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

Terminal - Historisches (etwas vereinfacht)

- Als die Computer noch größer waren:

Seriell — Computer

Televideo 925 (Public Domain: Wtshymanski @Wikipedia)

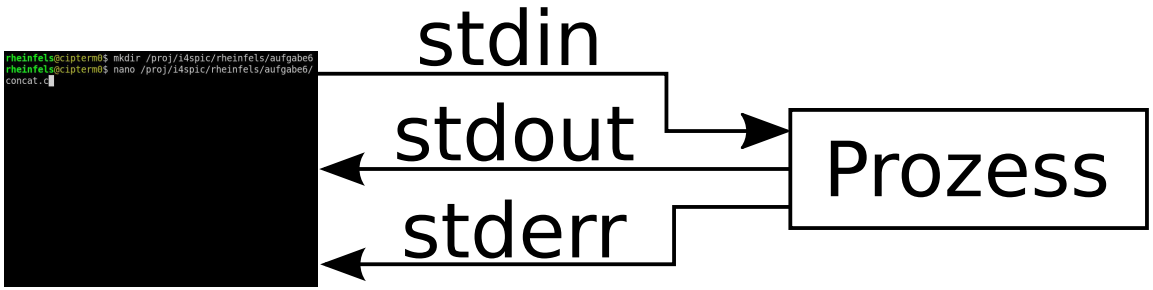
- Als das Internet noch langsam war:

Netzwerk — Server

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

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

Terminal - Standardkanäle umleiten

- `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

Shell - Wichtige Kommandos

- 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
```

Shell - Wichtige Kommandos

- 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/
```

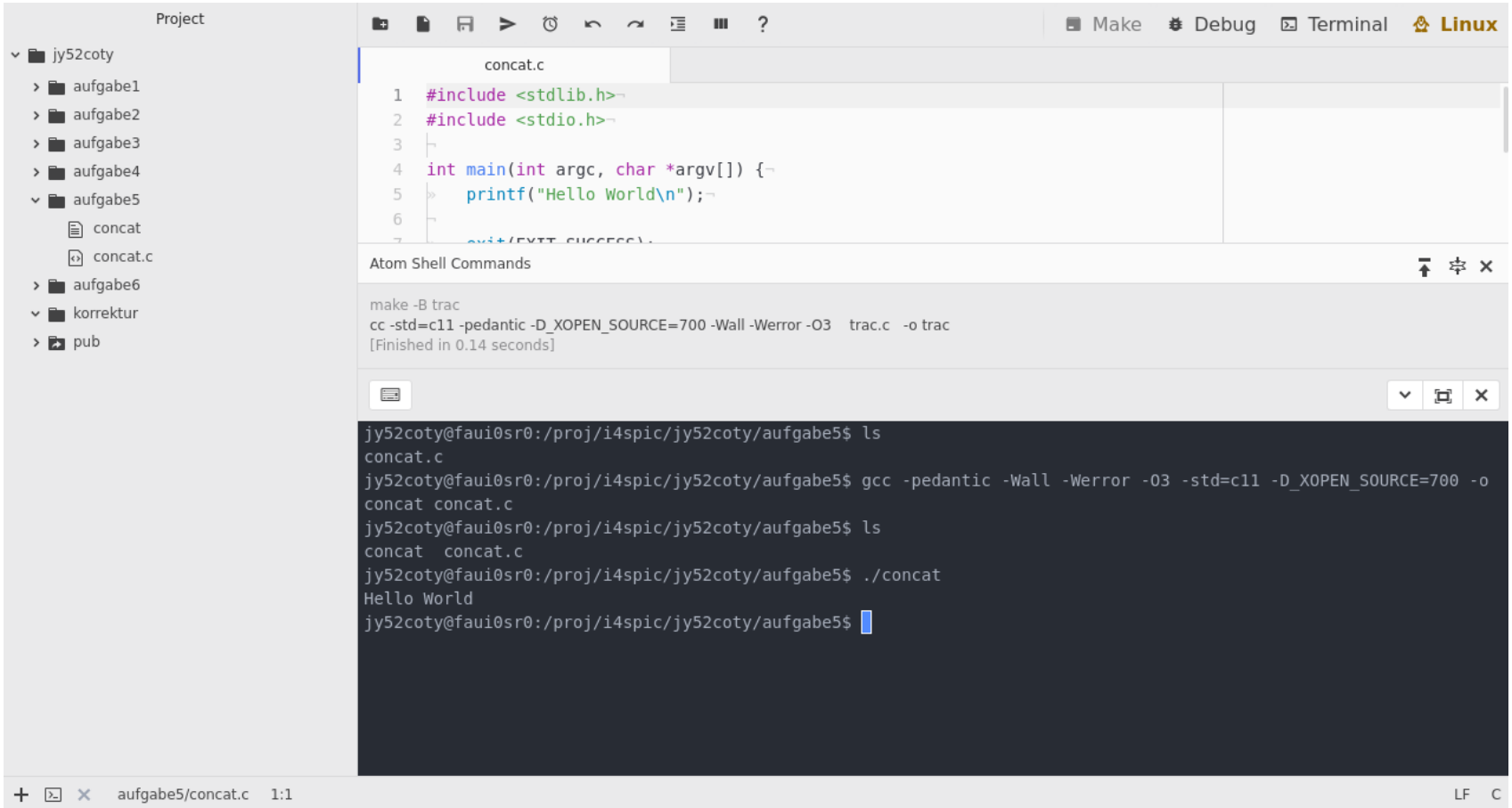
- Datei oder Ordner unwiederbringlich löschen mit `rm` (remove)

