Lehrstuhl für Connected Mobility
School of Computation, Information and Technology
Technische Universität München

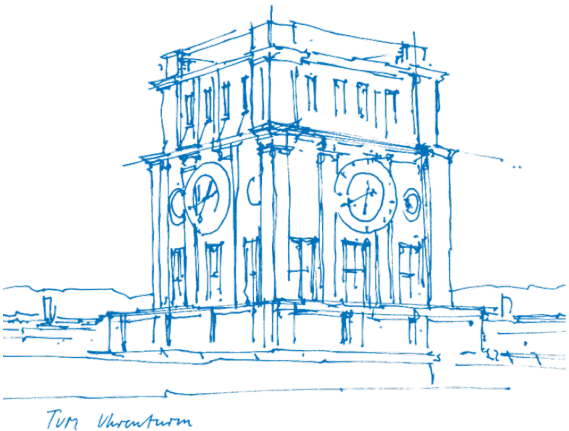
# Grundlagen: Betriebssysteme und Systemsoftware

## Tutorübung

**Mario Delic**

Lehrstuhl für Connected Mobility
School of Computation, Information and Technology
Technische Universität München

Übungswoche 1



# Generelle Agenda

- Kurzes Recap der wichtigsten Vorlesungsinhalte
- Darüber hinaus relevantes Wissen
- Eventuelle Fragen von euch

- Bearbeitung vom Übungsblatt

# Aufgabe 1

## Aufgabe 1 C the difference

**Vorbereitung**: Vorbereitend auf diese Aufgabe sollten Sie den C-Primer[1] von Jonas Pfoh lesen.

Die Programmiersprache C verhält sich in vielen Aspekten anders als die Ihnen bekannte Sprache Java. Ihr Tutor wird Ihnen anhand des folgenden Beispielprogramms zur Berechnung der Fakultät einige Grundlagen und Besonderheiten von C erläutern.

```java
import java.util.Scanner;

public class Fakultaet
{
  public static void main (String[] args)
  {
    Scanner scan = new Scanner(System.in);

    int fakultaet = fak(scan.nextInt());
    System.out.println("Fakultaet: " + fakultaet);
  }
  private static int fak(int x) {
      return x <= 1 ? 1 : (x*fak(x-1));
  }
}
```

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

int fak(int);

int main(int argc, char *argv[]) {
  char *buf = malloc(100 * sizeof(char));
  fgets(buf, 100, stdin);

  int fakultaet = fak(atoi(buf));
  printf("Fakultaet: %d\n", fakultaet);
  free(buf);
}

