
	  in inc1: p=2
	nach inc1: 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++

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

K&R C

```
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

`$ 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));`
`}`
`$`
	`$ 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) ``
	`$`

<table>
<tr><th>C++</th><th>C</th></tr>
<tr><td>

```
$ cat for.cpp
#include <stdio.h>
int main()
{
        for (int i=1; i<=3; i++)
                printf("%d\n", i);
}
$
```

</td><td>

```
$ 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 `)'
```

</td></tr>
</table>

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  Aus-
       nahme,  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
```

STRCPY(3) Bibliotheksfunktionen STRCPY(3)

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 Zeicheketten 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
       auswertet.

       getchar() ist äquivalent zu getc(stdin).

       gets() liest einen Zeile von stdin in der 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  Zeichen
       gets()  lesen wird, ohne die Daten vorher zu kennen und da
       gets() fortfährt und Daten über das Ende des Puffers  hin-
       aus  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  Datei-
       deskriptor  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');
}
$
```

Beispiel (simpleCat):

```
$ cat simpleCat02.c
#include <stdio.h>
main() {
        char line[80];
        while (gets(line) != NULL)
                puts(line);
        putchar('\n');
}
$
```

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 aud 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

$
```

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 funk-
tinieren (irgendjemand sollte das testen und diese Manpage
berichtigen);  ANSI C verlangt, daß eine Dateizeiger-Posi-
tionierfunktion 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 Zeichenketten-variable, 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 Signalroutine 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 abgebrochen 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.

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 des 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);
        }
}
$
```

➔Rechner

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, "lssek 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  \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/schuette/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

$
```

Will man die Zeit messen, die ein Programmfragment zur Ausführung braucht, kann man auf Microsekundenebene mittels der Struktur timeval auf die real verflossene Zeit oder mittels clock_t die von der CPU verbrauchte Zeit (Ticks) zugreifen.

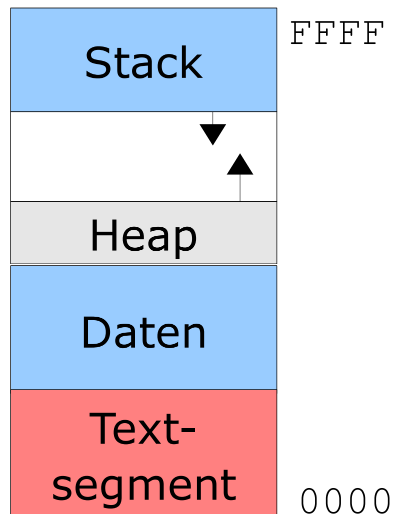
```
#include <iostream>
using namespace std;
#include <sys/time.h>

void f() {
    for (int i=0;i<999999999;i++) double j = i/17.0;
    sleep(10);
}
```

```
int main() {
  {
    /*
      struct timeval {
        unsigned long  tv_sec;   // seconds since 1.1.1970
        long           tv_usec;  // and microssecs (10^-6 secs)
      };
    */
    timeval start, end;
    gettimeofday(&start, 0);
    f();
    gettimeofday(&end, 0);
    cout << ((end.tv_sec*1000000+end.tv_usec)-
             (start.tv_sec*1000000+start.tv_usec))*0.000001 << " secs" << endl;
  }
  {
    /*
      CPU time used since programm start
    */
    clock_t start, end;
    start = clock();
    f();
    end = clock();
    cout << (1.0*end-start)/CLOCKS_PER_SEC << " secs" << endl;
  }
}
```

```
$ duration
17.895 secs
7.87479 secs
$
```

