

# Übungen zu Systemnahe Programmierung in C (SPiC) – Wintersemester 2022

---

## Übung 7

Phillip Raffeck  
Maximilian Ott

Lehrstuhl für Informatik 4  
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Verteilte Systeme  
und Betriebssysteme



FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

# Fehlerbehandlung

---



- Fehler können aus unterschiedlichsten Gründen auftreten
  - Systemressourcen erschöpft
    - ⇒ `malloc(3)` schlägt fehl
  - Fehlerhafte Benutzereingaben (z.B. nicht existierende Datei)
    - ⇒ `fopen(3)` schlägt fehl
  - Vorübergehende Fehler (z.B. nicht erreichbarer Server)
    - ⇒ `connect(2)` schlägt fehl



- Gute Software:
  - Erkennt Fehler
  - Führt angebrachte Behandlung durch
  - Gibt aussagekräftige Fehlermeldung aus
  
- Kann ein Programm trotz Fehler sinnvoll weiterlaufen?

**Beispiel 1:** Ermittlung des Hostnamens zu einer IP-Adresse, um beides in eine Logdatei einzutragen

⇒ IP-Adresse ins Log eintragen, Programm läuft weiter

**Beispiel 2:** Öffnen einer zu kopierenden Datei schlägt fehl

⇒ Fehlerbehandlung: Kopieren nicht möglich, Programm beenden

⇒ Oder den Kopiervorgang bei der nächsten Datei fortsetzen

⇒ Entscheidung liegt beim Softwareentwickler



- Fehler treten häufig in `libc` Funktionen auf
  - Erkennbar i.d.R. am Rückgabewert (Manpage)
  - Fehlerüberprüfung essentiell
  
- Fehlerursache steht meist in `errno` (globale Variable)
  - Einbinden durch `errno.h`
  - Fehlercodes sind  $> 0$
  - Fehlercode für jeden möglichen Fehler (siehe `errno(3)`)
  
- `errno` nur interpretieren, wenn Fehler signalisiert wurde
  - Funktionen dürfen `errno` beliebig verändern
    - ⇒ `errno` kann auch im Erfolgsfall geändert worden sein



- Fehlercodes ausgeben:
  - `perror(3)`: Ausgabe auf `stderr`
  - `strerror(3)`: Umwandeln in Fehlermeldung (String)

## Beispiel:

```
01 char *mem = malloc(...);
02
03 // Fehlerfall
04 if(NULL == mem) {
05     fprintf(stderr, "%s:%d: malloc failed with reason: %s\n",
06         __FILE__, __LINE__-5, strerror(errno));
07     //alternativ: perror("malloc");
08
09     exit(EXIT_FAILURE);
10 }
```



- Signalisierung durch Rückgabewert nicht immer möglich
- Rückgabewert EOF: Fehlerfall **oder** End-Of-File

```
01 int c;  
02 while ((c=getchar()) != EOF) { ... }  
03 /* EOF oder Fehler? */
```

- Erkennung bei I/O Streams: `ferror(3)` bzw. `feof(3)`

```
01 int c;  
02 while ((c=getchar()) != EOF) { ... }  
03 /* EOF oder Fehler? */  
04 if(ferror(stdin)) {  
05     /* Fehler */  
06     ...  
07 }
```

# Debuggen

---





- Übersetzen mit Debug-Symbolen (-g) & ohne Optimierungen (-O0)

```
01 gcc -g -pedantic -Wall -Werror -O0 -std=c11 -D_XOPEN_SOURCE=700
```

- **Wichtig:** Vor der Abgabe wieder mit Optimierungen testen!
- SPiC IDE hat eine graphische Variante des GDB integriert
- Alternativ: Starten des Debuggers per Kommandozeile

```
01 gdb ./concat  
02 # alternativ ...  
03 cgdb --args ./concat arg0 arg1 ...
```

- Kommandos
  - b(reak): Breakpoint setzen
  - r(un): Programm bei main() starten
  - n(ext): nächste Anweisung (nicht in Unterprogramme springen)
  - s(step): nächste Anweisung (in Unterprogramme springen)
  - p(rint) <var>: Wert der Variablen var ausgeben

⇒ **Debuggen ist (fast immer) effizienter als Trial-and-Error!**



- Informationen über:
  - Speicherlecks (malloc/free)
  - Zugriffe auf nicht gültigen Speicher
- Ideal zum Lokalisieren von Segmentation Faults (SIGSEGV)
- Aufrufe:
  - `valgrind ./concat`
  - `valgrind --leak-check=full --show-reachable=yes`  
↳ `--track-origins=yes ./concat`

## Die Funktion main()

---



- Funktion `main()`: Einsprungstelle für ein C Programm
- Signatur nach Anwendungszweck:
  - AVR: Nur ein Programm
    - ⇒ `void main(void)`
  - Linux: Mehrere Programme
    - ⇒ `int main(void)`
    - ⇒ `int main(int argc, char *argv[])`
- Parameter und Rückgabewert zur Kommunikation



- Kommandozeilenparameter: Argumente für Programme
- `main()` erhält sie als Funktionsparameter:
  - `argc`: Anzahl der Argumente
  - `argv`: Array aus Zeigern auf Argumente⇒ Array von Strings
- Erstes Argument: Programmname



```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 int main(int argc, char *argv[]) {
05     for(int i = 0; i < argc; ++i) {
06         printf("argv[%d]: %s\n", i, argv[i]);
07     }
08
09     return EXIT_SUCCESS;
10 }
```

```
01 $ ./commandline
02 argv[0]: ./commandline
03 $ ./commandline Hallo Welt
04 argv[0]: ./commandline
05 argv[1]: Hallo
06 argv[2]: Welt
```



