

# Übungen zu Systemnahe Programmierung in C (SPiC) – Sommersemester 2019

## Übung 8

Benedict Herzog  
Bernhard Heinloth

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



Lehrstuhl für Verteilte Systeme  
und Betriebssysteme



## Linux

### Terminal - Historisches (etwas vereinfacht)



- Als die Computer noch größer waren:



- Als das Internet noch langsam war:



- Farben, Positionssprünge, etc. werden durch spezielle Zeichenfolgen ermöglicht

### Terminal - Funktionsweise



- Drei Standardkanäle für Ein- und Ausgaben



**stdin** Eingaben  
**stdout** Ausgaben  
**stderr** Fehlermeldungen

- Standardverhalten
  - Eingaben kommen von der Tastatur
  - Ausgaben & Fehlermeldungen erscheinen auf dem Bildschirm



- stdout in Datei schreiben

```
01 find . > ordner.txt
```

- stdout als stdin für anderer Programme

```
01 cat ordner.txt | grep tmp | wc -l
```

- Vorteil von `stderr`
  - ⇒ Fehlermeldungen werden weiterhin am Terminal ausgegeben
- Übersicht
  - > Standardausgabe stdout in Datei schreiben
  - >> Standardausgabe stdout an exist. Datei anhängen
  - 2> Fehlerausgabe stderr in Datei schreiben
  - < Standardeingabe stdin aus Datei einlesen
  - | Ausgabe eines Befehls als Eingabe für anderen Befehl

3



- Wechseln in ein Verzeichnis mit `cd` (change directory)

```
01 cd /proj/i4spic/<login>/aufgabeX/ # absolut in Ordner
02 cd aufgabe5/ # relativ zum aktuellen Ordner
03 cd ~ # Benutzerverzeichnis (Home)
04 cd .. # übergeordnete Verzeichnis
```

- Verzeichnisinhalt auflisten mit `ls` (list directory)

```
01 ls # zeige Dateien im akt. Ordner
02 ls -A # zeige auch versteckte Dateien
03 ls -lh # zeige mehr Metadaten
```

4



- Datei oder Ordner kopieren mit `cp` (copy)

```
01 # Kopiere Datei ampel.c aus dem Home in Projektverzeichnis
02 cp ~/ampel.c /proj/i4spic/xy42abcd/aufgabe5/ampel.c
03
04 # Kopiere Ordner aufgabe5/ aus dem Home in Projektverzeichnis
05 cp -r ~/aufgabe5/ /proj/i4spic/xy42abcd/
```

- Datei oder Ordner unwiederbringlich löschen mit `rm` (remove)

```
01 # Lösche Datei test1.c im aktuellen Ordner
02 rm test1.c
03
04 # Lösche Unterordner aufgabe1/ mit allen Dateien
05 rm -r aufgabe1
```

4

- Per Signal: CTRL-C (kann von Programmen ignoriert werden)
- Von einer anderen Konsole aus: `killall concat` beendet alle Programme mit dem Namen "concat"
- Von der selben Konsole aus:
  - CTRL-Z hält den aktuell laufenden Prozess an
  - `killall concat` beendet alle Programme namens concat
    - ⇒ Programme anderer Benutzer dürfen nicht beendet werden
  - `fg` setzt den angehaltenen Prozess fort
- Wenn nichts mehr hilft: `killall -9 concat`

5



```
Project
├── jys2coty
│   ├── aufgabe1
│   ├── aufgabe2
│   ├── aufgabe3
│   ├── aufgabe4
│   └── aufgabe5
│       ├── concat
│       ├── concat.c
│       ├── aufgabe6
│       ├── korrektur
│       └── pub
└── aufgabe5/concat.c 1:1 LF C
```

```
concat.c
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main(int argc, char *argv[]) {
5     printf("Hallo World\n");
6     return EXIT_SUCCESS;
7 }
```

```
Atom Shell Commands
make -B trac
cc -std=c11 -pedantic -D_XOPEN_SOURCE=700 -Wall -Werror -O3 -trac -o trac
[Finished in 0.14 seconds]
```

```
Terminal
jys2coty@fau10sr0:~/proj/14spic/jys2coty/aufgabe5$ ls
concat.c
jys2coty@fau10sr0:~/proj/14spic/jys2coty/aufgabe5$ gcc -pedantic -Wall -Werror -O3 -std=c11 -D_XOPEN_SOURCE=700 -o
concat concat.c
jys2coty@fau10sr0:~/proj/14spic/jys2coty/aufgabe5$ ls
concat  concat.c
jys2coty@fau10sr0:~/proj/14spic/jys2coty/aufgabe5$ ./concat
Hallo world
jys2coty@fau10sr0:~/proj/14spic/jys2coty/aufgabe5$
```