8.5. Prozessverwaltung

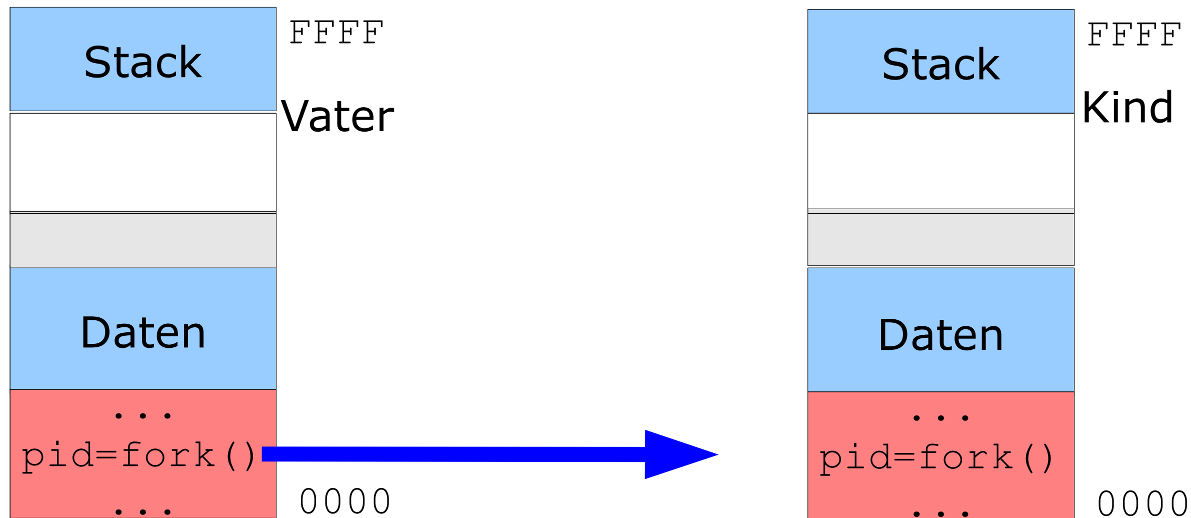
Ein Prozess besitzt einen Adressraum, in dem er abläuft. Der Prozessraum ist in mehrere Teile (Segmente) aufgeteilt.



Der Programmcode befindet sich im Textsegment.
Im Datensegment sind globale Objekte abgelegt, dann folgt der Heap für dynamische Objekte.
Der Stack ist zur Speicherung lokaler Objekte und für Rücksprungadressen bei Rekursionen nötig.

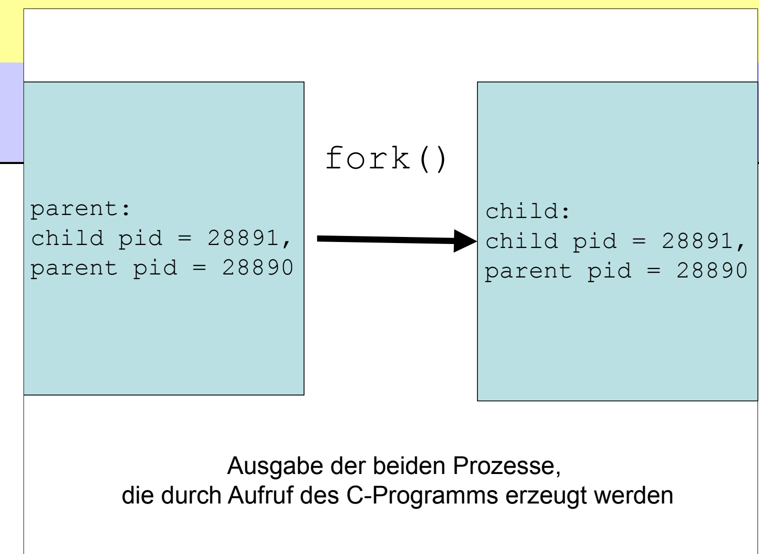
Ein Prozess wird erzeugt, in dem ein **Eltern Prozess** durch den Systemaufruf „**`fork`**" einen Kind Prozess erzeugt. Der Aufruf erzeugt eine **exakte Kopie** des Originalprozesses (Kind=Clone des Vaters), einschließlich aller Dateideskriptoren, Register usw.

Nach dem `fork` werden beide Prozesse unterschiedliche Aktivitäten übernehmen. Zum Zeitpunkt des `fork` haben alle Variablen die gleichen Werte, nach dem `fork` wirken sich Änderungen der Variablen nur noch im jeweiligen Prozess aus.



Der `fork` Aufruf gibt einen **Wert zurück**, durch den im Programm unterschieden werden kann, ob der Code des Kindes oder des Vaters gemeint ist: **0** ist der **Kindprozess**, Wert **grösser 0** ist die Prozessidentifikation (**pid**) des **Kindprozesses für den Eltern Prozess**. Ein Rückgabewert von `fork`, der **kleiner als 0** ist, zeigt an, dass kein neuer Prozess erzeugt werden konnte (**Fehler**).