int fak(int x) {
  return x <= 1 ? 1 : (x*fak(x-1));
}
```

# Aufgabe 1

- public static void main (String[] args) $\leftrightarrow$ int main()
- import $\leftrightarrow$ #include
- OOP Zwang $\leftrightarrow$ OOP “Verbot“
- Funktionen müssen Vorausdeklariert werden, um sie vor der Definition nutzen zu können
- Java static: objektunabhängigkeit $\leftrightarrow$ C static: encapsulation/speicherort
- Automatischer Garbagecollector $\leftrightarrow$ Manuelles Memorymanagement
- **void* malloc(size_t size)**: Sucht nach freiem Speicher $\geq$ size und gibt Pointer auf diesen zurück
- **void free(void* p)**: Gibt Speicherplatz am Pointer p frei
- Auf jedes malloc kommt exakt ein free!
- malloc loops ohne free $\rightarrow$ Memory Leak! Use-After-Free $\rightarrow$ undefined behavior!

## Aufgabe 2 Binär- und Dezimalpräfixe

Einheitenpräfixe dienen dazu, Basiseinheiten zu skalieren. Ein bekanntes Beispiel hierfür ist *k* (Kilo) in km oder kg. Im Alltag werden häufig Dezimalpräfixe verwendet, die auf Potenzen der Zahl 10 basieren.

Im Kontext von Betriebssystemen werden jedoch häufig Binärpräfixe (Potenzen von 2) verwendet.

| Binär | | | Dezimal |
|---|---|---|---|
| Kibi | $1024 = 2^{10}$ | $1000 = 10^{3}$ | Kilo |
| Mebi | $2^{20}$ | $10^{6}$ | Mega |
| Gibi | $2^{30}$ | $10^{9}$ | Giga |
| Tebi | $2^{40}$ | $10^{12}$ | Tera |

# Aufgabe 2

a)* Übersetzen Sie von Binär- zu Dezimalpräfix: 2 KiB, 3 MiB, 4 GiB.

b)* Übersetzen Sie von Dezimal- zu Binärpräfix: 4 kB, 3 MB, 2 GB.

c)* Hersteller von Speichermedien preisen diese mit Kapazitäten an, die auf Dezimalpräfixen basieren. Das führt häufig zu Verwirrung, da Software meist Binärpräfixe verwendet, jedoch die falschen Einheiten anzeigt (z.B. GB statt GiB). Wie viel Speicherplatz wird dem Käufer einer externen 2 TB-Festplatte nach dem Anschließen an ein solches System angezeigt?

# Aufgabe 2

a)* Übersetzen Sie von Binär- zu Dezimalpräfix: 2 KiB, 3 MiB, 4 GiB.

2 KiB = 2 * 1024 B = 2048 B = 2,048 kB
3 MiB = 3 * $1024^2$ B = 3145728 B $\approx$ 3,145 MB
4 GiB = 4 * $1024^3$ B = 4294967296 B $\approx$ 4,294 GB

b)* Übersetzen Sie von Dezimal- zu Binärpräfix: 4 kB, 3 MB, 2 GB.

# Aufgabe 2

a)* Übersetzen Sie von Binär- zu Dezimalpräfix: 2 KiB, 3 MiB, 4 GiB.

2 KiB = 2 * 1024 B = 2048 B = 2,048 kB
3 MiB = 3 * $1024^2$ B = 3145728 B $\approx$ 3,145 MB
4 GiB = 4 * $1024^3$ B = 4294967296 B $\approx$ 4,294 GB

b)* Übersetzen Sie von Dezimal- zu Binärpräfix: 4 kB, 3 MB, 2 GB.

4 kB = 4 * 1000 B $\approx$ 3,906 KiB
3 MB = 3 * $1000^2$ B = 3000000 B $\approx$ 2929,687 KiB $\approx$ 2,861 MiB
2 GB = 2 * $1000^3$ B = 2000000000 B $\approx$ 1,862 GiB

# Aufgabe 2

c)* Hersteller von Speichermedien preisen diese mit Kapazitäten an, die auf Dezimalpräfixen basieren. Das führt häufig zu Verwirrung, da Software meist Binärpräfixe verwendet, jedoch die falschen Einheiten anzeigt (z.B. GB statt GiB). Wie viel Speicherplatz wird dem Käufer einer externen 2 TB-Festplatte nach dem Anschließen an ein solches System angezeigt?

# Aufgabe 2

c)* Hersteller von Speichermedien preisen diese mit Kapazitäten an, die auf Dezimalpräfixen basieren. Das führt häufig zu Verwirrung, da Software meist Binärpräfixe verwendet, jedoch die falschen Einheiten anzeigt (z.B. GB statt GiB). Wie viel Speicherplatz wird dem Käufer einer externen 2 TB-Festplatte nach dem Anschließen an ein solches System angezeigt?

1,82 TB, ihm „fehlen“ also knapp 10%

# Recap
## Binär- und Hexarithmethik

Dezimal:

- Basis 10: 0-9
- Wertigkeit der i-ten Stelle: $Ziffer * 10^i$
