Konzepte systemnaher Programmierung

Technische Hochschule Mittelhessen

Andre Rein

— Einführung —

Konzepte systemnaher Programmierung

Der Fokus der Lehrveranstaltung **Konzepte systemnaher Programmierung** liegt darauf, die Architektur systemnaher Software und die Zusammenhänge zwischen den Abstraktionen einer Programmiersprache und den tatsächlichen Strukturen und Abläufen in Computerhardware, zu verstehen.

- Systemnahe Programmierung beschäftigt sich mit dem Lösen von Problemen auf einer relativ niedrigen Abstraktionsebene.
- Im Fokus stehen hierbei Verfahren, Methoden und Konzepte zu verstehen und anzuwenden, die sehr nah an den tatsächlichen internen Abläufen eines Rechners liegen.

Warum C

- Die Verwendung einer systemnahen Programmiersprache wie `C` ermöglicht es, einen **ungefilterten** Blick auf diese internen Abläufe zu werfen und diese zu verstehen.
- `C` versteckt keine Funktionalität vor dem Programmierer und abstrahiert (im Basisumfang) nur das nötigste, um eine **relativ** *einfache* Programmierung zu ermöglichen.
  - Z.B. (Direkte Verwendung von Zeigern, keine implizite Speicherverwaltung, schwache Typisierung, keine Unterscheidung zwischen bestimmten Datentypen (Ansammlung von Bytes im Speicher, die eine bestimmte Semantik haben))
- Durch die fehlende Abstraktion ist es möglich, sehr angepassten und performanten Code zu entwickeln, der **exakt** und **ausschließlich** das tut, was der Programmierer möchte/implementiert hat.
  - Das birgt sehr viele Gefahren (*Fehler zu machen*) aber auch sehr große Chancen (**Optimierung** und **Geschwindigkeit**)

Insbesondere bei der Entwicklung von Betriebssystemen und im Bereich eingebetteter Systeme ist die Entwicklung von Software in `C` auch heute noch der Standard! Außerdem basieren sehr viele andere Sprachen auf Konzepten von `C` und haben starke Ähnlichkeit zu dessen Syntax.

Warum C

Auf den Punkt gebracht: Ein Verständnis von systemnaher Programmierung und das Erlernen der Programmiersprache `C` macht Sie zu einem **besseren** Informatiker für den Rest Ihres Lebens, egal welche Richtung in der Informatik Sie später einmal einschlagen werden!

Ein erstes Programm

```c
int  main (int argc, char *argv[]) {
    return 0;
}
```

- `int main` → main-Funktion mit Rückgabewert vom Typ `int`
- `int argc` *(argument counter)* → Anzahl der Argumente, d.h. Elemente in `argv`
- `argv` *(argument vector)* → Array von Strings, die die übergebenen Argumente enthalten
- `return 0` → Rückgabe des `int`-Wertes `0` (`0` signalisiert dem Betriebssystem **Erfolg** und `!=0` **Misserfolg**)

Lesen von Deklarationen

```
int argc; 1
char str[100]; 2
int *w; 3
unsigned int y; 4
char *argv[]; 5
char **argv2; 6
```

Generell: Erst Variablenname dann Typ lesen!

1. `argc` ist ein Integer (ist eine *vorzeichenbehaftete Ganzzahl*)
2. `str` ist ein Array von 100 Zeichen
3. `w` ist ein Zeiger auf einen Integer (auf eine *vorzeichenbehaftete Ganzzahl*)
4. `y` ist ein vorzeichenloser Integer (eine *vorzeichenlose Ganzzahl*)
5. `argv` ist eine Array unbestimmter Größe, das Zeiger auf Zeichen enthält
6. `argv2` ist Zeiger auf (*einen* oder *mehrere*) Zeiger auf Zeichen

Ein erstes Programm (revisit)

```c
int  main (int argc, char *argv[]) {
    return 0;
}
```