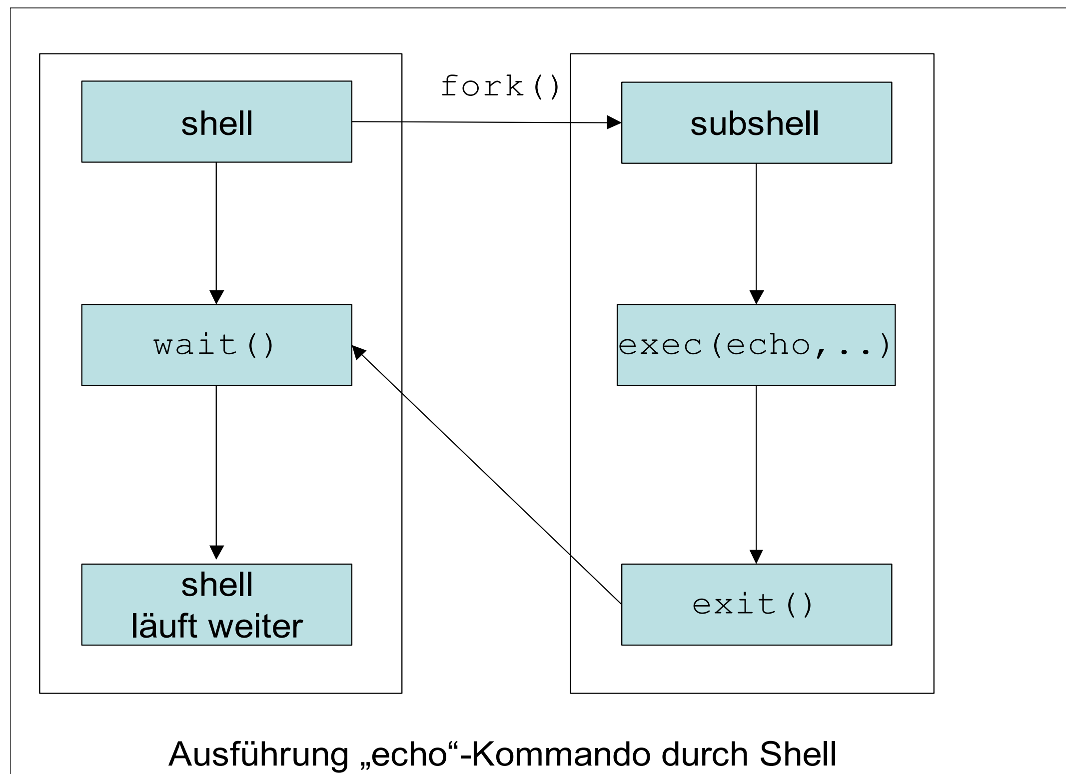
```
$ cat fork.c
#include <stdio.h>
main()
{
    int childPid;

    if ((childPid = fork()) == -1) {
      fprintf(stderr,"can't fork\n");
      exit(1);
   } else if (childPid == 0) { /* child process */
     fprintf(stdout, "child: child pid = %d, parent pid = %d\n",
     getpid(), getppid());
     exit(0);
   } else { /* parent process */
     fprintf(stdout, "parent: child pid = %d, parent pid = %d\n",
     childPid, getpid());
     exit(0);
   }
}
$
```



Ausgabe der beiden Prozesse,
die durch Aufruf des C-Programms erzeugt werden

Ein r**eales Beispiel**, bei dem ein Prozess erzeugt wird, ist die **Shell**. Für jedes Kommando, das aus der Shell heraus ausgeführt wird, wird von der Shell ein eigener Prozess erzeugt. Dabei dupliziert sich die Shell, überlagert den eigenen Code mit dem Code des auszuführenden Kommandos und wartet bis der so erzeugte Prozess terminiert.

Dazu sind die Systemaufrufe „`exec`" und „`wait`", wie im nachfolgenden Schaubild verdeutlicht, verwendet. Dabei von der Shell das Kommando echo aufgerufen.



Ausführung „echo"-Kommando durch Shell

Die Shell erzeugt durch fork einen Prozess und führt den Systemaufruf „`wait`" aus. Dadurch wartet sie, bis sie ein Signal erhält. Dieses Signal wird vom „`exit`" Aufruf des Kindprozesses

erzeugt. Der Kindprozess ruft den Systemaufruf „`exec`“ auf. Er nimmt den Code des ersten Parameters (hier das echo Kommando) und überlagert den eigenen Code damit. Somit kann in der Umgebung des Kindprozesses, das echo Kommando ausgeführt werden.

