Agenda

1. Einführung in die Programmiersprache C ☑
2. Gültigkeitsbereiche, komplexe Datentypen ☑
3. Kontrollstrukturen, Ein- und Ausgabe ☑
4. Zeiger, Felder und Zeichenketten ☑
5. **Makros, C-Entwicklungswerkzeuge**
6. Dateisystem
7. Ausgewählte Beispiele
   (Prozesse, Threads, …)

© 2018 Prof. Dr. Stefan Böhmer

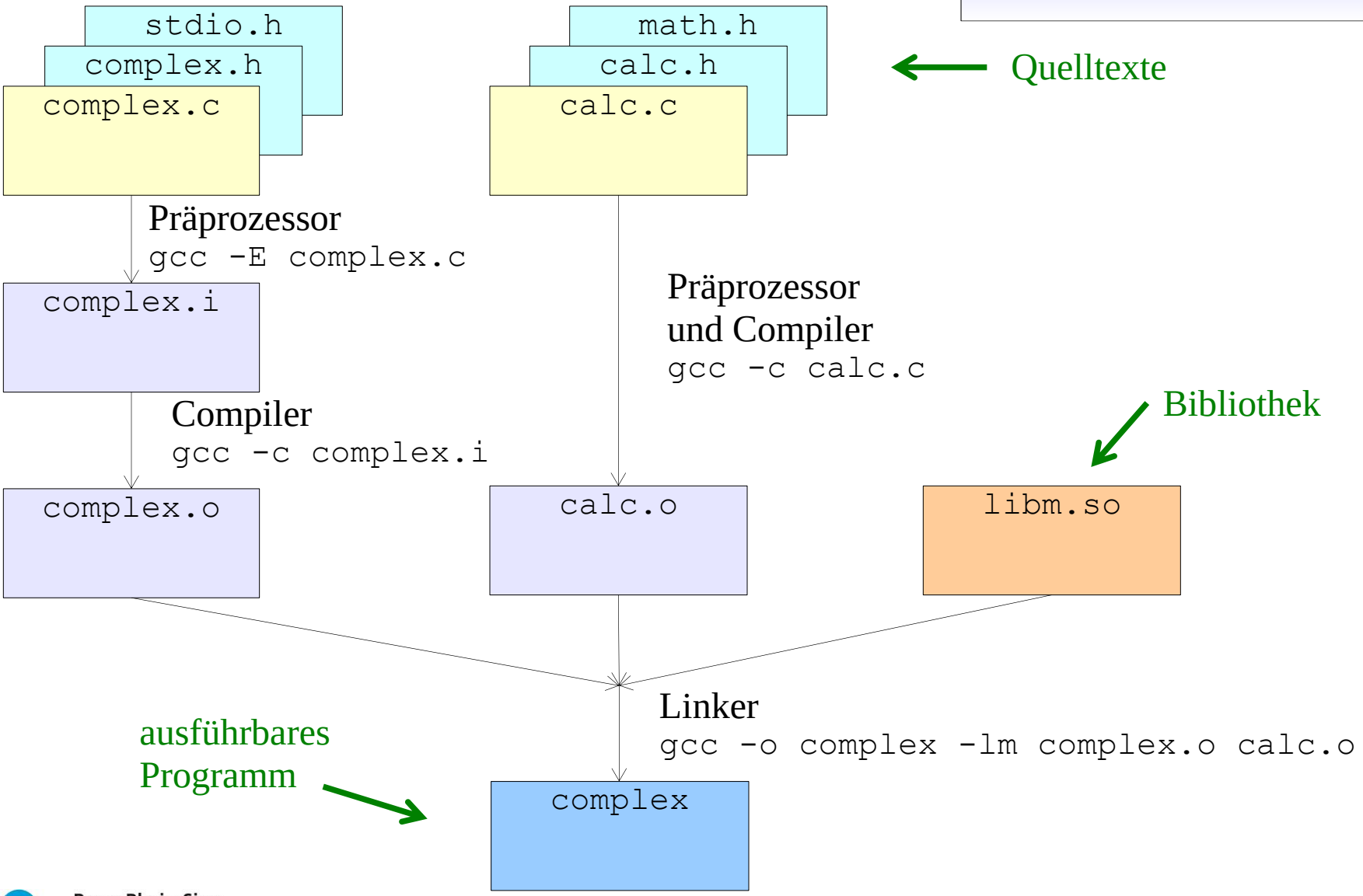
Vom Quelltext zum ausführbaren Code

Übersicht über C-Entwicklungswerkzeuge

- Präprozessor (**`gcc`**)
  - Textverarbeitung
  - Anweisungen an den Präprozessor werden mit '#' eingeleitet
- Compiler (gcc)
  - Übersetzt C-Quellcode in Objekt-Code
  - ggf. Zwischenschritt über Assembler-Code
- Assembler (**`gcc`**)
  - Übersetzt Assembler-Quellcode in Objekt-Code
- Linker (**`gcc`**)
  - Bindet alle Objekt-Dateien und ggf. Library-Dateien zu einer ausführbaren Binär-Datei zusammen
- Make (**`make`**)
  - Steuert Übersetzungs- und Bindevorgänge

