

# E 3. Übung

- Besprechung 1. Aufgabe
- Programmiersprache C: Speicherklassen und Sichtbarkeit von Variablen
- Infos zur Aufgabe 3: Verzeichnisse
- Dateisystem: Systemaufrufe

## E.1 Aufgabe 1

- 1. Include, Deklarationen

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

void append_element(int value);
int remove_element(void);

struct listelement {
    int value;
    struct listelement *next;
};

struct listelement *first = NULL;
```

## E.1 Aufgabe 1 (2)

### ■ Anfügen an die Liste

```

void append_element(int value) {
    struct listelement *e;

    if (value < 0) return;

    e = (struct listelement*) malloc(sizeof(struct listelement));
    if (e == NULL) {
        perror("Kann Listenelement nicht anlegen.");
        exit(EXIT_FAILURE);
    }

    e->value = value;
    e->next = NULL;

    if (first == NULL) {
        first = e;
    } else {
        /* Hinweis: man vermeidet das Durchlaufen der Liste, wenn man
           einen Zeiger auf das Listenende vorhaelt.
           Hier aber einfaches Durchlaufen.
        */
        struct listelement *p;
        for(p=first; p->next != NULL; p=p->next);
        p->next = e;
    }
}

```

## E.1 Aufgabe 1 (3)

### ■ Entnehmen aus Liste

```

int remove_element() {
    struct listelement *e;
    int v;
    if (first == NULL) return -1;
    v = first->value;
    e = first;
    first = first->next;
    free(e);
    return v;
}

```

## E.1 Aufgabe 1 (4)

- Fehlerbehandlung nicht vergessen!

```
e = (struct listelement*) malloc(sizeof(struct listelement));
if (e == NULL) {
    perror("Kann Listenelement nicht anlegen.");
    exit(EXIT_FAILURE);
}
```

- Fehlermeldungen immer auf `stderr` ausgeben!

z.B. mit `fprintf`

```
fprintf(stderr, "%s(%d): %s\n", __FILE__, __LINE__, strerror(errno));
```

oder mit `perror`