Das folgende Programm stellt eine Minishell dar, die nach dem o.g. Prinzip funktioniert.

```
void read_command(char *com, char **par){
  fprintf(stdout, "$ ");
  .....
  return;
}
int main(){ int childPid;
    int status;
    char command[20];
    char *parameters[60];
    while (1) {
      read_command(command, parameters);
      if ((childPid = fork()) == -1) {
        fprintf(stderr,"can't fork\n");
        exit(1);
      } else if (childPid == 0) { /* child */
        execv(command, parameters);
        exit(0);
      } else { /* parent process */
        wait(&status);
      }
    }
}
```

„`exit`" hat einen Parameter, den so genannten Exitstatus; dies ist ein Integerwert zwischen 0 und 255. Konvention in Unix ist, dass ein Exitstatus von Null bedeutet, dass die Aktion erfolgreich ausgeführt werden konnte. Jeder andere Wert wird als Fehler angesehen. Dieser Status wird dem Elternprozess in der Variablen „`status`" des wait Aufrufs mitgegeben.

Soll z.B. eine Kommunikation zwischen Kind und Eltern stattfinden, so kann das Kind z.B. durch

```
exit(4);
```

dem Elternprozess die Nachricht „4" übergeben. Der Elternprozess wird durch

```
wait(status);
```

dann die Information in der Variablen „`status`" sehen.

Im Wert von **Status** sind unterschiedliche Informationen kodiert. Neben dem **exit-Wert** des Kindes, sind **Informationen** über die **Terminierung des Kindes** abgelegt. Um diese Informationen auszulesen existieren C-Macros:

Macro	*Beschreibung*
`WIFEXITED(status)`	True, wenn status vom Kind gesetzt und das Kind normal beendet wurde. Dann kann man durch `WEXITSTATUS(status)` die niederwertigen 8 Bit auslesen, die den Exit-Wert des Kindes beinhalten
`WIFSGNALES(status)`	True, wenn status vom Kind gesetzt und das Kind abnormal beendet wurde, durch signal. Dann kann man durch `WTERMSIG(status)` die Signalnummer, die das Kind beendet hat, abfragen
`WIFSTOPPED(status)`	True, wenn status vom Kind gesetzt und das Kind gerade gestopped wurde. Dann kann man durch `WSTOPSIG(status)` die Signalnummer, die das Kind gestopped hat, abfragen

Der Aufruf von **`wait`**`()` bewirkt, dass der **Vaterprozess blockiert**, bis irgend **ein** Kinder beendet sind. Beim Systemaufruf `waitpid()` kann über Parameter gesteuert werden, wann und auf wen der Vater warten will (->später).

```
#include <stdio.h>
#include <sys/wait.h>
main()
{
  int childPid;
  int status;
  if ((childPid = fork()) == -1) {
    fprintf(stderr,"can't fork\n");
    exit(1);
  } else if (childPid == 0) {
    /* child process */
    fprintf(stdout, "child: exit 4\n");
    sleep(10);
    exit(4);
  } else  {
    /* parent process */
    if (wait(&status) != childPid) {
      fprintf(stderr, "wait error\n");
      exit(1);
    }
    fprintf(stdout, "parent: status %d\n", status);
    /* check status */
    if (WIFEXITED(status)) {
      fprintf(stdout, "parent: normal termination of child, status %d\n",
                      WEXITSTATUS(status) );
    } else if (WIFSIGNALED(status)) {
      fprintf(stdout, "parent: abnormal termination of child, status %d\n",
                      WEXITSTATUS(status) );
      fprintf(stdout, "parent: abnormal termination of child, signal number %d\n",
```

```
                      WTERMSIG(status) );
    } else if (WIFSTOPPED(status)) {
      fprintf(stdout, "parent: child stopped, status %d\n",
                      WEXITSTATUS(status) );
      fprintf(stdout, "parent: child stopped, signal number %d\n",
                      WSTOPSIG(status) );
    }
    exit(0);
  }
}
$ status
child: exit 4
parent: status 1024
parent: normal termination of child, status 4
$
```

Hörsaalübung

Welche Ausgabe erzeugt das folgende Programm?

```
#include <stdio.h>
#include <unistd.h> // wg. getpid
int x = 0;
int main() {
        fork();
        fork();
        fork();
        printf("pid=%d x=%d\n", getpid(), x++);
}
```

Mit `wait()` kann der Elternprozess nur auf die Beendigung irgend eines Kindes warten. Soll der Vater hingegen auf die Beendigung eines bestimmten Prozesses warten, dann ist **wait-pid()** zu verwenden. Mit dem ersten Argument von

```
waitpid(pid wpid, int *status, int options)
```

legt man fest, worauf gewartet werden soll:

pid	Bedeutung
-1	auf beliebigen Prozess warten – äquivalent zu wait()
pid	auf die Beendigung von pid warten.
0	auf Beendigung warten, dessen Prozessgruppen-ID gleich der Prozessgruppen-ID des aufrufenden Prozesses ist.
< -1	auf Beendigung warten, dessen Prozessgruppen-ID gleich dem absoluten Wert von pid ist.