- **Terminal:** öffnet ein Terminal und startet eine Shell
  - effiziente Interaktion mit dem System
  - optional Vollbild
- **Debug:** startet den Debugmodus
- **Make:** siehe nächste Woche

6

## ■ Programm mit dem GCC übersetzen

```
01 gcc -pedantic -Wall -Werror -O3 -std=c11 -D_XOPEN_SOURCE=700 -o
    concat concat.c
```

- **gcc** ruft den Compiler auf (GNU Compiler Collection)
- **-pedantic** aktiviert Warnungen (Abweichungen vom C-Standards)
- **-Wall** aktiviert Warnungen (typische Fehler, z.B.: `if (x = 7)`)
- **-Werror** wandelt Warnungen in Fehler um
- **-O3** aktiviert Optimierungen (Level 3)
- **-std=c11** setzt verwendeten Standard auf C11
- **-D\_XOPEN\_SOURCE=700**
  - fügt POSIX Erweiterungen hinzu
- **-o concat** legt Namen der Ausgabedatei fest (Standard: `a.out`)  
`concat.c ...` zu kompilierende Datei(en)
- Ausführen des Programms mit `./concat`
- **Abgaben werden von uns mit diesen Optionen getestet**

7



- Programm mit dem GCC übersetzen  
(inklusive Debugsymbole und ohne Optimierungen)

```
01 gcc -pedantic -Wall -Werror -O0 -std=c11 -D_XOPEN_SOURCE=700 -g -
    o concat concat.c
```

- **-O0** verhindert Optimierungen des Programms
- **-g** belässt Debugsymbole in der ausführbaren Datei

⇒ ermöglicht dem Debugger Verweise zur Quelldatei herzustellen

- Hinweis: Pfeiltaste ↑ iteriert durch frühere Befehle
- ⇒ GCC Aufruf nur einmal tippen

8

- 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`

9

- Das Linux-Hilfesystem
- aufgeteilt nach verschiedenen Sections
  - 1 Kommandos
  - 2 Systemaufrufe
  - 3 Bibliotheksfunktionen
  - 5 Dateiformate (spezielle Datenstrukturen, etc.)
  - 7 verschiedenes (z.B. Terminaltreiber, IP, ...)
- man-Pages werden normalerweise mit der Section zitiert: `printf(3)`

```
01 # man [section] Begriff
02 man 3 printf
```

- Suche nach Sections: `man -f Begriff`
- Suche von man-Pages zu einem Stichwort: `man -k Stichwort`

10

## Fehlerbehandlung

---

- Abgespeckte (hübschere) Version der Manpages
- Bieten nur eine Übersicht, keine vollständige Spezifikation
- Aus der SPiC-IDE abrufbar (Hilfe-Button wenn im Linux-Modus)
- Auf der Webseite zu finden  
[https://www4.cs.fau.de/Lehre/SS19/V\\_SPIC/Linux/libc-api/](https://www4.cs.fau.de/Lehre/SS19/V_SPIC/Linux/libc-api/)
- Unsere Übersicht ersetzen die Manpages nicht
- In der Klausur: ausgedruckte Manpages!

11

## Fehlerursachen



- 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

12



- 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

13

- 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

14



- 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__-3, strerror(errno));
07     //alternativ: perror("malloc");
08
09     exit(EXIT_FAILURE);
10 }

```

15

- 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 }

```

16



## Kommandozeilenparameter

```

01 ...
02 int main(int argc, char *argv[]){
03     strcmp(argv[argc - 1], ... )
04     ...
05     return EXIT_SUCCESS;
06 }

```

- Übergabeparameter:
  - main() bekommt vom Betriebssystem Argumente
  - argc: Anzahl der Argumente
  - argv: Array aus Zeigern auf Argumente (Indizes von 0 bis argc-1)
- Rückgabeparameter:
  - Rückgabe eines Wertes an das Betriebssystem
  - Zum Beispiel Fehler des Programms: return EXIT\_FAILURE;

17

## Vertiefung: Strings

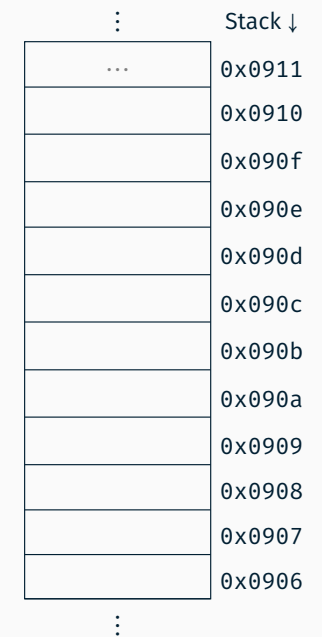


- char: Einzelnes Zeichen (z.B. 'a')
- String: Array von chars (z.B. "Hello")
- In C: Letztes Zeichen eines Strings: '\0'  
⇒ Speicherbedarf: strlen(s) + 1

```

01 char *s = "World\n";
02 char c = s[0];
03 char c = s[4];
04 char *s2 = s + 2;
05 char c = s2[1];