- Umrechnung zu Bin bzw. Hex: Division mit Rest durch 2 bzw. 16 und dann *von hinten zusammenbauen*

Binär

- Basis 2: 0-1
- Wertigkeit der i-ten Stelle: $Ziffer * 2^i$
- Umrechnung zu Dec durch addieren der Stellen: 0b11'0100 = $2^2+2^4+2^5 = 4+16+32$ = 52

Hex:

- Basis 16: 0-9 und A-F
- Wertigkeit der i-ten Stelle: $Ziffer * 16^i$
- Umrechnung zu Dec durch addieren der Stellen: 0xBA7 = $7*1+10*16+11*256 = 7+160+2816$ = 2983
- Umrechnung zu Bin direkt möglich da 1 Hex-Zeichen = 4 Bit: 0xC = 0b1100 = 12

# Zahlenentwicklung:

| Dezimal: | Binär: | Hex: |
|---|---|---|
| 1 | $2^0$ = 1 | 0x1 |
| 2 | $2^1$ = 10 | 0x2 |
| 4 | $2^2$ = 100 | 0x4 |
| 8 | $2^3$ = 1000 | 0x8 |
| 16 | $2^4$ = 1'0000 | 0x10 |
| 32 | $2^5$ = 10'0000 | 0x20 |
| 64 | $2^6$ = 100'0000 | 0x40 |
| 128 | $2^7$ = 1000'0000 | 0x80 |
| 256 | $2^8$ = 1'0000'0000 | 0x100 |
| 512 | $2^9$ = 10'0000'0000 | 0x200 |
| 1024 | $2^{10}$ = 100'0000'0000 | 0x400 |
| 2048 | $2^{11}$ = 1000'0000'0000 | 0x800 |
| 4096 | $2^{12}$ = 1'0000'0000'0000 | 0x1000 |

## Aufgabe 3 **Bin verHext**

Im Verlaufe dieser Veranstaltung werden Sie mit Werten in unterschiedlichen Basen umgehen müssen, sowie diese von einer in die andere Basis überführen. In GBS sind besonders die Basen 2, 10 und 16 relevant. Das Umrechnen von Zahlen zwischen diesen Basen sollten Sie unbedingt beherrschen.

Übersetzen Sie die gegebenen Werte von einer Basis in die Andere.

a)* Binär $\rightarrow$ Hex

0b101010
0b111000111
0b1100000011011110

## Aufgabe 3 **Bin verHext**

Im Verlaufe dieser Veranstaltung werden Sie mit Werten in unterschiedlichen Basen umgehen müssen, sowie diese von einer in die andere Basis überführen. In GBS sind besonders die Basen 2, 10 und 16 relevant. Das Umrechnen von Zahlen zwischen diesen Basen sollten Sie unbedingt beherrschen.

Übersetzen Sie die gegebenen Werte von einer Basis in die Andere.

a)* Binär $\rightarrow$ Hex

0b101010 $\rightarrow$ 0x2A
0b111000111 $\rightarrow$ 0x1C7
0b1100000011011110 $\rightarrow$ 0xC0DE

# Aufgabe 3

b)* Hex $\rightarrow$ Binär

0xAFFE
0xBADE
0xC0FFEE

c)* Dezimal $\rightarrow$ Hex

123
65
262

# Aufgabe 3

b)* Hex $\rightarrow$ Binär

0xAFFE $\rightarrow$ 0b1010111111111110
0xBADE $\rightarrow$ 0b1011101011011110
0xC0FFEE $\rightarrow$ 0b110000001111111111101110

c)* Dezimal $\rightarrow$ Hex

123
65
262

# Aufgabe 3

b)* Hex $\rightarrow$ Binär

0xAFFE $\rightarrow$ 0b1010111111111110
0xBADE $\rightarrow$ 0b1011101011011110
0xC0FFEE $\rightarrow$ 0b110000001111111111101110

c)* Dezimal $\rightarrow$ Hex

123 $\rightarrow$ 0x7B
65 $\rightarrow$ 0x41
262 $\rightarrow$ 0x106

# Aufgabe 3

d)* Dezimal → Binär

255
99
54

$2^8 - 1 = 0xFF = 0b\ 1111\ 1111$

$64 + 32 + 2 + 1 = 2^6 + 2^5 + 2^1 + 2^0 = 1100011$

6 5 1 0 → bits 65 10

e)* Hex → Dezimal

0xABC
0x64
0x420

# Aufgabe 3

d)* Dezimal $\rightarrow$ Binär

255 $\rightarrow$ 0b11111111
99 $\rightarrow$ 0b1100011
54 $\rightarrow$ 0b110110

e)* Hex $\rightarrow$ Dezimal

0xABC
0x64
0x420

# Aufgabe 3

d)* Dezimal $\rightarrow$ Binär

255 $\rightarrow$ 0b11111111
99 $\rightarrow$ 0b1100011
54 $\rightarrow$ 0b110110

e)* Hex $\rightarrow$ Dezimal

0xABC $\rightarrow$ 2748
0x64 $\rightarrow$ 100
0x420 $\rightarrow$ 1056

# *Pointer &Arithmetik

**Asterisk ***

In Initialisierugen: Hier steht ein Pointer
Als Paramenter: Gib mit einen Pointer
Als Präfixoperator: Gib mir den Wert, auf den dieser Pointer zeigt „Dereferenzierung"

**Ampersand &**

Als Präfixoperator: Gib mir eine Referenz (≅Pointer) auf diesen Wert

„Referenzierung"