Das dritte Argument „options“ bestimmt das Verhalten von waitpid(). Folgende Konstanten können dabei verwendet werden:

Konstante	Beschreibung
WNOHANG	Der aufrufende Prozess wird nicht blockiert, wenn der Prozess "pid" noch nicht beendet wurde bzw. noch nicht im Zombie-Status ist. waitpid() liefert in diesem Fall 0 zurück.
WUNTRACED	Status des angehaltenen Kindprozesses, der mit pid spezifiziert wurde.
WIFSTOPPED	liefert 1 zurück, wenn es sich beim Rückgabewert um die PID des angehaltenen Kindprozesses handelt.

Im folgenden Beispiel wird **nicht** auf die Terminierung des 1. Kindes gewartet, es wird ein zweites Kind erzeugt, auf dessen Beendigung gewartet wird.

```
$ cat waitpid.c
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
```

```
int main (void) {
  pid_t pid1, pid2;
  int status;
  switch (pid1 = fork ()) {
    case -1:
      perror("fork()");
      return EXIT_FAILURE;
    case 0: // child 1
      printf ("child 1: %d\n", getpid());
      sleep (10);
      printf ("child 1: %d terminated\n", getpid());
      break;
```

```
    default: // parent code
      if (waitpid (pid1, NULL, WNOHANG) != 0) { //will not wait for first child
        perror("waitpid()");
        return EXIT_FAILURE;
      }
      printf (" --- parent will not block, creating new child ---\n");
      switch (pid2 = fork ()) {
        case -1:
          perror("fork()");
          return EXIT_FAILURE;
        case 0: // child 2
          printf ("child 2: %d\n", getpid());
          sleep (5);
          printf ("child 2: %d terminated\n", getpid());
          break;
        default: // parent code
          if (wait(NULL) != pid2) { // will wait for sec child
            perror("waitpid()");
            return EXIT_FAILURE;
          }
          printf (" --- parent can continue ---\n");
          sleep(20);
          printf (" --- parent terminated ---\n");
      }
  }
  return EXIT_SUCCESS;
}
```

9. Terminaleigenschaften verändern

Die Eigenschaften eines Terminals, aus dem ein Programm gestartet wird, kann man beeinflussen. Verwendet ein eine Bibliothek und die Headerdatei termios.h.

Dies wird an einem Beispiel für ein Menüprogramm verdeutlicht, bei dem die Auswahl durch einen Tastendruck (ohne abschließendes Return) erfolgt.

```
$ cat readWithoutReturn.cpp
#include <termios.h>
#include <unistd.h> // read
#include <iostream>
using namespace std;

char getch(bool echoOn) { //read char from stdin without waiting for return key
  char buf=0;
  struct termios old = {0};
  if (tcgetattr(0, &old) < 0) perror("tcgetattr");
  old.c_lflag &= ~ICANON;
  if (!echoOn) old.c_lflag &= ~ECHO;    // echo off
  old.c_cc[VMIN] = 1;
  old.c_cc[VTIME] = 0;
  if (tcsetattr(0, TCSANOW, &old) < 0) perror("tcsetattr ICANOW");
  if (read(0, &buf, 1) < 0) perror("read");
  old.c_lflag |= ICANON;
  old.c_lflag |= ECHO;     // echo on
  if (tcsetattr(0, TCSADRAIN, &old) < 0) perror("tcsetattr ~ICANOW");
  return buf;
}
```

```
int main() {
   char c;
   cout <<  "Menue Demo, press any key to start! " << endl;
   getch(false);
   while (1) {
     system("clear");
     cout << " ---- Menue ----" << endl;
     cout << "1  item 1" << endl;
     cout << "2  item 2" << endl;
     cout << "0  exit" << endl;
     cout << "Your choice (0-2): ";
     cout.flush(); c = getch(true);
     switch (c) {
       case '1': cout << endl << "item 1" << endl;
                 sleep(2);
                 break;
       case '2': cout << endl << "item 2" << endl;
                 sleep(2);
                 break;
       case '0': cout << endl<< "bye" << endl;
                 exit(0);
       default: cout << endl << c << ": wrong input" << endl;
                 sleep(2);
     }
   }
}
$
```