FH | JOANNEUM
University of Applied Sciences

Systemnahe Programmierung
SS 2024

Unit 5

Helmut Lindner

Präprozessor

Vor dem eigentlichen Übersetzungsvorgang wird er Präprozessor ausgeführt, der alle Präprozessor-Anweisungen auswertet und textuell ersetzt.

Alle Anweisungen des Präprozessors beginnen mit einem „#“

Includes

`#include` fügt den Inhalt einer Datei in den Sourcecode ein.

```
#include <stdio.h>
```
-> sucht in vordefinierten Verzeichnissen (/usr/include ...)

```
#include "myheader.h"
```
-> sucht in projektspezifischen Verzeichnissen

Präprozessor

Präprozessor-Symbole - define

`#define` definiert ein Symbol das durch einen Text ersetzt werden soll (Makro).

Syntax: `#define <Symbol> <Ersetzungstext>`

Das Symbol kann im Sourcecode verwendet werden und wird überall wo es vorkommt inline ersetzt.

```
#define PI 3.14159  -> Verwendung: u = 2* r * PI;
```

Ein Symbol kann Parameter ähnlich wie in Funktionen haben:

```
#define MULT(a,b) ((a)*(b))
int prod = MULT(2,4);
```

Wird ersetzt zu: `int prod = ((2)*(4));`

Parameter in Klammern garantieren die Reihenfolge der Operatoren!

```
#define MULT(a,b) (a * b)
int prod = MULT(1+2,5+3);
```

Wird ersetzt zu: `int prod = (1+2*5+3);` **-> Falsch**

Präprozessor

Conditional Compilation

Mit den Präprozessor-Conditionals

```
#if, #ifdef, #ifndef, #else, #elif, #endif
```

kann kontrolliert werden, welcher Code compiliert wird.
Wenn die entsprechende Bedingung zutrifft, dann wird der entsprechende Bereich in den Quelltext übernommen, sonst weggelassen.

Beispiel:

```
#define LINUX_X86_64  // Symbol definieren

....

#ifdef LINUX_X86_64
    long variable;
#else
    int variable;
#endif
```

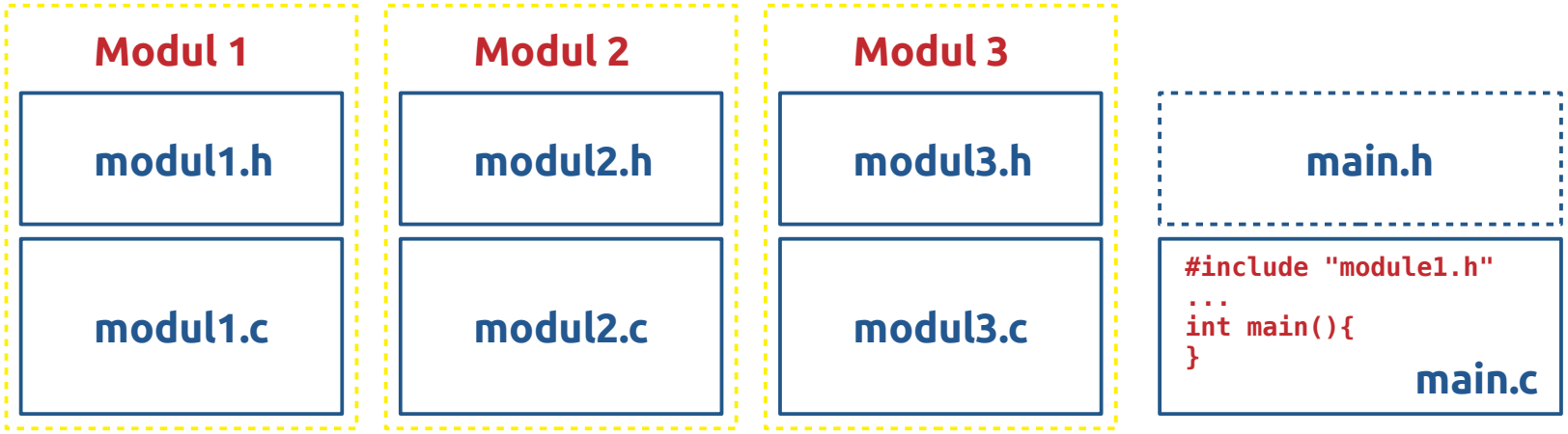
Dieser Text wird in die Quelldatei eingefügt

Module

Projekte mit mehreren Quell-Dateien

C-Programme können in „Module“ (oder Compilation Units) aufgeteilt werden.