© 2018 Prof. Dr. Stefan Böhmer

Vom Quelltext zum ausführbaren Code



© 2018 Prof. Dr. Stefan Böhmer

Präprozessor

- Aufgabe: Vorverarbeitung des Quellcodes
- Im Quellcode werden Regeln zur Textverarbeitung angegeben
- Der Präprozessor wird durch Direktiven gesteuert:

`#include`-Direktive:	Einfügen einer Datei
`#define`-Direktive:	Definition eines Makros
`#if`-Direktive:	Bedingte Übersetzung
`#pragma`-Direktive:	Compiler-Steuerung

- Direktiven sind zeilenorientiert.
  Direktiven können durch Verwendung eines ‚Backslash' am Zeilenende auf mehrere Zeilen verteilt werden

© 2018 Prof. Dr. Stefan Böhmer

Präprozessor

#include-Direktive

- Einfügen einer Datei
- Meistens werden Header-Dateien (**`*.h`**) eingebunden
- Verschachtelung ist möglich, so dass eine per **`#include`** eingebundene Datei selbst Dateien einbinden kann
- Die **`#include`**-Direktive kann an beliebiger Stelle im Quelltext positioniert werden
- Syntax:

  `#include "dateiname"`

  Die Datei wird relativ zum aktuellen Pfad gesucht

  oder

  `#include <dateiname>`

  Die Datei wird in system- bzw. compilerspezifischen Pfaden gesucht. Unter Linux sind dies z.B. : **`/usr/include`** und **`/usr/local/include`**
- Der Dateiname kann relative oder absolute Pfadangaben enthalten

© 2018 Prof. Dr. Stefan Böhmer

Präprozessor

#include-Direktive

Beispiel:

```
/* include.c */

#include "include1.h"

int main(int argc, char **argv)
{
  double radius = 2.5;
  double flaeche;

  flaeche = Flaeche(radius);
}
```

```
/* include1.h */

#include "include2.h"

extern double Flaeche(double r);
```

```
/* include2.h */

const double PI = 3.14159;
const double e  = 2.71828;
```

Präprozessor

```
const double PI = 3.14159;
const double e  = 2.71828;

extern double Flaeche(double r);

int main(int argc, char **argv)
{
  double radius = 2.5;
  double flaeche;

  flaeche = Flaeche(radius);
}
```


© 2018 Prof. Dr. Stefan Böhmer

Präprozessor

#define-Direktive

- Es wird ein Makro definiert.
- Anwendung: Definition von Konstanten, Abkürzung wiederkehrender Zeichenfolgen, Mapping und bedingte Übersetzung
- Syntax:

```
#define makroname
```

oder:

```
#define makroname ersetzungstext
```

oder:

```
#define makroname() ersetzungstext
```

oder:

```
#define makroname(parameterliste) ersetzungstext
```

- Der Präprozessor ersetzt im folgenden Text **`makroname`** durch den **`ersetzungstext`**
- Mit

```
#undef makroname
```

kann die Definition wieder aufgehoben werden

© 2018 Prof. Dr. Stefan Böhmer

Präprozessor

Beispiele #define-Direktive

Quelltext:

```
#define PI 3.14159

umfang = 2.0*PI*r;
```

Durch Präprozessor erzeugter Code:

```
umfang = 2.0*3.14159*r;
```

```
#define QUADRAT(X) (X*X)

y = QUADRAT(r);
z = QUADRAT(3.2E+7);
```

```
y = (r*r);
z = (3.2E+7*3.2E+7);
```

```
#define ERROR(...)                \
  fprintf(stderr,__VA_ARGS__)

ERROR("Fehler (%d): %s",7,file);

ERROR("das war wohl nix...");
```

```
fprintf(stderr,"Fehler (%d): %s",7,file);

fprintf(stderr,"das war wohl nix...");
```


© 2018 Prof. Dr. Stefan Böhmer

Präprozessor

#define-Direktive

- Die Ersetzung kann mit einem **#**-Operator weiter modifiziert werden:
  - Wird im Ersetzungstext dem Parameter ein '#' vorangestellt, so wird dieser in Anführungszeichen gesetzt.
  - Zeichenfolgen im Ersetzungstext können mit einem '##' zusammen-gezogen werden

- Beispiele:

Quelltext:

```
#define DEBUG_AUSGABE(X)        \
  printf("Wert" #X "=%d\n", X)

DEBUG_AUSGABE(zaehler);
```

Durch Präprozessor erzeugter Code:

```
printf("Wert" "zaehler" "=%d", zaehler);
```

```
#define MAPPING(X)              \
   (modulA_##X = modulB_##X)

MAPPING(re);
MAPPING(li);
```

```
(modulA_re = modulB_re);
(modulA_li = modulB_li);
```


© 2018 Prof. Dr. Stefan Böhmer

Präprozessor

#define-Direktive

- Es lassen sich auch komplexere Konstrukte als Makro zusammenfassen
- Beispiel:

Quelltext:

```
#define ERROR(...)                  \
  fprintf(stderr,__VA_ARGS__)

#define OPEN(FP,FILE)               \
  {                                 \
    if((FP=open(FILE))==NULL)       \
    {                               \
      ERROR(FILE);                  \
    }                               \
  }

OPEN(fp,"test.tmp");
```

Durch Präprozessor erzeugter Code:

```
{
  if((fp=open("test.tmp"))==NULL)
  {
    fprintf(stderr,"test.tmp");
  }
};
```


© 2018 Prof. Dr. Stefan Böhmer

Präprozessor

#define-Direktive

Beispiele für potentielle Fehler bei der Verwendung von Makros :

```
#define FORMEL(X)  X * X + 1.0

y = FORMEL(a+1);
y = FORMEL(a++);
y = 0.5*FORMEL(a);
```

```
y = a+1 * a+1 + 1.0;
y = a++ * a++ + 1.0;
y = 0.5 * a * a + 1.0;
```

a = 2:

y = 6 statt 10

y = 5; a = 4

y = 3 statt 2.5

```
#define FORMEL(X)  ((X) * (X) + 1.0)

y = 0.5*FORMEL(a+1);
```

```
y = 0.5*((a+1) * (a+1) + 1.0);
```

y = 5

© 2018 Prof. Dr. Stefan Böhmer

Präprozessor

#if-Direktive

- Bedingte Übersetzung
- Syntax:

```
#if ausdruck1
        text1
#elif ausdruck2
        text2
...
#else
        textN
#endif
```

  Es können beliebig viele (oder keine) **`#elif`**-Direktiven angegeben werden.
  Die **`#else`**-Direktive ist optional

- Der Präprozesor wertet die Ausdrücke nacheinander aus und setzt den Text ein, bei dem der Ausdruck erstmalig von **0** (TRUE) verschieden ist
- Ein Ausdruck muss aus Konstanten bestehen, kann aber C-übliche Operatoren enthalten

© 2018 Prof. Dr. Stefan Böhmer

Präprozessor

Beispiel #if-Direktive

Quelltext:

```
#define VERSION DRAFT

#if VERSION == DEBUG

  printf("Debug-Version");

#elif VERSION == DRAFT

  printf("Draft-Version");

#elif VERSION == TESTED

  printf("Tested");

#else

  printf("Released");

#endif
```

Durch Präprozessor erzeugter Code:

```
printf("Draft-
Version");
```


© 2018 Prof. Dr. Stefan Böhmer

Präprozessor

#if-Direktive

- Mit dem **`defined`**-Operator kann auch geprüft werden, ob eine Konstante existiert (ohne, dass ihr ein Wert zugewiesen wurde). Alternativ kann dazu auch eine **`#ifdef`** oder **`#ifndef`**-Direktive verwendet werden.
- Beispiel:

Quelltext:

```
#define DEBUG

#if defined(DEBUG)
  printf("Debug-Version");
#endif

#ifdef DEBUG
  printf("Debug-Version");
#endif

#ifndef DEBUG
  printf("Released");
#endif
```

Durch Präprozessor erzeugter Code:

```
printf("Debug-Version");

printf("Debug-Version");
```


© 2018 Prof. Dr. Stefan Böhmer

Compiler

Aufruf und Optionen

- Der Compiler übersetzt den Quelltext in Binär-Code
- Hier: Linux-Compiler gcc
- Aufruf (stark vereinfacht):

```
gcc [optionen] infile
```

- Einige wichtige Optionen:

`-E`	Nur Präprozessor
`-c`	Nur übersetzen
`-l name`	Die Bibliothek *name* (File: libname.so oder libname.a) wird eingebunden
`-L dirname`	Fügt *dirname* zur Verzeichnisliste hinzu, in denen der Linker nach einer Bibliothek sucht
`-I dirname`	Fügt *dirname* zur Verzeichnisliste hinzu, in denen der Compiler nach einer Include-Datei sucht
`-D name=wert`	Definiert Präprozessor-Makro (vgl.: #define name wert)
`-o outfile`	Gibt den Namen der Ausgabedatei vor. Falls die Option nicht gesetzt ist, wird der Name durch gcc automatisch vergeben (typisch a.out)