- **Abmachungen**:
  - `argv` ist ein Array von Strings (das wissen wir hier aus dem Kontext)
  - `argc` enthält die Anzahl der Strings in `argv`
  - `argv[0]` enthält den Namen des aufgerufenen Programms inklusive der Pfadangabe

Ein erstes Programm (Kompiliervorgang)

Datei test1.c

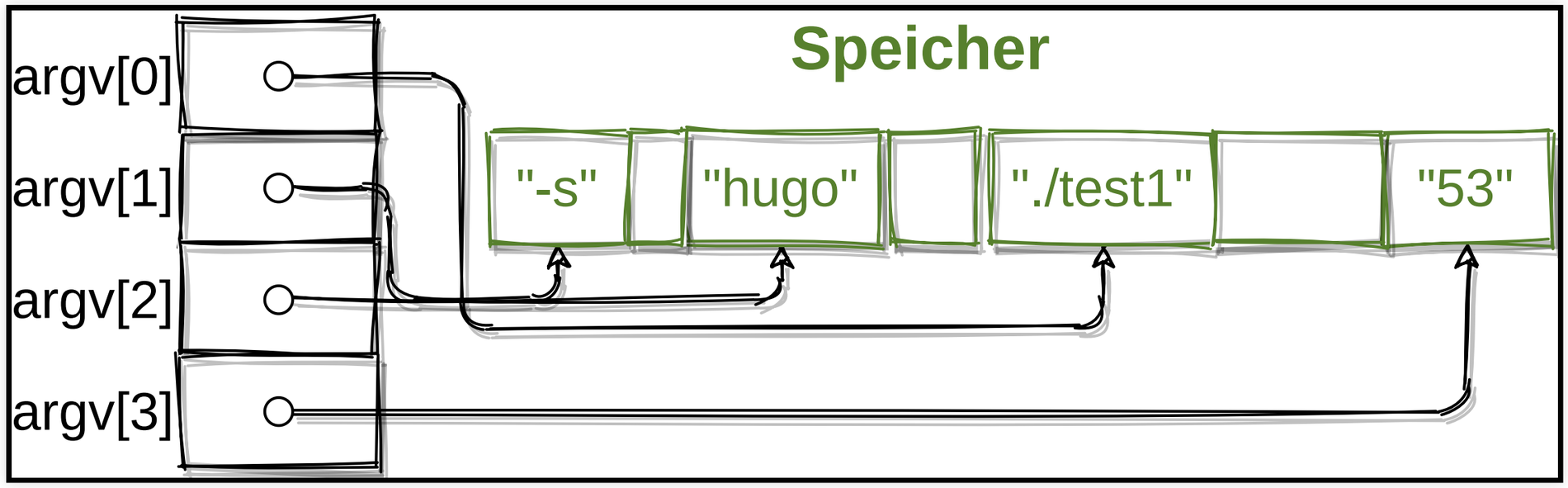
```c
int  main (int argc, char *argv[])
{
    return 0;
}
```

Kompilieren und Aufrufen

```
$ ls
test1.c
$ gcc -g -Wall -std=c99 -pedantic -o test1 test1.c
$ ls
test1 test1.c
$ ./test1
```

Ein erstes Programm (Speicher)