```
perror("Beschreibung wobei");
```

- ...

## E.2 Programmstruktur & Module

### 1 Softwaredesign

- Grundsätzliche Überlegungen über die Struktur eines Programms **vor** Beginn der Programmierung
- Verschiedene Design-Methoden
  - ◆ Top-down Entwurf / Prozedurale Programmierung
    - traditionelle Methode
    - bis Mitte der 80er Jahre fast ausschließlich verwendet
    - an Programmiersprachen wie Fortran, Cobol, Pascal oder C orientiert
  - ◆ Objekt-orientierter Entwurf
    - moderne, sehr aktuelle Methode
    - Ziel: Bewältigung sehr komplexer Probleme
    - auf Programmiersprachen wie C++, Smalltalk oder Java ausgerichtet

## 2 Top-down Entwurf

### ■ Zentrale Fragestellung

- ◆ was ist zu tun?
- ◆ in welche Teilaufgaben lässt sich die Aufgabe untergliedern?

- Beispiel: Rechnung für Kunden ausgeben
  - Rechnungspositionen zusammenstellen
    - Lieferungsposten einlesen
    - Preis für Produkt ermitteln
    - Mehrwertsteuer ermitteln
  - Rechnungspositionen addieren
  - Positionen formatiert ausdrucken

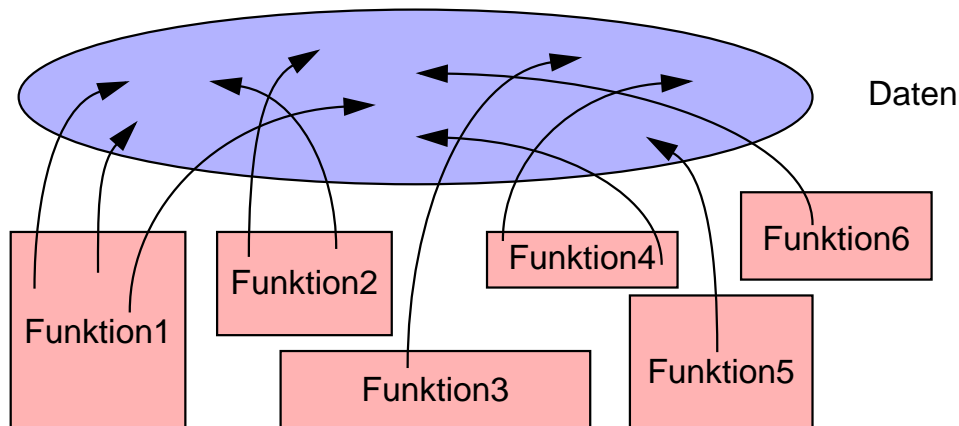
## 2 Top-down Entwurf (2)

### ■ Problem:

Gliederung betrifft nur die Aktivitäten, nicht die Struktur der Daten

### ■ Gefahr:

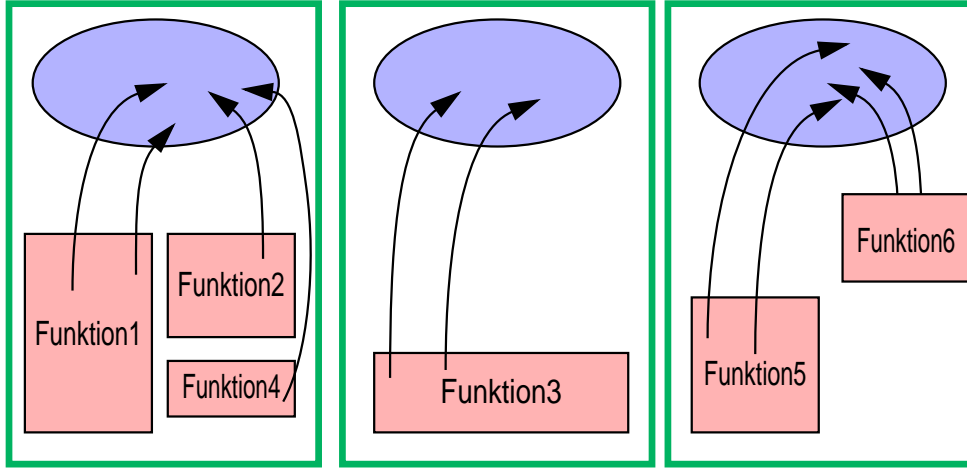
Sehr viele Funktionen arbeiten "wild" auf einer Unmenge schlecht strukturierter Daten



## 2 Top-down Entwurf (3) Modul-Bildung

- Lösung:  
Gliederung von Datenbeständen zusammen mit Funktionen, die darauf operieren

➔ **Modul**



Übungen zu Systemprogrammierung I

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2003

E-Uebung3.fm 2003-11-12 09.14

E.9

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

## 3 Module in C

- Teile eines C-Programms können auf mehrere .c-Dateien (C-Quelldateien) verteilt werden
- Logisch zusammengehörende Daten und die darauf operierenden Funktionen sollten jeweils zusammengefaßt werden

➔ **Modul**

- Jede C-Quelldatei kann separat übersetzt werden (Option `-c`)
  - Zwischenergebnis der Übersetzung wird in einer `.o`-Datei abgelegt

```
% cc -c main.c           (erzeugt Datei main.o)
% cc -c f1.c             (erzeugt Datei f1.o)
% cc -c f2.c f3.c       (erzeugt f2.o und f3.o)
```

- Das Kommando `cc` kann mehrere `.c`-Dateien übersetzen und das Ergebnis — zusammen mit `.o`-Dateien — binden:

```
% cc -o prog main.o f1.o f2.o f3.o f4.c f5.c
```

Übungen zu Systemprogrammierung I

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2003

E-Uebung3.fm 2003-11-12 09.14

E.10

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

### 3 Module in C

- !!! **.c-Quelldateien auf keinen Fall mit Hilfe der `#include`-Anweisung in andere Quelldateien einkopieren**
- Bevor eine Funktion aus einem anderen Modul aufgerufen werden kann, muss sie **deklariert** werden
  - Parameter und Rückgabewerte müssen bekannt gemacht werden
- Makrodefinitionen und Deklarationen, die in mehreren Quelldateien eines Programms benötigt werden, werden zu **Header-Dateien** zusammengefaßt
  - ◆ *Header-Dateien* werden mit der `#include`-Anweisung des Preprozessors in C-Quelldateien einkopiert
  - ◆ der Name einer *Header-Datei* endet immer auf `.h`

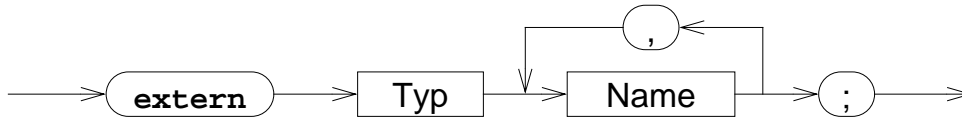
### 4 Gültigkeit von Namen

- Gültigkeitsregeln legen fest, welche Namen (Variablen und Funktionen) wo im Programm bekannt sind
- Mehrere Stufen
  1. Global im gesamten Programm  
(über Modul- und Funktionsgrenzen hinweg)
  2. Global in einem Modul  
(auch über Funktionsgrenzen hinweg)
  3. Lokal innerhalb einer Funktion
  4. Lokal innerhalb eines Blocks
- Überdeckung bei Namensgleichheit
  - eine lokale Variable innerhalb einer Funktion überdeckt gleichnamige globale Variablen
  - eine lokale Variable innerhalb eines Blocks überdeckt gleichnamige globale Variablen und gleichnamige lokale Variablen in umgebenden Blöcken

## 5 Globale Variablen

Gültig im gesamten Programm

- Globale Variablen werden außerhalb von Funktionen definiert
- Globale Variablen sind ab der Definition in der gesamten Datei zugreifbar
- Globale Variablen, die in anderen Modulen **definiert** wurden, müssen vor dem ersten Zugriff bekanntgemacht werden  
( **extern-Deklaration** = Typ und Name bekanntmachen)



- Beispiele:

```
extern int a, b;
extern char c;
```

## 5 Globale Variablen (2)

- Probleme mit globalen Variablen

- ◆ Zusammenhang zwischen Daten und darauf operierendem Programmcode geht verloren
- ◆ Funktionen können Variablen ändern, ohne daß der Aufrufer dies erwartet (Seiteneffekte)
- ◆ Programme sind schwer zu pflegen, weil bei Änderungen der Variablen erst alle Programmteile, die sie nutzen gesucht werden müssen

➔ **globale Variablen möglichst vermeiden!!!**

## 5 Globale Funktionen

- Funktionen sind generell global  
(es sei denn, die Erreichbarkeit wird explizit auf das Modul begrenzt)
- Funktionen aus anderen Modulen müssen ebenfalls vor dem ersten Aufruf **deklariert** werden  
(= Typ, Name und Parametertypen bekanntmachen)
- Das Schlüsselwort **extern** ist bei einer Funktionsdeklaration nicht notwendig

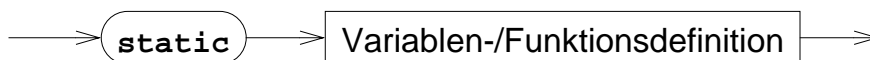
- Beispiele:

```
double sinus(double);
float power(float, int);
```

- Globale Funktionen (und soweit vorhanden die globalen Daten) bilden die äußere Schnittstelle eines Moduls
  - "vertragliche" Zusicherung an den Benutzer des Moduls

## 6 Einschränkung der Gültigkeit auf ein Modul

- Zugriff auf eine globale Variable oder Funktion kann auf das Modul (= die Datei) beschränkt werden, in der sie definiert wurde
  - Schlüsselwort **static** vor die Definition setzen



- **extern**-Deklarationen in anderen Modulen sind nicht möglich
- Die **static**-Variablen bilden zusammen den Zustand eines Moduls, die Funktionen des Moduls operieren auf diesem Zustand
- Hilfsfunktionen innerhalb eines Moduls, die nur von den Modulfunktionen benötigt werden, sollten immer **static** definiert werden
  - sie werden dadurch nicht Bestandteil der Modulschnittstelle  
(= des "Vertrags" mit den Modulbenutzern)

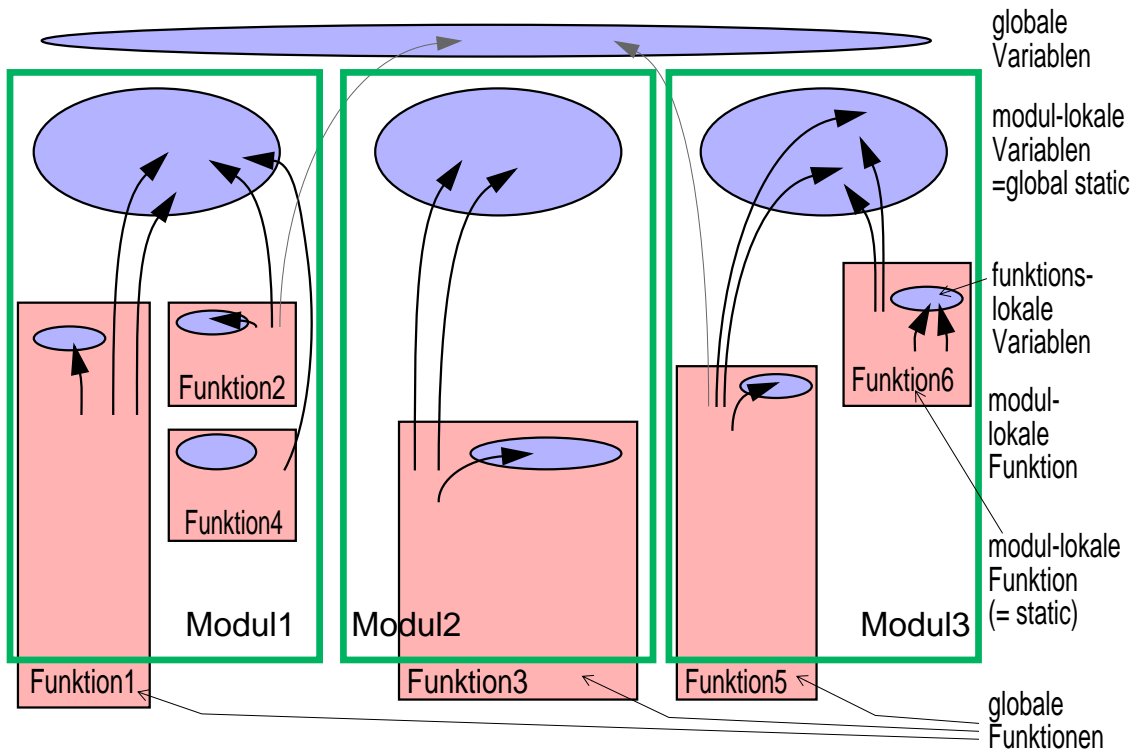
- !!! das Schlüsselwort **static** gibt es auch bei lokalen Variablen  
(mit anderer Bedeutung!)



## 7 Lokale Variablen

- Variablen, die innerhalb einer Funktion oder eines Blocks definiert werden, sind lokale Variablen
- bei Namensgleichheit zu globalen Variablen oder lokalen Variablen eines umgebenden Blocks gilt die jeweils letzte Definition
- lokale Variablen sind außerhalb des Blocks, in dem sie definiert wurden, nicht zugreifbar und haben dort keinen Einfluß auf die Zugreifbarkeit von Variablen

## 8 Gültigkeitsbereiche — Übersicht



## 9 Lebensdauer von Variablen

- Die Lebensdauer einer Variablen bestimmt, wie lange der Speicherplatz für die Variable aufgehoben wird
- Zwei Arten
  - ◆ Speicherplatz bleibt für die gesamte Programmausführungszeit reserviert
    - statische (`static`) Variablen
  - ◆ Speicherplatz wird bei Betreten eines Blocks reserviert und danach wieder freigegeben
    - dynamische (`automatic`) Variablen

## 9 Lebensdauer von Variablen (2)

### auto-Variablen

- Alle lokalen Variablen sind automatic-Variablen
  - der Speicher wird bei Betreten des Blocks / der Funktion reserviert und bei Verlassen wieder freigegeben