Dabei entsprechen die Header-Dateien (.h) dem Interface und die Source-Dateien (.c) der Implementierung.



Alle C-Dateien compilieren und linken:

```
cc modul1.c modul2.c modul3.c main.c -o modules
```

Module

Header Dateien

Die Headerdatei sollte das enthalten, was nach außen exportiert wird (Makros, Deklaration von globalen Variablen und Funktionen).

Die zugehörige C-Datei enthält die Definition der globalen Variablen und der Funktionen.

```
<modulname>.h
<modulname>.c  -> enthält #include "modulname.h"
```

Verhindern der mehrfachen Inkludierung einer Headerdatei (durch verschachtelte Includes)

Anfang der Headerdatei:

```
#ifndef _MODULNAME_H
#define _MODULNAME_H  -> stellt sicher, dass modulname.h
                         nur 1x als Text expandiert wird

#endif /* _MODULNAME_H  */
```

Externe Variable

Externe Variable

Ist eine Variable mit extern deklariert, so wird angenommen, dass sie in einer anderen Datei als globale Variable und nicht als `static` definiert wurde.

Beispiel:

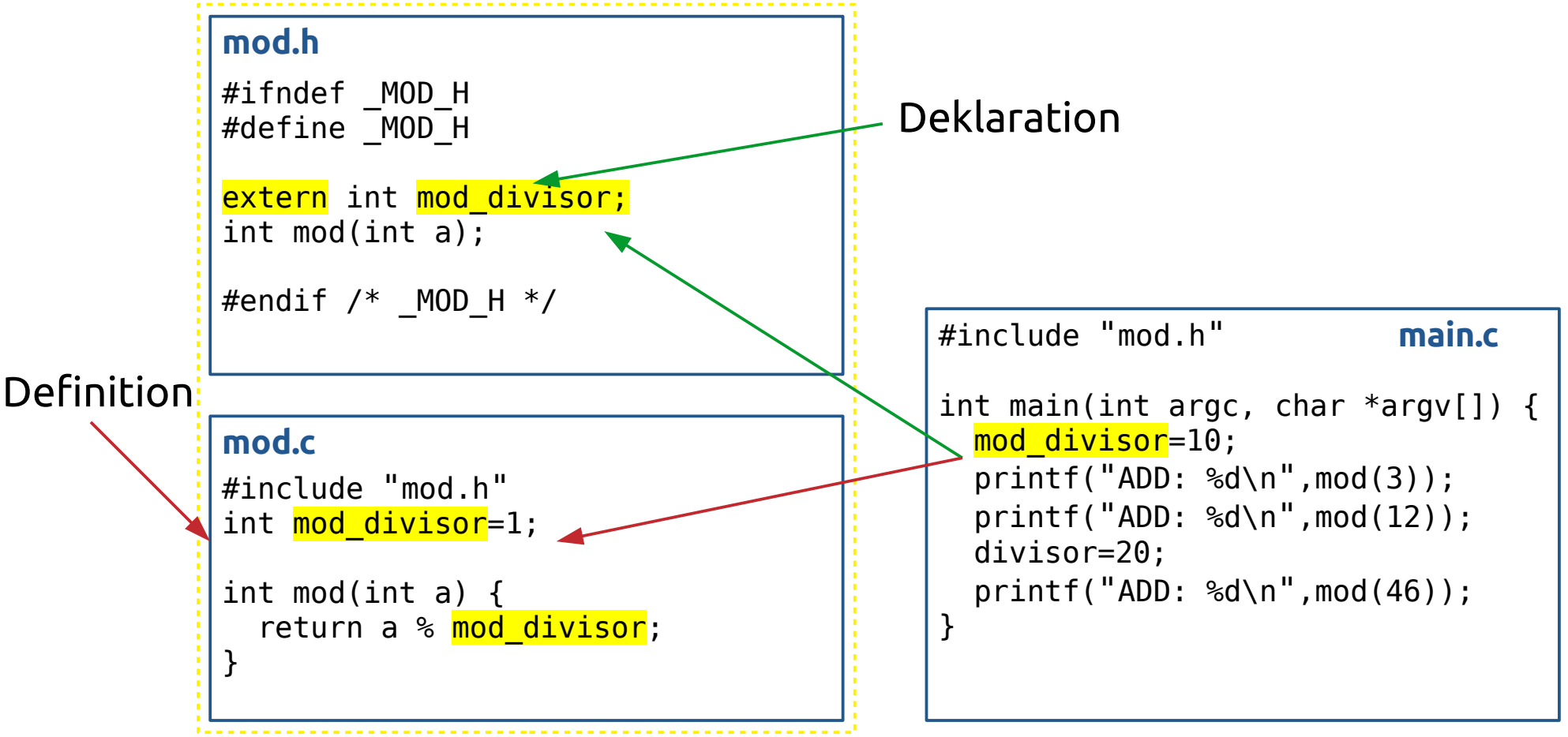
```
datei.c: int globalVar=3; // Definition (nur mit Zuweisung!)

datei.h: extern int globalVar; // Deklaration als extern
```