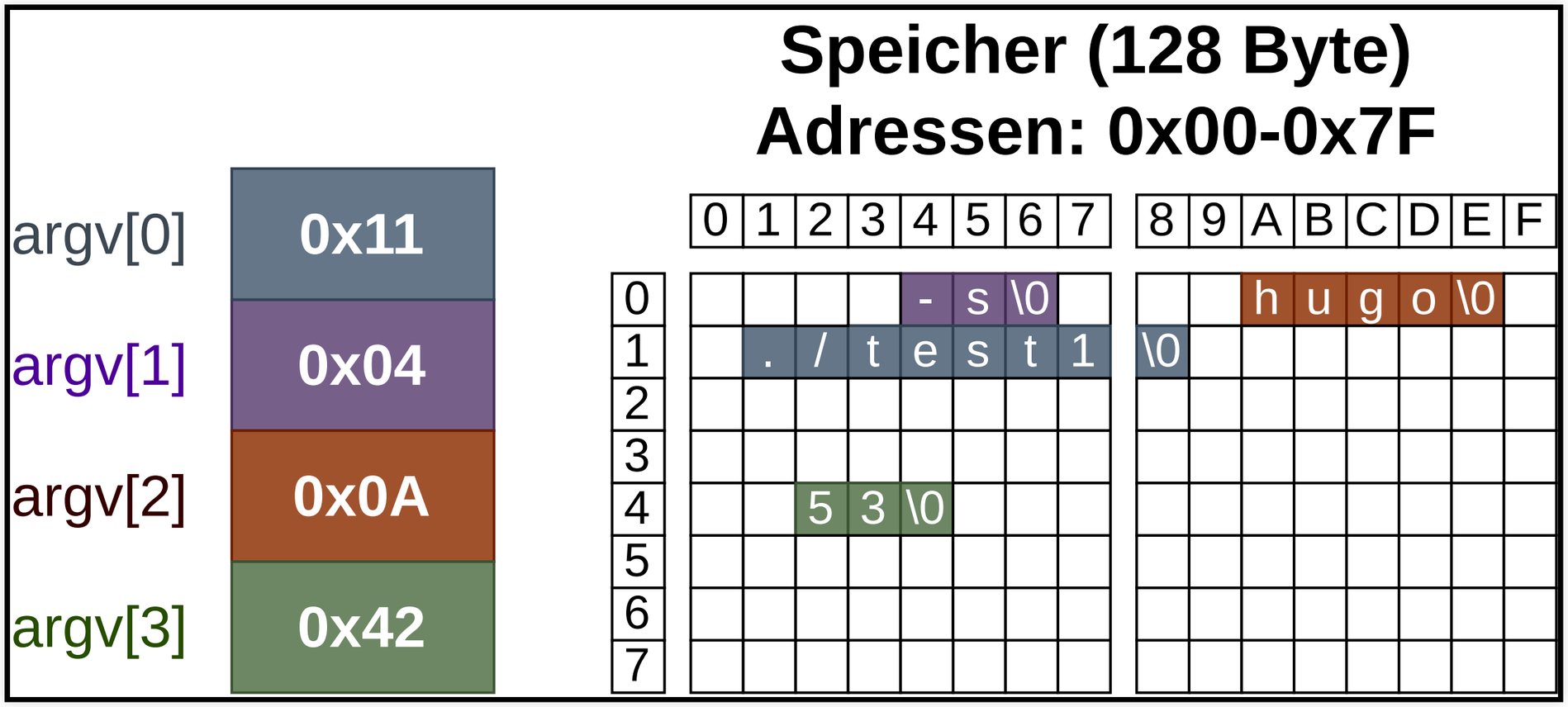
Aufruf: `$ ./test1 -s hugo 53`



- In den Feldern des `argv` -Arrays befinden sich Zeiger zu Speicheradressen, die die Übergabeparameter als Strings beinhalten.
  - Es handelt sich dabei um sog. *C-Strings*: **Null-terminierte** Strings im Speicher die beliebige Zeichen enthalten und deren Ende mit einer `\0` (`0x00` in ASCII) angegeben wird.

Ein erstes Programm (Speicher)

