

Übungen zu Systemnahe Programmierung in C (SPiC) – Sommersemester 2025

Übung 9

Maxim Ritter von Onciul
Eva Dengler

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



Lehrstuhl für Informatik 4
Systemsoftware



Friedrich-Alexander-Universität
Technische Fakultät

Linux

- Als die Computer noch größer waren:



Seriell

Computer

Televideo 925 (Public Domain: Wtshymanski @Wikipedia)

- Als das Internet noch langsam war:

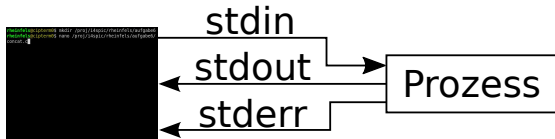


Netzwerk

Server

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

- 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 . > directories.txt
```

- stdout als stdin für anderer Programme

```
01 cat directories.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



■ 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
```



■ Datei oder Ordner kopieren mit cp (copy)

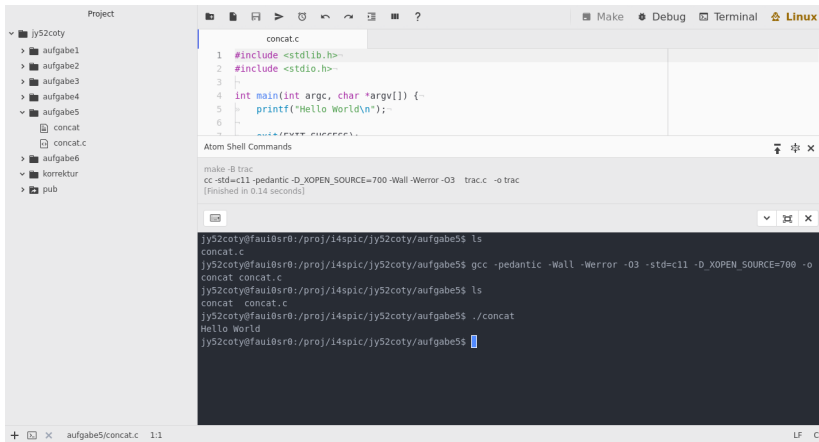
```
01 # Kopiere Datei ampel.c aus dem Home in Projektverzeichnis
02 cp ~/ampel.c /proj/i4spic/xy42abcd/aufgabe5/ampel.c
03
04 # Kopiere Ordner aufgabe5/ aus dem Home in Projektverzeichnis
05 cp -r ~/aufgabe5/ /proj/i4spic/xy42abcd/
```

■ 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
```



- 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`



The screenshot displays the SPiC IDE interface. On the left, a 'Project' sidebar shows a tree view with folders 'aufgabe1' through 'aufgabe6' and a file 'pub'. The main editor area shows a C file named 'concat.c' with the following code:

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

Below the editor, the 'Atom Shell Commands' panel shows the command: `make -B trac cc -std=c11 -pedantic -D_XOPEN_SOURCE=700 -Wall -Werror -O3 -o trac.c -o trac` [Finished in 0.14 seconds]. At the bottom, a terminal window shows the execution of these commands and the output 'Hello World'.

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

■ 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



- 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

- Informationen über:
 - Speicherlecks (`malloc(3)/free(3)`)
 - 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 Ausgabe ist deutlich hilfreicher, wenn das analysierte Binary mit Debugsymbolen gebaut wird



- 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`



- 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://sys.cs.fau.de/lehre/ss25/spic/uebung/libcapi/>

- Unsere Übersicht ersetzen die Manpages nicht
- In der Klausur: ausgedruckte Manpages!



- 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 }
```



- 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

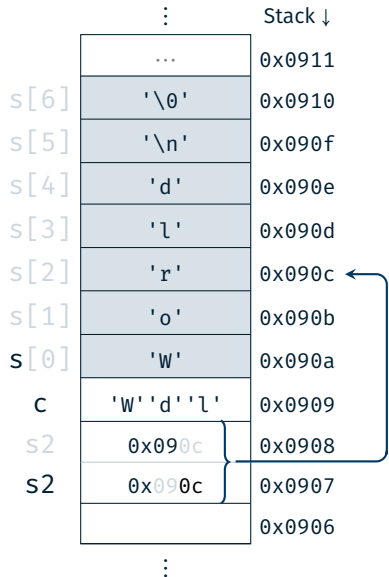


```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 int main(int argc, char *argv[]) {
05     if(argc == 1) {
06         fprintf(stderr, "No parameters given!\n");
07         return EXIT_FAILURE;
08     }
09
10     // [...]
11
12     return EXIT_SUCCESS;
13 }
```

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

- 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 c = s[4];  
04 char *s2 = s + 2;  
05 c = s2[1];
```



- `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 ($\Rightarrow$ `strncpy(3)`)
- `char *strcat(char *dest, const char *src)`
 - Anhängen (konkateneren, engl. `concat`, kurz `cat`) einer Zeichenkette `src` an eine existierende Zeichenkette im Puffer `dest` (inkl. Null-Zeichen)
 - Gefahr: Buffer Overflow ($\Rightarrow$ `strncat(3)`)
- Dokumentation: `strlen(3)`, `strcpy(3)`, `strcat(3)`



```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <string.h>
04
05 int main(void) {
06     const char *hallo = "Hallo";
07     const char *spic = "SPiC";
08
09     char altered_string[11]; // Platz für "Hallo SPiC"
10
11     strcpy(altered_string, hallo); // "Hallo"
12     strcat(altered_string, " ");   // "Hallo "
13     strcat(altered_string, spic);  // "Hallo SPiC"
14     strlen(altered_string);        // -> 10
15
16     return EXIT_SUCCESS;
17 }
```



- 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);
```



■ 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 }
```



```
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)



```
01 long pass = 0;
02 char buff[8];
03 strcpy(buff, password);
04
05 if(strcmp(buff, "spic")) {
06     printf("Wrong Pass.\n");
07 } else {
08     printf("Correct Pass.\n");
09     pass = 1;
10 }
11
12 return pass; // pass = 1
13
```

	:	Stack ↓
	0	0x090d
	0	0x090c
	0 ('\0')	0x090b
pass	65 ('A')	0x090a
buff[7]	65 ('A')	0x0909
buff[6]	65 ('A')	0x0908
buff[5]	65 ('A')	0x0907
buff[4]	65 ('A')	0x0906
buff[3]	65 ('A')	0x0905
buff[2]	65 ('A')	0x0904
buff[1]	65 ('A')	0x0903
buff[0]	65 ('A')	0x0902
	:	

Hands-on: Linux, GCC & Valgrind

Screencast: <https://www.video.uni-erlangen.de/clip/id/18667>



- *Nur online!*
- Ziele:
 - SPiC IDE für Linux verwenden
 - Programm via Kommandozeile übersetzen
 - Umgang mit Valgrind üben