Wird immer in einer Headerdatei (.h) angegeben, damit die Deklaration in anderen Dateien verwendet (inkludiert) werden kann.

Die globale Variable selbst nicht in einer Headerdatei deklarieren sondern in der zugehörigen .c Datei!

Externe Variable

Beispiel



Projektstruktur

Template

Das vorgestellte Template dient als allgemeine Basis für C-Projekte.

Verzeichnisse:

`src`	… alle C Dateien (Sourcecode)
`include`	… Header Dateien
`bin`	… linked Executables
`build`	… compilierte Objektdateien
`test`	… Testdaten und Scripts

Dateien:

- Makefile
- test.cfg (Konfigurationsdatei für Beispielcode)
- .gitlab-ci.yml Gitlab Pipeline Definition
- .gitignore

Makefile

Das Template verwendet `make` um die Projektbestandteile zu compilieren, Tests durchzuführen usw. Die Regeln nach den `make` vorgeht sind im sog. „Makefile“ festgelegt.

Makefile

Kommentare

```
# compile .c files
```

Variablendefinitionen

```
INC_DIR := ./include
CFLAGS := -I$(INC_DIR) -Wall -c
```

Regeln

```
<Ziel>: <Bedingungen>
<tab>Kommando
        ...

$(BUILD_DIR)/%.o: $(SRC_DIR)/%.c*
        $(CC) $(CFLAGS) -o $@ $<
```

siehe auch https://makefiletutorial.com/

Low-level I/O

Übersicht

Die Standard I/O Funktionen zeichnen sich durch vielfältige Möglichkeiten aus (zeichenweise, zeilenweise und blockweise Ein- Ausgabe).
Sie basieren auf den systemabhängigen Low-level Funktionen, die meistens für ungebufferte Byte-Streams verwendet werden.

Standard I/O Funktionen interpretieren bei Bedarf Teile des Byte-Streams, Low-level Funktionen nicht.

Low-level I/O Funktionen sind systemspezifisch, d.h. die Portabilität ist nicht immer gewährleistet (im Gegensatz zu Standard I/O)

Low-level I/O

Dateien öffnen und schließen

Der low-level Call `open()` liefert einen Dateideskriptor vom Typ `int` zurück.
Im Fehlerfall wird -1 zurückgegeben und `errno` entsprechend gesetzt.
Der Dateideskriptor beschreibt einen Stream der für alle I/O Operationen in Linux verwendet werden kann(Dateien, Geräte, FIFO, Netzwerk usw.)

```
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
int open(const char *pfadname, int flags);
int open(const char *pfadname, int flags, mode_t access);
int close(int fd);
```

`flags`: Access Mode, Creation Flags, Status Flags
Access Mode kann nur einer gesetzt werden, die anderen Flags können mit bitweisem ODER (|) verknüpft werden.

`access`:
Die Zugriffsrechte können für user, group und others gesetzt werden, wenn die Datei neu angelegt wird .