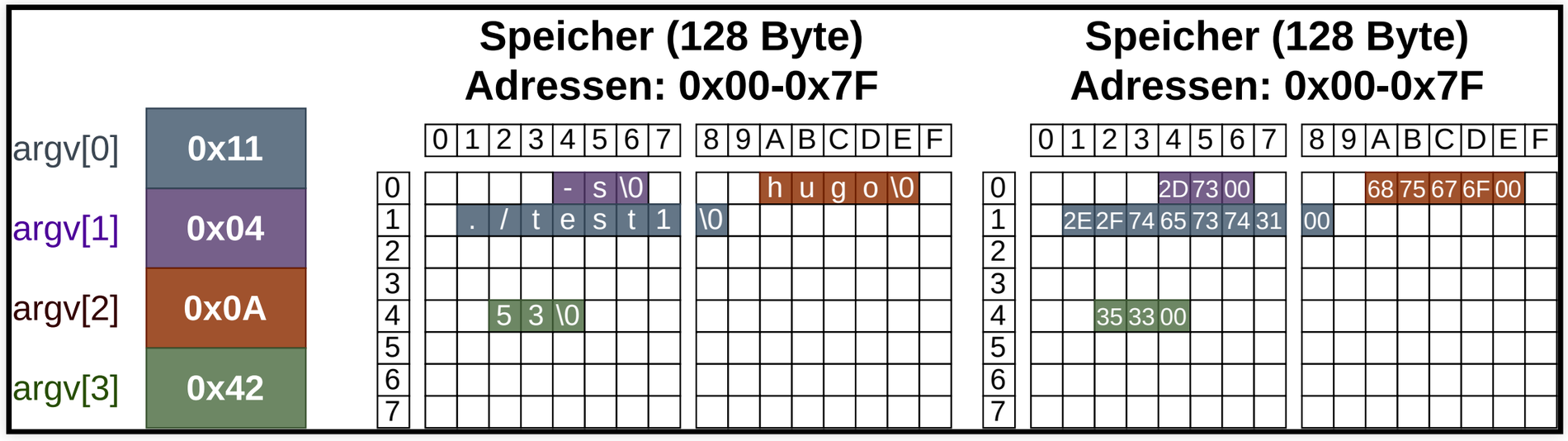
Aufruf: `$ ./test1 -s hugo 53`



- Die Adressen sind hier natürlich beliebig gewählt

Ein erstes Programm (Speicher)

Aufruf: `$ ./test1 -s hugo 53`



- Im Speicher stehen keine Zeichen im Sinne von Buchstaben, sondern Bytes, die bei der Ausgabe eines Strings `%s` oder eines Chars `%c` als ASCII-Zeichen **interpretiert** werden.

Formatierte Ausgabe mit printf

- Eine formatierte Ausgabe erzeugt man mit der Funktion `printf(char * fmt, ...);`
  - Hierbei erfolgt die Übergabe der Variablen in `...`
- Beispiel: `printf("%d 0x%x [%c]\n", 42, 42, 0x42);` → Ausgabe: `42 [0x2A] [B]\n`

Datei test2.c

```c
#include <stdio.h>

int main (int argc, char *argv[]){
    printf("%d 0x%x [%c]\n", 42, 42, 0x42);
    return 0;
}
```

```
$ gcc -g -Wall -std=c99 -pedantic -o test2 test2.c
$ ./test2
42 0x2a [B]
$
```

- Manpage: `man 3 printf`

%s	String bis zum abschließenden `\0`
%d,%i	Ganzzahl in Dezimaldarstellung
%x,%X	Ganzzahl in Hexadezimaldarstellung
%u	Ganzzahl in vorzeichenloser Dezimaldarstellung
%c	Ganzzahl als ASCII-Zeichen
%f	Fließkommazahl
%p	Zeiger in maschinenabhängiger Darstellung