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

Shell - Programme beenden

- 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`

SPiC IDE (Linux)



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

Übersetzen & Ausführen

- 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

Übersetzen & Ausführen

- 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

Valgrind

- 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

Manual Pages

- 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`

Linux `libc`-API

- 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!

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

Fehlerbehandlung

- 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 in Bibliotheksfunktionen

- 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

Fehler in Bibliotheksfunktionen

- 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 }
```

Erweiterte Fehlerbehandlung

- 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 }
```

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

- Kommandozeilenparameter: Argumente für Programme
- `main()` erhält sie als Funktionsparameter:
  - `argc`: Anzahl der Argumente
  - `argv`: Array aus Zeigern auf Argumente

  $\Rightarrow$ Array von Strings
- Erstes Argument: Programmname

Kommandozeilenparameter – Beispiel

```c
#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;
}
```

```
$ ./commandline
argv[0]: ./commandline
$ ./commandline Hallo Welt
argv[0]: ./commandline
argv[1]: Hallo
argv[2]: Welt
```

Rückgabestatus

- Rückgabestatus: Information für den Aufrufenden
- Übliche Codes:
  - `EXIT_SUCCESS`: Ausführung erfolgreich
  - `EXIT_FAILURE`: Fehler aufgetreten

Rückgabestatus – Beispiel

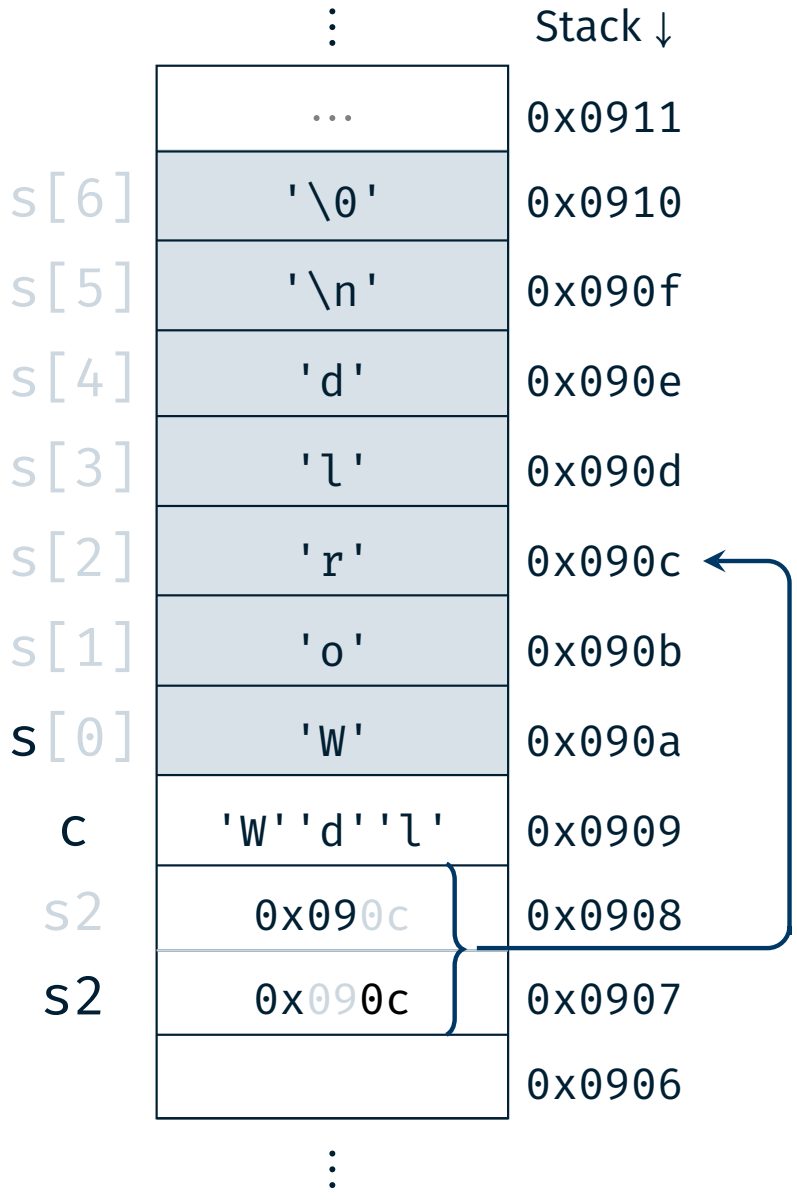
```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  if(argc == 1) {
    fprintf(stderr, "No parameters given!\n");
    return EXIT_FAILURE;
  }

  // [...]

  return EXIT_SUCCESS;
}
```

```
$ ./exitcode
No parameters given!
$ echo $?
1
$ ./exitcode Hello world
$ echo $?
0
```