Low-level I/O

open() - flags - Access Mode

Flag	Beschreibung
O_RDONLY	Öffnet die Datei zum Lesen
O_WRONLY	Öffnet die Datei zum Schreiben
O_RDWR	Öffnet die Datei zum Lesen und Schreiben

Low-level I/O

open() - flags - Creation Flags

Flag	Beschreibung
O_CREAT	Wenn die Datei nicht existiert, wird sie angelegt. Owner wird die effektive Userid des Prozesses.
O_EXCL	Wird dieses Flag zusammen mit O_CREAT verwendet, kann die Datei nicht geöffnet werden, wenn sie bereits existiert. Mit diesem Flag können parallele Schreibzugriffe auf eine Datei verhindert werden.
O_NOCTTY	Wenn der Pfad auf ein Terminal-Device verweist, dann wird dieses Terminal nicht das neue Controlling-Terminal.
O_TRUNC	Wenn die Datei existiert und sie wird zum Schreiben geöffnet, dann wird sie auf Länge 0 gekürzt. Für FIFO Dateien wird dieses Flag ignoriert.

Low-level I/O

open() -flags - Status Flags (Auszug)

Flag	Beschreibung
O_DIRECTORY	If pathname is not a directory, cause the open to fail. This flag is Linux-specific
O_APPEND	Die Datei wird zum Schreiben am Ende geöffnet. Der Schreib-/Lesezeiger wird vor jeder Schreiboperation auf die Endposition gesetzt.
O_NONBLOCK	Falls der Pfadname der Name eines FIFO oder einer Gerätedatei ist, wird der Prozess beim Öffnen und bei nachfolgenden I/O-Operationen nicht blockiert.
O_SYNC	Jeder Schreibvorgang wird synchron ausgeführt, es wird gewartet, bis die Operation abgeschlossen ist.

Low-level I/O

open() - access - Zugriffsrechte

```
S_IRWXU  00700 user (file owner) has read, write, and execute permission
S_IRUSR  00400 user has read permission
S_IWUSR  00200 user has write permission
S_IXUSR  00100 user has execute permission

S_IRWXG  00070 group has read, write, and execute permission
S_IRGRP  00040 group has read permission
S_IWGRP  00020 group has write permission
S_IXGRP  00010 group has execute permission

S_IRWXO  00007 others have read, write, and execute permission
S_IROTH  00004 others have read permission
S_IWOTH  00002 others have write permission
S_IXOTH  00001 others have execute permission

S_ISUID  0004000 set-user-ID bit
S_ISGID  0002000 set-group-ID bit (see inode(7)).
S_ISVTX  0001000 sticky bit (see inode(7)).
```

Low-level I/O

Vergleich fopen - open

fopen() Modus	open() Modus
r	O_RDONLY
w	O_WRONLY | O_CREAT | O_TRUNC
a	O_WRONLY | O_CREAT | O_APPEND
r+	O_RDWR
w+	O_RDWR | O_CREAT | O_TRUNC
a+	O_RDWR | O_CREAT | O_APPEND

Low-level I/O

Beispiele

Eine Datei zum Lesen öffnen:

```
int fd = open(pfad, O_RDONLY);
```

Eine Datei exklusiv zum Lesen und Schreiben öffnen, nur der User hat Lese- und Schreibrechte falls sie angelegt wird:

```
int fd = open(pfad, O_RDWR | O_EXCL | O_CREAT, S_IRUSR | S_IWUSR);
if ( fd < 0 ) {
  errorhandling ...
}
```

Der Access-Mode muss angegeben werden, wenn O_CREAT verwendet wird.

Low-level I/O

Lesen und schreiben

Von einem Stream lesen:

```
#include <unistd.h>
int read(int fd, const void *buffer, size_t count);
```

`read()` liest `count` Bytes vom Stream in den Buffer. Die Anzahl der tatsächlich gelesenen Bytes wird zurückgegeben, ein Rückgabewert von 0 bedeutet EOF.

Auf einen Stream schreiben:

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

`write()` schreibt `count` Bytes aus dem Buffer auf den Stream. Die Anzahl der tatsächlich geschriebenen Bytes wird zurückgegeben.

Beide Funktionen geben im Fehlerfall -1 zurück und setzen `errno` entsprechend.