    - ➔ der Wert einer lokalen Variablen ist beim nächsten Betreten des Blocks nicht mehr sicher verfügbar!
- Lokale auto-Variablen können durch beliebige Ausdrücke initialisiert werden
  - die Initialisierung wird bei jedem Eintritt in den Block wiederholt
  - !!! wird eine auto-Variable nicht initialisiert, ist ihr Wert vor der ersten Zuweisung undefiniert (= irgendwas)**

## 9 Lebensdauer von Variablen (2)

### static-Variablen

- Der Speicher für alle globalen Variablen ist generell von Programmstart bis Programmende reserviert
- Lokale Variablen erhalten bei Definition mit dem Schlüsselwort `static` eine **Lebensdauer über die gesamte Programmausführung** hinweg
  - ↳ der Inhalt bleibt bei Verlassen des Blocks erhalten und ist bei einem erneuten Eintreten in den Block noch verfügbar
- !!! Das Schlüsselwort `static` hat bei globalen Variablen eine völlig andere Bedeutung (Einschränkung des Zugriffs auf das Modul)
- Static-Variablen können durch beliebige konstante Ausdrücke initialisiert werden
  - die Initialisierung wird nur einmal beim Programmstart vorgenommen (auch bei lokalen Variablen!)
  - erfolgt keine explizite Initialisierung, wird automatisch mit 0 vorbelegt

## 10 Wertaustausch zwischen Funktionen

Mechanismus	Aufrufer → Funktion	Funktion → Aufrufer
Parameter	ja	mit Hilfe von Zeigern
Funktionswert	nein	ja
globale Variablen	ja	ja

- Verwendung globaler Variablen?
  - ◆ Variablen, die von vielen Funktionen verwendet werden und/oder oft als Parameter übergeben werden müßten
    - Menge der Funktionen muß überschaubar bleiben
    - ➔ Zugriff auf Modul begrenzen (globale static-Variablen)
    - **sonst sehr schlechter Programmierstil**
  - ◆ Variablen, die keiner Funktion als Variable oder Parameter fest zugeordnet werden können
    - Modul suchen, dem die Variable zugeordnet werden kann!!!
  - ◆ Variablen, deren Lebensdauer nicht beschränkt sein darf, die aber nicht in `main()` deklariert werden sollen
    - in zugehöriger Funktion lokal-static definieren

# 11 Getrennte Übersetzung von Programmteilen — Beispiel

## ■ Hauptprogramm (Datei `fplot.c`)