```



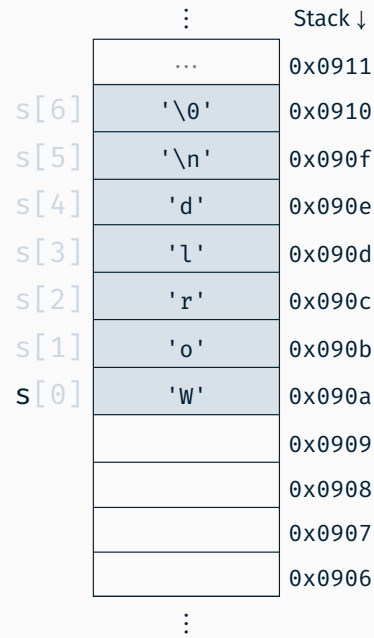
## Vertiefung C Strings

18



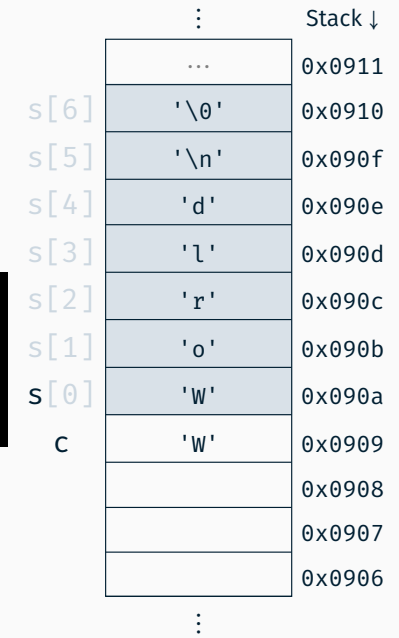
- char: Einzelnes Zeichen (z.B. 'a')
- String: Array von chars (z.B. "Hello")
- In C: Letztes Zeichen eines Strings: '\0'  
⇒ Speicherbedarf: strlen(s) + 1

```
01 char *s = "Worl\rd\n";
02 char c = s[0];
03 char c = s[4];
04 char *s2 = s + 2;
05 char c = s2[1];
```



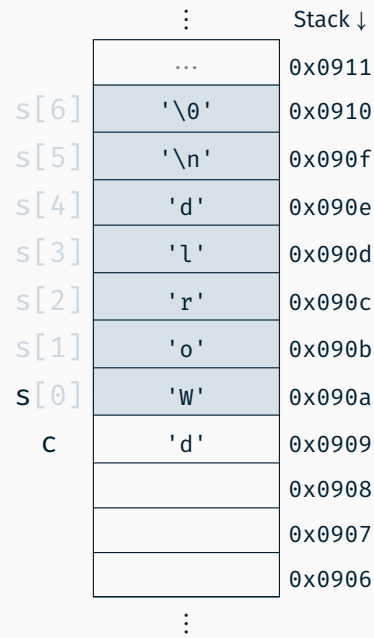
- char: Einzelnes Zeichen (z.B. 'a')
- String: Array von chars (z.B. "Hello")
- In C: Letztes Zeichen eines Strings: '\0'  
⇒ Speicherbedarf: strlen(s) + 1

```
01 char *s = "Worl\rd\n";
02 char c = s[0];
03 char c = s[4];
04 char *s2 = s + 2;
05 char c = s2[1];
```



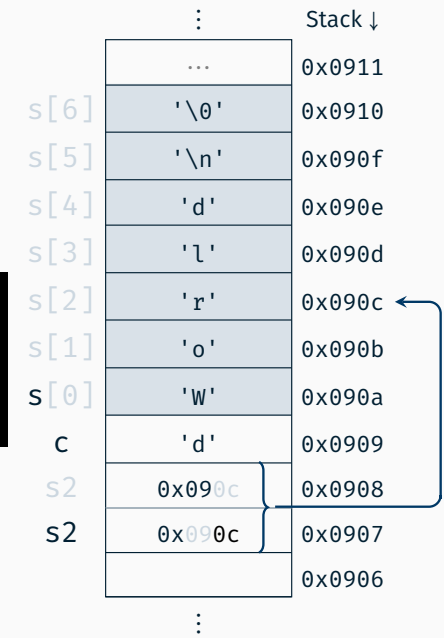
- char: Einzelnes Zeichen (z.B. 'a')
- String: Array von chars (z.B. "Hello")
- In C: Letztes Zeichen eines Strings: '\0'  
⇒ Speicherbedarf: strlen(s) + 1

```
01 char *s = "Worl\rd\n";
02 char c = s[0];
03 char c = s[4];
04 char *s2 = s + 2;
05 char c = s2[1];
```