```c
#include <stdio.h>
int main(){
    int a = 12;
    printf(" %d \n %x \n %p \n", a, a, &a);
    return 1;
}
```

```
12
c
00000024441ffd98
```

Output

# *Pointer &Arithmetik

```c
#include <stdio.h>
int afunc(int i){
    i+=5;
    return i;
}
int reffunc(int* i){
    *i+=5;
    return *i;
}
int main(){
    int a = 12;
    int* p = &a; //p = pointer auf a
    printf(" %d    \n %d        \n %d      \n %d       \n %d    \n",
           afunc(a), reffunc(&a), afunc(*p), reffunc(p), afunc(a));
    return 1;
}
```

a=12 a=17 a=17 a=22 a=22

```
17
17
22
22
27
```

Output

# *Pointer &Arithmetik

Pointer können inkrementiert/dekrementiert werden.
Pointer + 1 bedeutet dabei “1 Element des Pointertyps weiter.“

- char*: +1 erhöht die Adresse um genau 1, da char 1 Byte breit
- int*: +1 erhöht die Adresse um 4, da 4 Byte breit
- void*: **illegal**, muss zuerst gecasted werden z.B. mit (char*)

void* verhält sich wie char* und kann benutzt werden mit +/-, aber gibt compiler Warning

```
int main(){
    int* ip = 0;
    char* cp = 0;
    ip+=1;
    cp+=1;
    printf("Incremented int  pointer: %p \nIncremented char pointer: %p \n", ip, cp);
    return 1;
}
```

```
Incremented int  pointer: 0000000000000004
Incremented char pointer: 0000000000000001
```

# *Pointer &Arithmetik

```c
int main(){
    char abc[8] = {'a','b','c','d','e','f','g','h'};
    printf("abc value: %c \nabc address: %p \n",*abc, abc);
    int* ip = (int*)abc;
    printf("abc value: %c \nip value: %c \n", *(abc+1), *(ip+1));
    printf("abc address: %p \nip address: %p \n", abc+1, ip+1);
    return 1;
}
```

```
abc value: a
abc address: 0000007f7e3ffc04
abc value: b
ip value: e
abc address: 0000007f7e3ffc05
ip address: 0000007f7e3ffc08
```

## Aufgabe 4 Pointerarithmetik und Operatorpräzedenz

Die folgenden Deklarationen bilden die Grundlage für alle Teilaufgaben dieser Aufgabe. Verstehen Sie zunächst, welche Variablen deklariert und initialisiert werden, und welche Typen diese besitzen.

```
1  int arrayXYZ[10] = {0};
2  int i = 0, intVar = 0;
3  int *pi = 0;
4  for (i = 0; i < 10; i++)
5      arrayXYZ[i] = i;
```

**a)*** Was ist der Inhalt der Variable `pi` jeweils nach den folgenden Statements?

```
1  pi = &arrayXYZ[7];
2  pi = arrayXYZ;
3  pi = &arrayXYZ[0];
```

**b)*** Sind die folgenden Zuweisungen äquivalent? Verdeutlichen Sie sich, wie ein Array-Zugriff in C umgesetzt wird.

```
1  intVar = arrayXYZ[8];
2  intVar = *(arrayXYZ+8);
3  intVar = *(int *) ((void *) arrayXYZ+(8*sizeof(int)));
```

# Aufgabe 4

**c)*** Kompilieren die folgenden Statements ohne Warnings oder Errors? Wenn nicht, was ist das Problem? Können Fehler zur Laufzeit auftreten? Wie würden sich diese Fehler zur Laufzeit auswirken?

```
1    i = pi;
2    intVar = i;
3    *(arrayXYZ+3) = 3;
4    arrayXYZ[1320] = "a";
5    arrayXYZ[1] = &*(arrayXYZ + 15);
```

**d)*** Was macht der folgende Code? Wofür wird `malloc` in C benutzt?

```
int *array123 = malloc(5 * sizeof(int));
```

Warum sollten nicht alle Werte in Variablen abgelegt werden, die in der Funktion direkt deklariert wurden?

**e)** Welche Rolle spielt der Operator `sizeof()`? Wieso wird an `malloc()` nicht 5 als Parameter übergeben?

**f)** Was hat es mit `free()` auf sich? Inwiefern verhält sich Java hier anders als C, wieso musste man diese Funktion in Java nicht nutzen?