```
#include "trig.h"
#define INTERVALL 0.01

/*
 * Funktionswerte ausgeben
 */
int main(void)
{
    char c;
    double i;

    printf("Funktion (Sin, Cos, Tan, cOt)? ");
    scanf("%x", &c);

    switch (c) {
        ...
        case 'T':
            for (i=-PI/2; i < PI/2; i+=INTERVALL)
                printf("%lf %lf\n", i, tan(i));
            break;;
        ...
    }
}
```

# 11 Getrennte Übersetzung — Beispiel (2)

## ■ Header-Datei (Datei `trig.h`)

```
#include <stdio.h>
#define PI 3.1415926535897932
double tan(double), cot(double);
double cos(double), sin(double);
```

## 11 Getrennte Übersetzung — Beispiel (3)

- Trigonometrische Funktionen  
(Datei `trigfunc.c`)

```
#include "trig.h"

double tan(double x) {
    return(sin(x)/cos(x));
}

double cot(double x) {
    return(cos(x)/sin(x));
}

double cos(double x) {
    return(sin(PI/2-x));
}
```

...

## 11 Getrennte Übersetzung — Beispiel (4)

- Trigonometrische Funktionen — Fortsetzung  
(Datei `trigfunc.c`)

...

```
double sin (double x)
{
    double summe;
    double x_quadrat;
    double rest;
    int k;

    k = 0;
    summe = 0.0;
    rest = x;
    x_quadrat = x*x;

    while ( fabs(rest) > 1e-9 ) {
        summe += rest;
        k += 2;
        rest *= -x_quadrat/(k*(k+1));
    }
    return(summe);
}
```

## E.3 Aufgabe 3: Verzeichnisse

- opendir(3), readdir(3), closedir(3)
- stat(2), lstat(2)
- readlink(2)
- getpwuid(3), getgrgid(3)

### 1 opendir

- Funktions-Prototyp:

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

DIR *opendir(const char *dirname);
```

- Argumente
  - ◆ **dirname**: Verzeichnisname
- Rückgabewert: Zeiger auf Datenstruktur vom Typ **DIR** oder **NULL**

## 2 readdir

### ■ Funktions-Prototyp:

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

struct dirent *readdir(DIR *dirp);
```

### ■ Argumente

- ◆ **dirp**: Zeiger auf **DIR**-Datenstruktur

### ■ Rückgabewert: Zeiger auf Datenstruktur vom Typ **struct dirent** oder **NULL** wenn fertig oder Fehler (**errno** vorher auf 0 setzen!)

### ■ Probleme: Der Speicher für **struct dirent** wird von der Bibliothek wieder verwendet!

## 3 struct dirent

### ■ Definition unter Linux (/usr/include/bits/dirent.h)

```
struct dirent {
    __ino_t d_ino;
    __off_t d_off;
    unsigned short int d_reclen;
    unsigned char d_type;
    char d_name[256];
};
```

### ■ Definition unter Solaris (/usr/include/sys/dirent.h)

```
typedef struct dirent {
    ino_t          d_ino;
    off_t          d_off;
    unsigned short d_reclen;
    char           d_name[1];
} dirent_t;
```

## 4 stat / lstat

### ■ Funktions-Prototyp:

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

### ■ Argumente:

- ◆ **path**: Dateiname
- ◆ **buf**: Puffer für Inode-Informationen

### ■ Rückgabewert: 0 wenn OK, -1 wenn Fehler

### ■ Beispiel:

```
struct stat buf;
stat("/etc/passwd", &buf); /* Fehlerabfrage ... */
printf("Inode-Nummer: %d\n", buf.st_ino);
```

## 5 stat / lstat: stat-Struktur

- **dev\_t st\_dev**; Gerätenummer
- **ino\_t st\_ino**; Inodenummer
- **mode\_t st\_mode**; Dateimode, u.a. Zugriffs-Bits (siehe `chmod(1)`)
- **nlink\_t st\_nlink**; Anzahl der (Hard-) Links auf den Inode
- **uid\_t st\_uid**; UID des Besitzers
- **gid\_t st\_gid**; GID der Dateigruppe
- **dev\_t st\_rdev**; DeviceID, nur für Character oder Blockdevices
- **off\_t st\_size**; Dateigröße in Bytes
- **time\_t st\_atime**; Zeit des letzten Zugriffs (in Sekunden seit 1.1.1970)
- **time\_t st\_mtime**; Zeit der letzten Veränderung (in Sekunden ...)
- **time\_t st\_ctime**; Zeit der letzten Änderung der Inode-Information (...)
- **unsigned long st\_blksize**; Blockgröße des Dateisystems
- **unsigned long st\_blocks**; Anzahl der von der Datei belegten Blöcke



## 6 readlink

### ■ Funktions-Prototyp:

```
#include <unistd.h>

int readlink(const char *path, char *buf, size_t bufsiz);
```

### ■ Argumente

- ◆ `path`: Dateiname
- ◆ `buf`: Puffer für Link-Inhalt
- ◆ `bufsiz`: Größe des Puffers

### ■ Rückgabewert: Anzahl der Bytes oder -1

## 7 getpwuid

### ■ Funktions-Prototyp:

```
#include <pwd.h>
struct passwd *getpwuid(uid_t uid);
```

### ■ struct passwd:

- ◆ `char *pw_name;` /\* user's login name \*/
- ◆ `uid_t pw_uid;` /\* user's uid \*/
- ◆ `gid_t pw_gid;` /\* user's gid \*/
- ◆ `char *pw_gecos;` /\* typically user's full name \*/
- ◆ `char *pw_dir;` /\* user's home dir \*/
- ◆ `char *pw_shell;` /\* user's login shell \*/

## 8 getgrgid

### ■ Prototyp:

```
#include <grp.h>
struct group *getgrgid(gid_t gid);
```

### ■ struct group:

- ◆ char \*gr\_name; /\* the name of the group \*/
- ◆ char \*gr\_passwd; /\* the encrypted group password \*/
- ◆ gid\_t gr\_gid; /\* the numerical group ID \*/
- ◆ char \*\*gr\_mem; /\* vector of pointers to member names \*/

## E.4 Dateisystem Systemcalls

- open(2) / close(2)
- read(2) / write(2)
- lseek(2)
- chmod(2)
- umask(2)
- utime(2)
- truncate(2)

# 1 open

## ■ Funktions-Prototyp:

```
#include <fcntl.h>
int open(const char *path, int oflag, ... /* [mode_t mode] */ );
```

## ■ Argumente:

- ◆ Maximallänge von path: `PATH_MAX`
- ◆ `oflag`: Lese/Schreib-Flags, Allgemeine Flags, Synchronisierungs I/O Flags
  - Lese/Schreib-Flags: `O_RDONLY`, `O_WRONLY`, `O_RDWR`
  - Allgemeine Flags: `O_APPEND`, `O_CREAT`, `O_EXCL`, `O_LARGEFILE`, `O_NDELAY`, `O_NOCTTY`, `O_NONBLOCK`, `O_TRUNC`
  - Synchronisierung: `O_DSYNC`, `O_RSYNC`, `O_SYNC`
- ◆ `mode`: Zugriffsrechte der erzeugten Datei (nur bei `O_CREAT`) - siehe `chmod`

## ■ Rückgabewert

- ◆ Filedeskriptor oder -1 im Fehlerfall (`errno` wird gesetzt)

# 1 open - Flags

- `O_EXCL`: zusammen mit `O_CREAT` - nur *neue* Datei anlegen
- `O_TRUNC`: Datei wird beim Öffnen auf 0 Bytes gekürzt
- `O_APPEND`: vor jedem Schreiben wird der Dateizeiger auf das Dateiende gesetzt
- `O_NDELAY`, `O_NONBLOCK`: Operationen arbeiten nicht-blockierend (bei Pipes, FIFOs und Devices)
  - ◆ `open` kehrt sofort zurück
  - ◆ `read` liefert -1 zurück, wenn keine Daten verfügbar sind
  - ◆ wenn genügend Platz ist, schreibt `write` alle Bytes, sonst schreibt `write` nichts und kehrt mit -1 zurück
- `O_NOCTTY`: beim Öffnen von Terminal-Devices wird das Device nicht zum Kontroll-Terminal des Prozesses

## 1 open - Flags (2)

- Synchronisierung
  - ◆ `O_DSYNC`: Schreibaufruf kehrt erst zurück, wenn Daten in Datei geschrieben wurden (Blockbuffer Cache!!)
  - ◆ `O_SYNC`: ähnlich `O_DSYNC`, zusätzlich wird gewartet, bis Datei-Attribute wie Zugriffszeit, Modifizierungszeit, auf Disk geschrieben sind
  - ◆ `O_RSYNC` | `O_DSYNC`: Daten die gelesen wurden, stimmen mit Daten auf Disk überein, d.h. vor dem Lesen wird der Buffercache geflushet
  - `O_RSYNC` | `O_SYNC`: wie `O_RSYNC` | `O_DSYNC`, zusätzlich Datei-Attribute

## 2 close

- Funktions-Prototyp:

```
#include <unistd.h>
int close(int fildes);
```

- Argumente:
  - ◆ `fildes`: Filedeskriptor der zu schließenden Datei
- Rückgabewert:
  - ◆ 0 bei Erfolg, -1 im Fehlerfall

## 3 read

### ■ Funktions-Prototyp:

```
#include <unistd.h>
ssize_t read(int fildes, void *buf, size_t nbyte);
```

### ■ Argumente

- ◆ **fildes**: Filedeskriptor, z.B. Rückgabe vom open-Aufruf
- ◆ **buf**: Zeiger auf Puffer
- ◆ **nbyte**: Größe des Puffers

### ■ Rückgabewert

- ◆ Anzahl der gelesenen Bytes oder -1 im Fehlerfall

```
char buf[1024];
int fd;
fd = open("/etc/passwd", O_RDONLY);
if (fd == -1) ...
read(fd, buf, 1024);
```

## 4 write

### ■ Funktions-Prototyp

```
#include <unistd.h>
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

### ■ Argumente

- ◆ äquivalent zu **read**

### ■ Rückgabewert

- ◆ Anzahl der geschriebenen Bytes oder -1 im Fehlerfall

## 5 lseek

### ■ Funktions-Prototyp

```
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

### ■ Argumente

- ◆ **fildes**: Filedeskriptor
- ◆ **offset**: neuer Wert des Dateizeigers
- ◆ **whence**: Bedeutung von offset
  - **SEEK\_SET**: absolut vom Dateianfang
  - **SEEK\_CUR**: Inkrement vom aktuellen Stand des Dateizeigers
  - **SEEK\_END**: Inkrement vom Ende der Datei

### ■ Rückgabewert

- ◆ Offset in Bytes vom Beginn der Datei oder -1 im Fehlerfall

## 6 chmod

### ■ Funktions-Prototyp:

```
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
```

### ■ Argumente:

- ◆ **path**: Dateiname
- ◆ **mode**: gewünschter Dateimodus, z.B.
  - **S\_IRUSR**: lesbar durch Besitzer
  - **S\_IWUSR**: schreibbar durch Benutzer
  - **S\_IRGRP**: lesbar durch Gruppe

### ■ Rückgabewert: 0 wenn OK, -1 wenn Fehler

### ■ Beispiel:

```
chmod("/etc/passwd", S_IRUSR | S_IRGRP);
```

## 7 umask

### ■ Funktions-Prototyp:

```
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

### ■ Argumente

- ◆ **cmask**: gibt Permission-Bits an, die beim Erzeugen einer Datei ausgeschaltet werden sollen

### ■ Rückgabewert

- ◆ voriger Wert der Maske

## 8 utime

### ■ Funktions-Prototyp:

```
#include <utime.h>
int utime(const char *path, const struct utimbuf *times);
```

### ■ Argumente

- ◆ **path**: Dateiname
- ◆ **times**: Zugriffs- und Modifizierungszeit (in Sekunden)

### ■ Rückgabewert: 0 wenn OK, -1 wenn Fehler

### ■ Beispiel: setze atime und mtime um eine Stunde zurück

```
struct utimbuf times;
struct stat buf;
stat("/etc/passwd", &buf); /* Fehlerabfrage */
times.actime = buf.st_atime - 60 * 60;
times.modtime = buf.st_mtime - 60 * 60;
utime("/etc/passwd", &times); /* Fehlerabfrage */
```

## 9 truncate

### ■ Funktions-Prototyp:

```
#include <unistd.h>
int truncate(const char *path, off_t length);
```

### ■ Argumente:

- ◆ **path**: Dateiname
- ◆ **length**: gewünschte Länge der Datei

### ■ Rückgabewert: 0 wenn OK, -1 wenn Fehler

## E.5 POSIX I/O vs. Standard-C-I/O

- POSIX Funktionen open/close/read/write/... arbeiten mit Filedescriptoren
- Standard-C Funktionen fopen/fclose/fgets/... arbeiten mit Filepointern
- Konvertierung von Filepointer nach Filedescriptor

```
#include <stdio.h>
int fileno(FILE *stream);
```

### ■ Konvertierung von Filedescriptor nach Filepointer

```
#include <stdio.h>
FILE *fdopen(int fd, const char* type);
```

- ◆ type kann sein "r", "w", "a", "r+", "w+", "a+"  
(fd muß entsprechend geöffnet sein!)

### ■ Filedescriptoren in <unistd.h>:

STDIN\_FILENO, STDOUT\_FILENO, STDERR\_FILENO