- Beispiel:

```
gcc -c  -D VERSION=DRAFT ping.c pong.c
gcc -o tt pong.o ping.o
```


© 2018 Prof. Dr. Stefan Böhmer

Make

- Make organisiert den Übersetzungsprozess
- Aufruf:

```
make [-f Makefile] [optionen] [target]
```

- Wird beim Aufruf von **make** kein Makefile angegeben, sucht **make** im aktuellen Verzeichnis nach einer Datei namens *Makefile*
- Der Übersetzungsprozess wird durch ein **Makefile** spezifiziert, das Regeln enthält
- Eine Regel besteht aus:
  - Ziel (target): symbolischer Name oder Dateiname
  - Abhängigkeitsliste (dependency): symbolischer Name oder Dateiname

    In der Abhängigkeitsliste darf der Name eines Zieles oder der Name einer existierenden Datei stehen
  - Aktion (action): ausführbares Kommando
- Syntax der Regel:

```
ziel: abhängigkeitsliste
        aktion
```

- Vor der **aktion** muss ein **Tabulatorzeichen** stehen!

© 2018 Prof. Dr. Stefan Böhmer

Make

- **`make`** wertet zuerst die Regel aus, zu der ein passendes Ziel beim **`make`**-Aufruf angegeben wurde. Wird beim Aufruf von **`make`** kein Ziel angegeben, startet **`make`** mit der ersten Regel (Default)
- Bei einer leeren Abhängigkeitsliste wird die Aktion immer ausgeführt und es werden keine weiteren Regeln angewendet

```
$ make
default ausführen

$ make inst
jetzt installieren
```

```
# Makefile

one:
        @echo default ausführen

comp:
        @echo jetzt übersetzen

inst:
        @echo jetzt installieren

clean:
        @echo jetzt aufraeumen
```


© 2018 Prof. Dr. Stefan Böhmer

Make

Falls die Abhängigkeitsliste weitere Ziele enthält, verzweigt **make** zunächst zu den entsprechenden Regeln und wertet diese aus, bevor die eigentliche Aktion ausgeführt wird (Rekursion)

```
$ make all
jetzt übersetzen
jetzt installieren
alles fertig

$ make inst
jetzt installieren
```

```
# Makefile

all: comp inst
        @echo alles fertig

comp:
        @echo jetzt übersetzen

inst:
        @echo jetzt installieren

clean:
        @echo jetzt aufraeumen
```


© 2018 Prof. Dr. Stefan Böhmer

Make

Regeln werden nur dann ausgeführt, wenn bezüglich des Ziels oder der Abhängigkeitsliste ein Aktualisierungs- oder Erstellungsvorgang erforderlich ist

```
$ make clean
alles sauber!

$ make all
jetzt übersetzen
jetzt binden
jetzt installieren
alles fertig

$ make all
jetzt installieren
alles fertig
```

```
# Makefile

all: prog inst
        @echo alles fertig

prog.o: prog.c prog.h
        @echo jetzt übersetzen
        gcc -c prog.c

prog: prog.o
        @echo jetzt binden
        gcc -o prog prog.o

inst:
        @echo jetzt intallieren
        mkdir -p bin
        cp prog bin/prog

clean:
        rm -f prog bin/prog
        @echo alles sauber!
```


© 2018 Prof. Dr. Stefan Böhmer

Agenda

1. Einführung in die Programmiersprache C ☑
2. Gültigkeitsbereiche, komplexe Datentypen ☑
3. Kontrollstrukturen, Ein- und Ausgabe ☑
4. Zeiger, Felder und Zeichenketten ☑
5. Makros, C-Entwicklungswerkzeuge ☑
6. **Dateisystem**
7. Ausgewählte Beispiele (Prozesse, Threads, …)

© 2018 Prof. Dr. Stefan Böhmer

Zugriff auf das Dateisystem

Low Level

- Elementare Funktionen (POSIX)
- unformatiert
- Filedescriptor

POSIX = **P**ortable **O**perating **S**ystem **I**nterface for Uni**X**,
IEEE-Standard für Schnittstelle zwischen Betriebssystem und Applikation

High Level

- Standard-Funktionen (ANSI-C)
- formatiert
- FILE-Zeiger (Stream)

ANSI-C = **A**merican **N**ational **S**tandards **I**nstitute,
Standard für Programmiersprache **C**