- char: Einzelnes Zeichen (z.B. 'a')
- String: Array von chars (z.B. "Hello")
- In C: Letztes Zeichen eines Strings: '\0'  
⇒ Speicherbedarf: strlen(s) + 1

```
01 char *s = "Worl\rd\n";
02 char c = s[0];
03 char c = s[4];
04 char *s2 = s + 2;
05 char c = s2[1];
```



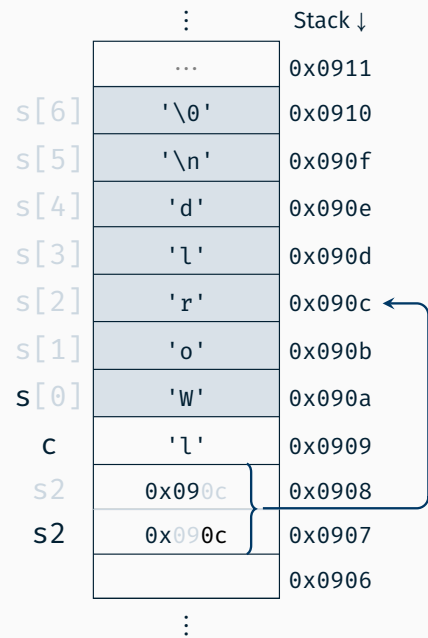


- char: Einzelnes Zeichen (z.B. 'a')
- String: Array von chars (z.B. "Hello")
- In C: Letztes Zeichen eines Strings: '\0'  
⇒ Speicherbedarf: strlen(s) + 1

```

01 char *s = "Worl d\n";
02 char c = s[0];
03 char c = s[4];
04 char *s2 = s + 2;
05 char c = s2[1];

```



18

- `size_t strlen(const char *s)`
  - Bestimmung der Länge einer Zeichenkette s (ohne abschließendes Null-Zeichen)
- `char *strcpy(char *dest, const char *src)`
  - Kopieren einer Zeichenkette src in einen Puffer dest (inkl. Null-Zeichen)
  - Gefahr: Buffer Overflow (⇒ `strncpy(3)`)
- `char *strcat(char *dest, const char *src)`
  - Anhängen einer Zeichenkette src an eine existierende Zeichenkette im Puffer dest (inkl. Null-Zeichen)
  - Gefahr: Buffer Overflow (⇒ `strncat(3)`)
- Dokumentation: `strlen(3)`, `strcpy(3)`, `strcat(3)`

19

## Aufgabe: concat

### 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)

```

20





- `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);

```

21

## Hands-on: Buffer Overflow

---



- Passwortgeschütztes Programm

```

01 # Usage: ./print_exam <password>
02 ./print_exam spic
03 Correct Password
04 Printing exam...

```

- Passwortgeschütztes Programm

```

01 # Usage: ./print_exam <password>
02 ./print_exam spic
03 Correct Password
04 Printing exam...

```

- Ungeprüfte Benutzereingaben ⇒ Buffer Overflow

```

01 long check_password(const char *password) {
02     char buff[8];
03     long pass = 0;
04
05     strcpy(buff, password);
06     if(strcmp(buff, "spic") == 0) {
07         pass = 1;
08     }
09     return pass;
10 }

```

22

22



### ■ Passwortgeschütztes Programm

```
01 # Usage: ./print_exam <password>
02 ./print_exam spic
03 Correct Password
04 Printing exam...
```

### ■ Ungeprüfte Benutzereingaben ⇒ Buffer Overflow

```
01 long check_password(const char *password) {
02     char buff[8];
03     long pass = 0;
04
05     strcpy(buff, password);
06     if(strcmp(buff, "spic") == 0) {
07         pass = 1;
08     }
09     return pass;
10 }
```

22

```
01 long check_password(const char *password) {
02     char buff[8];
03     long pass = 0;
04
05     strcpy(buff, password);
06     if(strcmp(buff, "spic") == 0) {
07         pass = 1;
08     }
09     return pass;
10 }
```

### ■ Mögliche Lösungen

- Prüfen der Benutzereingabe
- Dynamische Allokation des Puffers
- Sichere Bibliotheksfunktionen verwenden ⇒ z.B. strncpy(3)

23