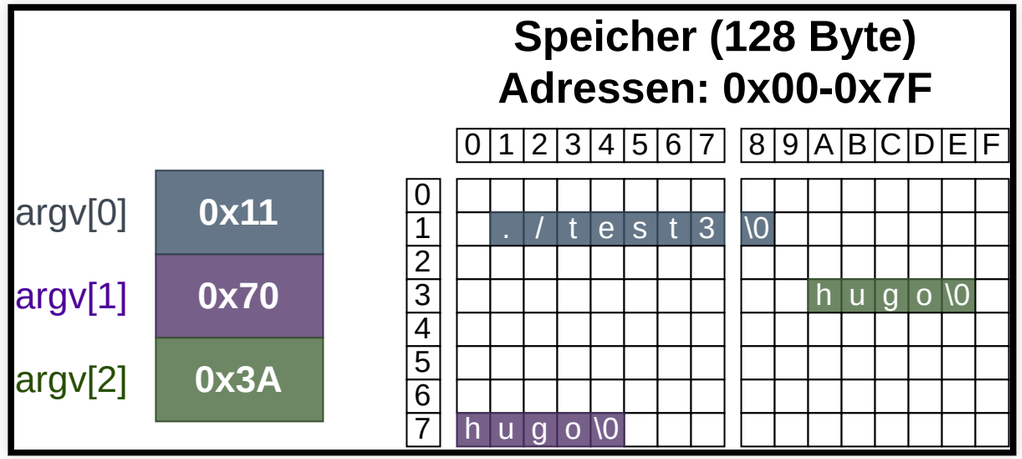
Stringvergleiche

Das Ziel ist es, zu vergleichen, ob der **Inhalt** eines String gleich dem Inhalt eines zweiten Strings ist.

- Aufruf: `$ ./test1 hugo hugo`

```c
#include <stdio.h>

int main (int argc, char *argv[]){
    int result;
    result=(argv[1] == argv[2]);
    printf("[%d] %s\n", result,
        (result == 1 ? "True": "False"));
    return 0;
}
```



Sehen Sie sich den Code an: *Was wird hier eigentlich miteinander verglichen?*

Es werden die Werte (also die Adressen) der Pointer miteinander verglichen! `result = (0x70 == 0xA)`

Stringvergleiche strcmp

- Prototyp: `int strcmp(const char * s1, const char * s2);` in `string.h`
- Manpage `man 3 strcmp` :
- Manpage → RETURN VALUES:
  - `0` , if the s1 and s2 are equal;
  - `<0` if s1 is less than s2;
  - `>0` if s1 is greater than s2.

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

int main (int argc, char *argv[]){
    int result;
    result=strcmp(argv[1], argv[2]);
    printf("[%d] %s\n", result,
        (result == 0 ? "True": "False")); (1)
    return 0;
}
```

```
$ gcc -g -Wall -std=c99 -pedantic -o test4 test4.c
$ ./test4 "HUGO" "HUGO"
[0] True
$ ./test4 "HUGO1" "HUGO2"
[-1] False
$ ./test4 "HUGO2" "HUGO1"
[1] False
```

(1) Hier wird nun auf `0` geprüft! `0` bedeutet gleich!

Stringvergleiche strcmp

Strings werden mit Hilfe der Funktion `strcmp` verglichen und nicht mit der Booleschen Vergleichsoperation `==`. Mit der Hilfe der Funktion `strcmp` werden die Strings lexikographisch verglichen, und bei `==` die Adressen der zugehörigen Zeiger. Ein Vergleich zwischen zwei Zeigeradressen ergibt im Sinne von String-Vergleichen keinerlei Sinn.

Beenden von Programmen

- Wenn ein Programm ordnungsgemäß, also ohne einen Fehler, beendet wurde, dann sollte dies mit dem Returnwert `0` andernfalls `1` (bzw. `!=0`) der Funktion `main` signalisiert werden.
  - Im Allgemeinen verwendet man hierzu `return <WERT>;` in der Funktion `main` mit `<WERT>=0` im Erfolgsfall und andernfalls `<WERT> !=0`, typischerweise `1`.
- Eine Alternative hierzu ist es, mit der Funktion `void exit(int status)` aus `stdlib.h` zu arbeiten.
- Insbesondere wenn ein Programm innerhalb einer Funktion (nicht `main`) beendet werden soll, kann eine direkte Beendigung mittels der Funktion `exit` erfolgen.

return Anweisung

```c
int main (int argc, char *argv[]) {
    ...
    ...
    ...
    if (failure){
        return 1;
    }
    return 0;
}
```

exit Funktion

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

void f(){
    if (x==y){ //success
        exit(0); // oder exit(EXIT_SUCCESS)
    } else { //failure
        exit(1); // oder exit(EXIT_FAILURE)
    }
}

int main (int argc, char *argv[]) {
    f();
    return 0;
}
```