© 2018 Prof. Dr. Stefan Böhmer

Zugriff auf das Dateisystem

Low-Level -- Filedescriptor

- Betriebssystemkern: Verwaltet für jeden Prozess eine Tabelle mit geöffneten Dateien
- Filedescriptor = Index des Tabelleneintrages (Integer-Wert)
- Auszug aus **<unistd.h>:**

```
/* Standard file descriptors.  */
#define STDIN_FILENO    0       /* Standard input  */
#define STDOUT_FILENO   1       /* Standard output */
#define STDERR_FILENO   2       /* Standard error output */
```

- Für elementare Zugriffe auf Dateien, Schnittstellen und Geräte wird immer der sogenannte *Filedescriptor* benötigt

© 2018 Prof. Dr. Stefan Böhmer

Dateien öffnen und schließen

open()

- **`open()`** - Öffnet eine Datei
- Syntax:

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

`pathname:`	Name der zu öffnenden Datei	
`flags:`	**`O_RDONLY`**	Datei wird zum Lesen geöffnet
	`O_WRONLY`	Datei wird zum Schreiben geöffnet
	`O_RDWR`	Datei wird zum Lesen und Schreiben geöffnet
	`O_CREAT`	Falls Datei nicht existiert, wird sie erzeugt
	`O_TRUNC`	Falls Datei existiert, wird der Inhalt gelöscht
	`O_APPEND`	Schreibbefehl hängt Inhalt ans Ende der Datei an
	`...`	
`mode:`	**`S_I...`**	Spezifiziert die Zugriffsrechte, wenn Datei erzeugt wird

- Rückgabewert:
  - Filedeskriptor, falls die Datei geöffnet werden konnte
  - -1 im Fehlerfall

© 2018 Prof. Dr. Stefan Böhmer

Dateien öffnen und schließen

close()

- `close()` - Schließt eine geöffnete Datei
- Syntax:

```
int close(int fd);
```

  `fd`: Filedescriptor der Datei

- Rückgabewert:
  0, falls Datei geschlossen werden konnte
  -1 im Fehlerfall

© 2018 Prof. Dr. Stefan Böhmer

Dateien lesen und schreiben

read()

- **`read()`** - Liest Daten aus einer geöffneten Datei
- Syntax:

```
ssize_t read(int fd, void *buf, size_t count);
```

  `fd:` Filedeskriptor der Datei

  `buf:` Zeiger auf Datenbereich, in dem gelesene Daten abgelegt werden

  `count:` Größe des Datenbereiches in Bytes

- Rückgabewert:

  Anzahl der gelesenen Bytes

  -1 im Fehlerfall

© 2018 Prof. Dr. Stefan Böhmer

Dateien lesen und schreiben

write()

- **`write()`** - Schreibt Daten in eine geöffnete Datei
- Syntax:

```
ssize_t write(int fd, const void *buf, size_t count);
```

  `fd`: Filedeskriptor der Datei

  `buf`: Zeiger auf Datenbereich, der in die Datei geschrieben werden soll

  `count`: Größe des Datenbereiches in Bytes

- Rückgabewert:

  Anzahl der geschriebenen Bytes

  -1 im Fehlerfall

© 2018 Prof. Dr. Stefan Böhmer

Dateien lesen und schreiben

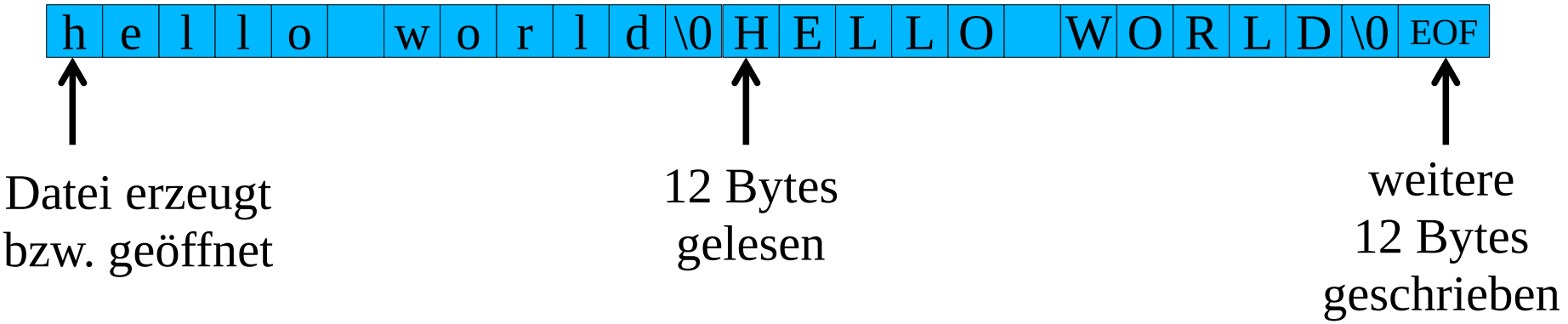
Beispiel

```
/* open.c */

int main(void)
{
  int fd;
  int i;
  char txt[] = "hello world";

  fd = open("open.tmp", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);
  write(fd, txt, sizeof(txt));
  close(fd);

  fd = open("open.tmp", O_RDWR);
  read(fd, txt, sizeof(txt));
  for(i=0; i<sizeof(txt); i++)
  {
    txt[i] = toupper(txt[i]);
  }
  write(fd, txt, sizeof(txt));
  close(fd);
}
```