Low-level I/O

Schreib- Leseposition setzen

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

off_t lseek(int fd, off_t offset, int mode);
```

mode	
SEEK_SET	Der Schreib-/Lesezeiger wird vom Anfang der Datei um `offset` Bytes versetzt. `offset` darf nicht negativ sein.
SEEK_CUR	Der Schreib-/Lesezeiger wird relativ zur aktuellen Position um `offset` Bytes versetzt. `offset` kann positiv oder negativ sein (vorwärts oder rückwärts suchen).
SEEK_END	Der Schreib-/Lesezeiger wird vom Ende der Datei ausgehend um `offset` Bytes versetzt. `offset` darf positiv oder negativ sein (kann über das Ende der Datei hinaus versetzt werden!).

Low-level I/O

Standardstreams

Die Filedeskriptoren für die Standardstreams sind vordefiniert:

```
#include <unistd.h>

    STDIN_FILENO
    STDOUT_FILENO
    STDERR_FILENO
```

Beispiel:

```
      write(STDERR_FILENO, buffer, size);
```

Diese Streams sind bereits beim Start des Programms geöffnet -> kein `open()` notwendig!

Kommandozeilen-Argumente

Einem Programm werden beim Start die Kommandozeilen-Argumente in der Funktion `main` übergeben:

```
int main(int argc, char *argv[])
```

Anzahl der Argumente:

```
int argc (>=1)
```

Liste der Argumente als Strings:

```
char *argv[]
```

Wobei: `argv[0]` ist immer der Name des Programms mit Pfad ist.

Kommandozeilen-Argumente

Beispiel

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

int main(int argc, char *argv[]) {

   for(int i=0; i < argc; i++) {
      printf("argv[%d] = %s\n", i, argv[i]);
   }
   return EXIT_SUCCESS;
}
```

`./printargs test.dat 4711 -r`

```
argv[0] = ./printargs
argv[1] = test.dat
argv[2] = 4711
argv[3] = -r
```

Low-level I/O

Übung 1a

Schreiben sie ein Programm, das den Inhalt einer Textdatei auf `stderr` ausgibt. Verwenden sie das vorgestellte Template dazu.

Das Programm nimmt den Namen der Datei als Kommandozeilenparameter entgegen. Der Aufruf hat die Form:

```
./a.out <dateiname>
```

Verwenden sie für das Lesen der Datei und die Ausgabe auf `stderr` ausschließlich Low-Level I/O Funktionen.

Verwenden sie zum Einlesen der Datei die Funktion

```
int read(int fd, const void *buffer, size_t count);
```

Zur Ausgabe verwenden sie

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

Den Buffer können sie als Array implementieren.

Fehlerbehandlung

errno und perror()

Ist bei einer Funktion der Standardbibliothek ein Fehler aufgetreten, so wird die globale Variable `errno` gesetzt. `errno` wird beim Start des Programms auf 0 gesetzt.

```
#include <errno.h>
```
`ESUCCESS` ... kein Fehler
`ENOENT` ... Datei/Verzeichnis existiert nicht
...

`errno` ist nur direkt nach dem Funktionsaufruf gültig!
Aufrufe der Standardbibliothek ändern den Wert nur dann, wenn sie ihn auf einen neuen Fehlerwert setzen. Also sollte man, wenn man `errno` verwendet und ein Fehler aufgetreten ist, `errno` wieder auf 0 setzen.

Fehlermeldung anzeigen:

```
void perror (const char *message); // message wird vorangestellt
```
ist nur sinnvoll wenn `errno` gesetzt ist (`errno != ESUCCESS`),
sonst eine eigene Fehlermeldung mit `fprintf()` auf `stderr` ausgeben.

Fehlerbehandlung

Exitcodes beim Beenden des Programms

Konvention: Ein Programm gibt beim Beenden den Exitcode 0 zurück wenn keine Fehler aufgetreten sind, sonst >0.
Damit wird sichergestellt, dass die Shell darauf reagieren kann (z.B.: in Shell-Scripts).

Dafür sind folgende Konstante definiert:

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

Der Exitcode des Programms ist der Returnvalue der `main()` Funktion.