- Rückgabestatus: Information für den Aufrufenden
- Übliche Codes:
  - EXIT\_SUCCESS: Ausführung erfolgreich
  - EXIT\_FAILURE: Fehler aufgetreten



```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 int main(int argc, char *argv[]) {
05     if(argc == 1) {
06         fprintf(stderr, "No parameters given!\n");
07         return EXIT_FAILURE;
08     }
09
10     // [...]
11
12     return EXIT_SUCCESS;
13 }
```

```
01 $ ./exitcode
02 No parameters given!
03 $ echo $?
04 1
05 $ ./exitcode Hallo Welt
06 $ echo $?
07 0
```



## Aufgabe: concat

---



- Zusammensetzen der übergebenen Kommandozeilenparameter zu einer Gesamtzeichenfolge und anschließende Ausgabe
- Ablauf:
  - Bestimmung der Gesamtlänge
  - Dynamische Allokation eines Puffers
  - Schrittweises Befüllen des Puffers
  - Ausgabe der Zeichenfolge auf dem Standardausgabekanal
  - Freigabe des dynamisch allokierten Speichers
- Reimplementierung der Stringfunktionen der `string.h`:
- Wichtig: Identisches Verhalten (auch im Fehlerfall)

```
01  size_t str_len(const char *s)
02  char *str_cpy(char *dest, const char *src)
03  char *str_cat(char *dest, const char *src)
```



- `malloc(3)` allokiert Speicher auf dem Heap
  - reserviert mindestens `size` Byte Speicher
  - liefert Zeiger auf diesen Speicher zurück
  - schlägt potenziell fehl
- `free(3)` gibt Speicher wieder frei

```
01 char* s = (char *) malloc(...);
02 if(s == NULL) {
03     perror("malloc");
04     exit(EXIT_FAILURE);
05 }
06
07 // [...]
08
09 free(s);
```



```
01 char *get_some_string(void);
02
03 int main(void) {
04     short i = strlen(get_some_string());
05     printf("Size of string: %hd\n", i);
06
07     return EXIT_SUCCESS;
08 }
```



```
01 char *get_some_string(void);
02
03 int main(void) {
04     short i = strlen(get_some_string());
05     printf("Size of string: %hd\n", i);
06
07     return EXIT_SUCCESS;
08 }
```

- Ausgeben der Länge eines Strings:
  - Reicht ein short (meist 32KB adressierbar)?



```
01 char *get_some_string(void);
02
03 int main(void) {
04     int i = strlen(get_some_string());
05     printf("Size of string: %d\n", i);
06
07     return EXIT_SUCCESS;
08 }
```

- Ausgeben der Länge eines Strings:  
→ Reicht ein `int` (meist 2GB adressierbar)?



```
01 char *get_some_string(void);
02
03 int main(void) {
04     unsigned int i = strlen(get_some_string());
05     printf("Size of string: %u\n", i);
06
07     return EXIT_SUCCESS;
08 }
```

- Ausgeben der Länge eines Strings:
  - Reicht ein `unsigned int` (meist 4GB adressierbar)?
- Wie groß ist der größte mögliche String?
- Oder allgemeiner: Wie groß ist das größte Datenobjekt?



- Größe von Datenobjekten wird in `size_t` angegeben
  - `size_t strlen(const char *s);`
  - `void *malloc(size_t size);`
- Breite von `size_t` ist architekturabhängig





- Größe von Datenobjekten wird in `size_t` angegeben
  - `size_t strlen(const char *s);`
  - `void *malloc(size_t size);`
- Breite von `size_t` ist architekturabhängig

```
01 int main(void) {
02     printf("Size of size_t: \t%zu\n",      sizeof(size_t));
03     printf("Size of unsigned int: \t%zu\n", sizeof(unsigned int));
04
05     return EXIT_SUCCESS;
06 }
```

```
01 # 32-Bit Linux
02 $> gcc -o print_sizes print_sizes.c
03 $> ./print_sizes
04     Size of size_t:          4
05     Size of unsigned int:   4
```



- Größe von Datenobjekten wird in `size_t` angegeben
  - `size_t strlen(const char *s);`
  - `void *malloc(size_t size);`
- Breite von `size_t` ist architekturabhängig

```
01 int main(void) {
02     printf("Size of size_t: \t%zu\n",      sizeof(size_t));
03     printf("Size of unsigned int: \t%zu\n", sizeof(unsigned int));
04
05     return EXIT_SUCCESS;
06 }
```

```
01 # 64-Bit Linux
02 $> gcc -o print_sizes print_sizes.c
03 $> ./print_sizes
04     Size of size_t:      8
05     Size of unsigned int: 4
```

## Hands-on: file-concat

---



- Konkateniert den Inhalt mehrerer Dateien  
⇒ Übergebene Dateien öffnen & nacheinander auf `stdout` ausgeben
- Hilfreiche Funktionen:
  - `fopen(3)` ⇒ Öffnen der Datei
  - `fgetc(3)` ⇒ Einlesen einzelner Zeichen
  - `fputc(3)` ⇒ Ausgeben einzelner Zeichen
  - `fclose(3)` ⇒ Schließen der Datei
- Sinnvolle Fehlerbehandlung beachten
  - Fehlende Dateien melden und überspringen
  - Fehlermeldungen auf `stderr` ausgeben
- Erweiterung
  - Behandlung von „-“ Zeichen als speziellen Parameter - vgl. `cat(1)`  
⇒ Zeichen von `stdin` einlesen (und auf `stdout` ausgeben) bis EOF
  - Verwendung: `./file-concat test1.txt - test2.txt`