© 2018 Prof. Dr. Stefan Böhmer

Schreib-/Leseposition

- Jeder geöffnete Datei besitzt einen Schreib-/Lesezeiger
- Schreib- und Leseoperationen beginnen ab aktueller Leseposition
- Schreib-/Leseposition wird automatisch um Anzahl geschriebener bzw. gelesener Bytes verschoben
- Nach dem Öffnen einer Datei befindet sich der Schreib-/Lesezeiger an Position 0
  Ausnahme: Datei wurde mit **`O_APPEND`** geöffnet
- Aktuelle Schreib-/Leseposition kann mit **`lseek()`** abgefragt bzw. geändert werden



© 2018 Prof. Dr. Stefan Böhmer

Beispiel für weitere Funktionen

Datei-Multiplexing mit select()

- **`select()`** - Prüft, ob Dateien zum Lesen oder Schreiben bereit sind
- Syntax:

```
int select(int n,fd_set *readfds,fd_set *writefds, \
                fd_set *exceptfds, struct timeval *timeout);
```

`n:`	Größter Filedeskriptor + 1
`readfds:`	Filedeskriptor-Liste der zu lesenden Dateien
`writefds`:	Filedeskriptor-Liste der zu schreibenden Dateien
`exceptfds:`	Filedeskriptor-Liste der Dateien mit Ausnahmezustand
`timeout:`	Spezifiziert ein Timeout

- Rückgabewert:
  Anzahl der schreib-/lesebereiten Dateien
  -1 im Fehlerfall

© 2018 Prof. Dr. Stefan Böhmer

Beispiel für weitere Funktionen

Datei-Multiplexing mit select()

- Die Liste **`set`** der zu überwachenden Filedeskriptoren kann mit folgenden Makros bearbeitet werden:

- Syntax:

```
FD_ZERO (         fd_set *set); // Löscht die gesamte Liste
FD_SET  (int fd,  fd_set *set); // Fügt fd in die Liste ein
FD_CLR  (int fd,  fd_set *set); // Löscht fd aus der Liste
FD_ISSET(int fd,  fd_set *set); // Prüft, ob fd in der Liste
                                // enthalten ist
```

- Vor jedem Aufruf von **`select()`** muss die Liste mit **`FD_ZERO`** gelöscht und neu befüllt werden.

- Nach dem Aufruf von **`select()`** enthalten die Listen nur noch die Filedeskriptoren, die zum Lesen bzw. zum Schreiben bereit sind.

© 2018 Prof. Dr. Stefan Böhmer

Beispiel für weitere Funktionen

Datei-Multiplexing mit select()

- Der Parameter **`timeout`** ist ein Zeiger auf eine Struktur, mit der festgelegt wird, wie lange **`select()`** auf eine schreib-/lesebereite Datei wartet.

```
struct timeval
{
        long tv_sec;        /* Seconds */
        long tv_usec;       /* Microseconds */
};
```

- Mit

```
timeout.tv_sec = 0;
timeout.tv_usec = 0;
```

  kehrt **`select()`** sofort zurück

- Wird anstelle von **`timeout`** der Wert **`NULL`** an übergeben, blockiert **`select()`** solange, bis eine Datei schreib-/lesebereit ist.

- Die Struktur **`timeout`** muss vor jedem Aufruf von **`select()`** neu befüllt werden!

© 2018 Prof. Dr. Stefan Böhmer

Zugriff auf das Dateisystem

High-Level -- Filedescriptor

- Bestandteil des C-Standards
- Verwendung eines FILE-Zeigers (= Stream) anstelle des Filedeskriptors
- Auszug aus **<stdio.h>**:

```
/* Standard streams  */
extern struct FILE *stdin;       // Standard input stream
extern struct FILE *stdout; // Standard output stream
extern struct FILE *stderr; // Standard error output stream
```

- Die Daten werden gepuffert und erst dann geschrieben, wenn der Puffer voll ist oder die Datei geschlossen wird

© 2018 Prof. Dr. Stefan Böhmer

High-Level-Dateioperationen

fopen()

- **`fopen()`** - Öffnet eine Datei
- Syntax:

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

  `pathname`: Name der zu öffnenden Datei

  `mode`: Zugriffsmodus

- Rückgabewert:

  Stream (Zeiger auf FILE-Struktur), wenn Datei geöffnet werden konnte

  NULL im Fehlerfall

- Bei einer neu erzeugten Datei werden die Dateirechte auf **`rw-rw-rw-`** (0666) gesetzt

© 2018 Prof. Dr. Stefan Böhmer

High-Level-Dateioperationen

fopen() - Zugriffsmodus

- Die Zugriffsart wird mit dem String **`mode`** gesetzt:

mode	Lesen/Schreiben	Position
"r"	nur Lesen	Anfang
"r+"	Lesen/Schreiben	Anfang
"w"	nur Schreiben	Anfang
"w+"	Lesen/Schreiben	Anfang
"a"	nur Schreiben	Ende
"a+"	Lesen/Schreiben	Ende

- Falls eine Datei nur zum Schreiben geöffnet wird, wird der Inhalt ab der aktuellen Schreibposition gelöscht
- Eine neue Datei kann nur mit “w“, “w+“, “a“ oder “a+“ erzeugt werden
- Der mode-String kann zusätzlich das Zeichen 'b' enthalten. Die Datei wird dann als Binärdatei (sonst: Textdatei) geöffnet (Unterscheidung gilt nicht für Linux).

© 2018 Prof. Dr. Stefan Böhmer

High-Level-Dateioperationen

fclose()

- **`fclose()`** - Schließt eine geöffnete Datei
- Syntax:

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

  `stream`: Datei-Stream

- Rückgabewert:

  0, falls Datei geschlossen werden konnte

  EOF im Fehlerfall

© 2018 Prof. Dr. Stefan Böhmer

High-Level-Dateioperationen

fscanf()

- **`fscanf()`** - Liest aus einem geöffneten Stream
- Syntax:

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

  `stream`: Datei-Stream

  `format`: Format-String

- Rückgabewert:

  Anzahl der erfolgreich eingelesenen Werte (Items) oder EOF

- Die Funktion **`fscanf()`** verhält sich analog zur Funktion **`scanf()`**

© 2018 Prof. Dr. Stefan Böhmer

High-Level-Dateioperationen

fprintf()

- **`fprintf()`** - Schreibt in einen geöffneten Stream
- Syntax:

```
int fprintf(FILE *stream, const char *format, ...);
```

  `stream`: Datei-Stream

  `format`: Format-String

- Rückgabewert:

  Anzahl der erfolgreich geschriebenen Zeichen

  < 0 im Fehlerfall

- Die Funktion **`fprintf()`** verhält sich analog zur Funktion **`printf()`**

© 2018 Prof. Dr. Stefan Böhmer

High-Level-Dateioperationen

weitere Funktionen

- **`fflush()`** - Schreibt den Puffer-Inhalt physikalisch in die Datei
- **`fwrite()`** - Schreibt in einen (binären) Stream
- **`fread()`** - Liest aus einem (binären) Stream
- **`fseek()`** - Setzt Lese-/Schreibposition
- **`ftell()`** - Liefert aktuelle Lese-/Schreibposition
- **`fgets()`** - Liest eine Zeile aus einem Stream
- **`fputs()`** - Schreibt einen String in einen Stream

© 2018 Prof. Dr. Stefan Böhmer

High-Level-Dateioperationen

Beispiel

```
/* fopen.c */

int main(void)
{
  FILE *fp;
  int  i;
  char txt[16];

  fp = fopen("fopen.tmp", "w");
  fprintf(fp, "hello world\n");
  fclose(fp);

  fp = fopen("fopen.tmp", "r+");
  fgets(txt, sizeof(txt), fp);
  for(i=0;i<sizeof(txt);i++)
  {
    txt[i] = toupper(txt[i]);
  }
  fputs(txt, fp);
  fclose(fp);
}
```


© 2018 Prof. Dr. Stefan Böhmer

Verzeichnisoperationen

opendir()

- **`opendir()`** - Öffnet ein Verzeichnis
- Syntax:

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

  `name`: Verzeichnisname
- Rückgabewert:

  Verzeichnis-Stream (Zeiger auf DIR-Struktur), wenn das Verzeichnis geöffnet werden konnte

  NULL im Fehlerfall

© 2018 Prof. Dr. Stefan Böhmer

Verzeichnisoperationen

closedir()

- **`closedir()`** - Schließt ein geöffnetes Verzeichnis
- Syntax:

```
int closedir(DIR *dir);
```

  `dir`: Verzeichnis-Stream

- Rückgabewert:
  0, falls Verzeichnis geschlossen werden konnte
  -1 im Fehlerfall

© 2018 Prof. Dr. Stefan Böhmer

Verzeichnisoperationen

readdir()