Vertiefung: Strings

- `char`: Einzelnes Zeichen (z.B. `'a'`)
- String: Array von `char`s (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];
```



Stringfunktionen

- `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 (konkatenieren, engl. concat, kurz cat) 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)`

Stringfunktionen – Beispiel

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

int main(void) {
  const char *hallo = "Hallo";
  const char *spic = "SPiC";

  char altered_string[11];  // Platz für "Hallo SPiC"

  strcpy(altered_string, hallo);  // "Hallo"
  strcat(altered_string, " ");    // "Hallo "
  strcat(altered_string, spic);   // "Hallo SPiC"
  strlen(altered_string);         // -> 10

  return EXIT_SUCCESS;
}
```

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)
```

Dynamische Speicherverwaltung

- `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);
```

Hands-on: Buffer Overflow

- Passwortgeschütztes Programm

```
# Usage: ./print_exam <password>
./print_exam spic
Correct Password
Printing exam...
```

- Ungeprüfte Benutzereingaben ⇒ Buffer Overflow

```
long check_password(const char *password) {
  char buff[8];
  long pass = 0;

  strcpy(buff, password);
  if(strcmp(buff, "spic") == 0) {
    pass = 1;
  }
  return pass;
}
```

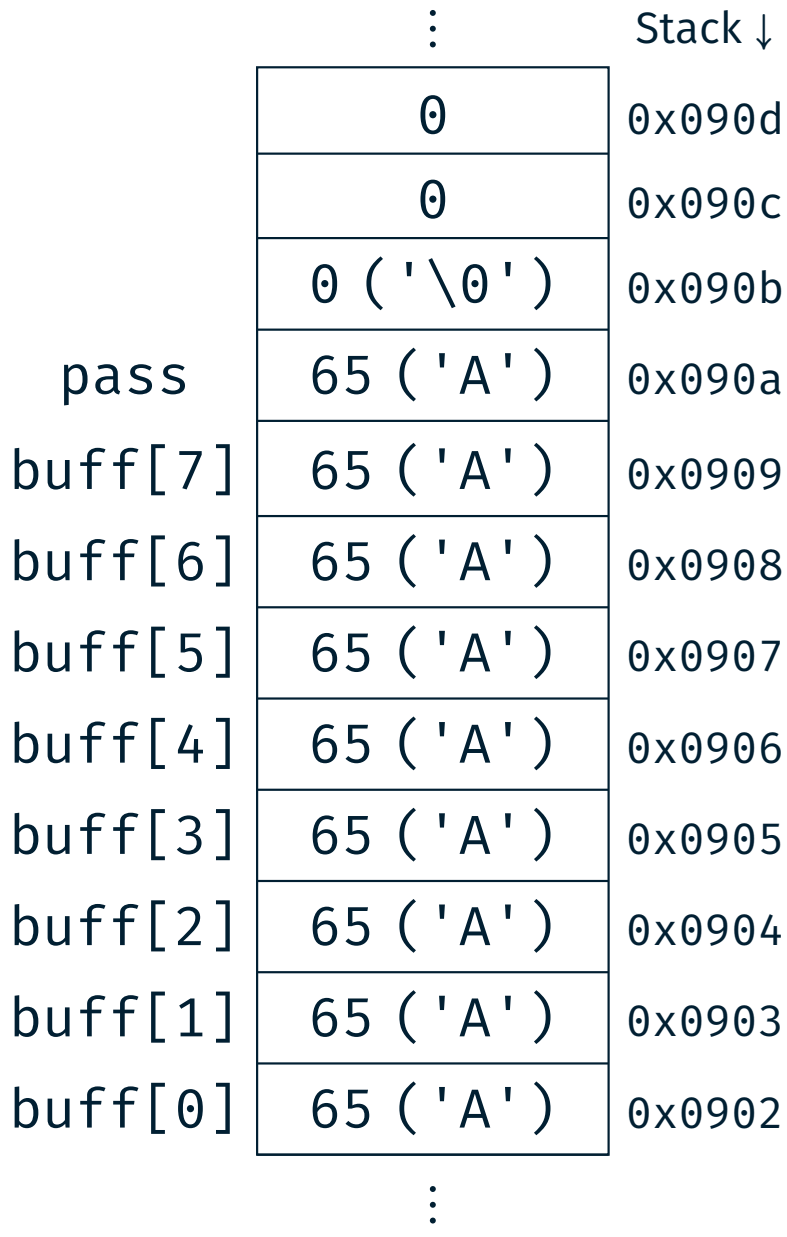
Hands-on: Buffer Overflow

```
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 }
```

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

Bufferoverflow

```
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
```



Hands-on: Linux, GCC & Valgrind

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

Hands-on: Linux, GCC & Valgrind

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