Um das Programm sofort zu beenden kann

```
#include <stdlib.h>
void exit(int status);
```

verwendet werden.

Fehlerbehandlung

Beispiel

Eine Datei zum lesen öffnen, falls sie nicht existiert diese anlegen (anstelle von “w+“)

```
fp = fopen("config.dat", "r");

if(errno == ENOENT) { // -> Datei existiert nicht
  errno = ESUCCESS;   // reset errno

  fp = fopen("config.dat", "w");
  if ( fp == NULL) {
    perror("Konfigurationsdatei");
    exit(EXIT_FAILURE);
  }
}
...
```

Low-level I/O

Übung 1b

Ergänzen sie das Programm aus Übung 1a um eine Fehlerbehandlung beim Dateihandling und prüfen sie das Vorhandensein des Commandline-Arguments.

Im Fehlerfall geben sie eine Meldung aus und beenden das Programm mit einem Fehlerstatus.

Umgebungsvariable

Shell Environment

Umgebungsvariable definieren:

```
$ export LINES=100
```

Abfragen:

```
$ env
```

getenv()

```
#include <stdlib.h>
char *getenv(const char *name);
```

Beispiel:

```
const char* env = getenv("PAGE_SIZE");
if (env == NULL) { // Variable nicht definiert
  ...
}
```

Externe Libraries

Externe Libraries werden mit dem Linker eingebunden. Dazu muss das Compiler-Flag `-l<Library Name>` angegeben werden.

libconfig

Wir verwenden libconfig, um Werte aus einer Konfigurationsdatei einzulesen.

Die Einträge in der Konfigurationsdatei haben folgende Syntax:

<Variable> = <Wert>

Das Paket mit libconfig installieren:

```
sudo apt install libconfig-dev
```

Das Manual zu libconfig finden sie hier:
https://hyperrealm.github.io/libconfig/libconfig_manual.html

Compilerflag um libconfig einzubinden: `-lconfig`

libconfig

Wichtige Funktionen der libconfig:

```
#include <libconfig.h>
config_t config;
```

Initialisieren der Bibliothek:

```
void config_init(config_t  *config)
```

Konfigurationsdatei einlesen:

```
int config_read_file(config_t *config, const char *filename)
```

Integerwert suchen und auslesen (path = Variablenname):

```
int config_lookup_int(const config_t *config, const char *path,
                      int *value)
```

String suchen und auslesen (path = Variablenname):

```
int config_lookup_string(const config_t *config, const char *path,
                         const char **value) → Kopieren wg. destroy()!
```

Speicher usw. wieder freigeben:

```
void config_destroy(config_t *config)
```

libconfig

Beispiel

```
#include <libconfig.h>

config_t config;
char *strVal;
char topDir[1024];
int port;

config_init(&config);

if (config_read_file(&config, "test.cfg") == CONFIG_FALSE) {
  perror("libconfig");
  exit(EXIT_FAILURE);
}

if (config_lookup_string(&config, "top_dir", &strVal) == CONFIG_FALSE)
  strcpy(topDir, "./top");
else
  strcpy(topDir, strVal);

if (config_lookup_int(&config, "port", &port) == CONFIG_FALSE)
  port = 17000;

config_destroy(&config);
```

libconfig

Übung 2

Erweitern sie das Programm aus der letzten Übung, so dass es die Werte `"num_bytes"` (Typ integer) und `"total"` (Typ String) aus einer Konfigurationsdatei einliest.

Die Ausgabe soll auf `num_bytes` Zeichen beschränkt werden, falls die Variable `num_bytes` in der Konfigurationsdatei enthalten und >0 ist.

Falls die Variable `total` (max. Länge 1024) in der Konfigurationsdatei enthalten ist, soll am Ende der Ausgabe eine Zusammenfassung wie folgt ausgegeben werden:

```
<inhalt von total>\n
Datei: <Dateiname>
Ausgegebene Zeichen: <anzahl zeichen>
```

Lesen sie den Pfad der Konfigurationsdatei aus der Umgebungsvariable `"CONFIG_FILE"` ein. Wenn diese nicht existiert, soll das Programm eine Meldung ausgeben und abbrechen.
Beispiel: Variable definieren in Shell `export CONFIG_FILE="./test.cfg"`