- **`readdir()`** - Liest einen Eintrag aus einem geöffneten Verzeichnis
- Syntax:

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

  `dir`: Verzeichnisstream

- Rückgabewert:
  Zeiger auf eine Struktur **`dirent`**
  NULL im Fehlerfall
- Bei jedem Aufruf wird ein Zeiger auf den nächsten Verzeichniseintrag zurückgeliefert
  Daten des vorhergehenden Aufrufs werden überschrieben
- Die Struktur **`dirent`** enthält unter anderem die Elemente

```
unsigned char d_type
char    d_name[256]
```


© 2018 Prof. Dr. Stefan Böhmer

Verzeichnisoperationen

stat()

- **`stat()`** - Ermittelt den Status (Typ, Zugriffsrechte, etc.) eines Verzeichniseintrages
- Syntax:

```
int stat(const char *name, struct stat *buf);
```

  `name`: Name des Verzeichniseintrages

  `buf`: Zeiger auf eine Struktur, in der Ergebnisse abgelegt werden

- Rückgabewert:

  0, falls Status ermittelt werden konnte

  -1 im Fehlerfall

© 2018 Prof. Dr. Stefan Böhmer

Verzeichnisoperationen

stat() - Strukturelemente

Die Struktur **`stat`** enthält unter anderem die Elemente:

```
unsigned long      st_ino;    // File serial number
unsigned int       st_mode;   // File mode
unsigned short int st_nlink;  // Link count
unsigned int       st_uid;    // User ID of the file's owner
unsigned int       st_gid;    // Group ID of the file's group
signed long        st_size;   // Size of file, in bytes
signed long        st_blksize;// Optimal block size for I/O
signed long        st_blocks; // Number 512-byte blocks allocated
struct timespec    st_atim;   // Time of last access
struct timespec    st_mtim;   // Time of last modification
struct timespec    st_ctim;   // Time of last status change
```


© 2018 Prof. Dr. Stefan Böhmer

Verzeichnisoperationen

stat() - Typ des Verzeichniseintrages

Das Element **`st_mode`** der Struktur **`stat`** enthält unter anderem den Typ des Verzeichniseintrages. Dieser kann mit folgenden Makros abgefragt werden:

Macro	Typ
`S_ISREG(m)`	Regular File
`S_ISDIR(m)`	Verzeichnis
`S_ISCHR(m)`	Character-Device
`S_ISBLK(m)`	Block-Device
`S_ISFIFO(m)`	Fifo (named Pipe)
`S_ISLNK(m)`	Link
`S_ISSOCK(m)`	Socket

© 2018 Prof. Dr. Stefan Böhmer

Verzeichnisoperationen

stat() - Zugriffsrechte

Die Zugriffsrechte sind im Element **`st_mode`** der Struktur **`stat`** als Bitmuster enthalten:

Name	Oktal-Wert	Flag
`S_IRWXU`	**`0700`**	mask for file owner permissions
`S_IRUSR`	**`0400`**	owner has read permission
`S_IWUSR`	**`0200`**	owner has write permission
`S_IXUSR`	**`0100`**	owner has execute permission
`S_IRWXG`	**`0070`**	mask for group permissions
`S_IRGRP`	**`0040`**	group has read permission
`S_IWGRP`	**`0020`**	group has write permission
`S_IXGRP`	**`0010`**	group has execute permission
`S_IRWXO`	**`0007`**	mask for permissions for others (not in group)
`S_IROTH`	**`0004`**	others have read permission
`S_IWOTH`	**`0002`**	others have write permisson
`S_IXOTH`	**`0001`**	others have execute permission

© 2018 Prof. Dr. Stefan Böhmer

Verzeichnisoperationen

weitere Funktionen

- **`mkdir()`** - Legt ein Verzeichnis an
- **`chdir()`** - Wechsel das aktuelle Verzeichnis
- **`rmdir()`** - Löscht ein leeres Verzeichnis
- **`remove()`** - Löscht einen Verzeichniseintrag (Datei oder leeres Verzeichnis)

© 2018 Prof. Dr. Stefan Böhmer

Verzeichnisoperationen

Beispiel

```
/* opendir.c */

int main(void)
{
  DIR           *dp;
  struct dirent *eintrag;
  struct stat    eigenschaften;

  dp = opendir(".");

  while( (eintrag = readdir(dp)) != NULL )
  {
    stat(eintrag->d_name, &eigenschaften);

    printf("%s %o %s\n",
           S_ISDIR(eigenschaften.st_mode)?"DIR":"---",
           eigenschaften.st_mode & (S_IRWXU|S_IRWXG|S_IRWXO),
           eintrag->d_name);
  }
  closedir(dp);
}
```


© 2018 Prof. Dr. Stefan Böhmer