

# Angewandte Informatik

## Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

8. Januar 2018

# Angewandte Informatik

## Hardwarenahe Programmierung

<https://gitlab.cvh-server.de/pgerwinski/hp.git>

- 1 Einführung**
- 2 Einführung in C**
- 3 Bibliotheken**
- 4 Hardwarenahe Programmierung**
  - 4.1** Bit-Operationen
  - 4.2** I/O-Ports
  - 4.3** Interrupts
  - 4.4** volatile-Variable
  - 4.5** Byte-Reihenfolge – Endianness
  - 4.6** Speicherausrichtung – Alignment
- 5 Algorithmen**
  - 5.1** Differentialgleichungen
  - 5.2** Rekursion
  - 5.3** Aufwandsabschätzungen

...

## 4 Hardwarenahe Programmierung

### 4.1 Bit-Operationen

#### 4.1.1 Zahlensysteme

Basis	Name	Beispiel	Anwendung
2	Binärsystem	1 0000 0011	Bit-Operationen
8	Oktalsystem	0403	Dateizugriffsrechte (Unix)
10	Dezimalsystem	259	Alltag
16	Hexadezimalsystem	0x103	Bit-Operationen
256	(keiner gebräuchlich)	0.0.1.3	IP-Adressen (IPv4)

### 4.1.1 Zahlensysteme

Oktal- und Hexadezimal-Zahlen lassen sich ziffernweise in Binär-Zahlen umrechnen:

000	0	0000	0	1000	8
001	1	0001	1	1001	9
010	2	0010	2	1010	A
011	3	0011	3	1011	B
100	4	0100	4	1100	C
101	5	0101	5	1101	D
110	6	0110	6	1110	E
111	7	0111	7	1111	F

## 4.1.2 Bit-Operationen in C

C-Operator	Verknüpfung	Anwendung
&	Und	Bits gezielt löschen
	Oder	Bits gezielt setzen
^	Exklusiv-Oder	Bits gezielt invertieren
~	Nicht	Alle Bits invertieren
<<	Verschiebung nach links	Maske generieren
>>	Verschiebung nach rechts	Bits isolieren

Numerierung der Bits: von rechts ab 0

Bit Nr. 3 auf 1 setzen: `a |= 1 << 3;`

Bit Nr. 4 auf 0 setzen: `a &= ~(1 << 4);`

Bit Nr. 0 invertieren: `a ^= 1 << 0;`

Abfrage, ob Bit Nr. 1 gesetzt ist: `if (a & (1 << 1))`

## 4.2 I/O-Ports

Aus Input-Port lesen = Sensoren abfragen

Beispiel: Taster

```
#include <avr/io.h>
```

```
...
```

```
DDRC = 0xfd;          binär: 1111 1101
```

```
while (PINC & 0x02 == 0) binär: 0000 0010
```

```
; /* just wait */
```

**Fehler!**

Herstellerspezifisch!

DDR = Data Direction Register

Bit = 1 für Output-Port

Bit = 0 für Input-Port

*Details: siehe Datenblatt und Schaltplan*

Praktikumsaufgabe: Druckknopfampel

## 4.2 I/O-Ports

Aus Input-Port lesen = Sensoren abfragen

Beispiel: Taster

```
#include <avr/io.h>
```

```
...
```

```
DDRC = 0xfd;          binär: 1111 1101
```

```
while ((PINC & 0x02) == 0) binär: 0000 0010
```

```
; /* just wait */
```

== hat Vorrang vor &

Herstellerspezifisch!

DDR = Data Direction Register

Bit = 1 für Output-Port

Bit = 0 für Input-Port

*Details: siehe Datenblatt und Schaltplan*

Praktikumsaufgabe: Druckknopfampel

## 4.4 volatile-Variable

Externes Gerät ruft (per Stromsignal) Unterprogramm auf  
Zeiger hinterlegen: „Interrupt-Vektor“  
Beispiel: Taster

```
#include <avr/interrupt.h>
```

```
...
```

```
volatile uint8_t key_pressed = 0;
```

```
ISR (INT0_vect)
```

```
{  
    key_pressed = 1;  
}
```

```
int main (void)
```

```
{
```

```
...
```

```
while (1)
```

```
{
```

```
    while (!key_pressed)
```

```
        ; /* just wait */
```

```
        PORTD ^= 0x40;
```


```
        key_pressed = 0;
```

```
    }
```

```
    return 0;
```

```
}
```

**volatile:**  
Speicherzugriff  
nicht wegoptimieren





## 4.4 volatile-Variable

Was ist eigentlich PORTD?

```
avr-gcc -std=c99 -Wall -Os -mmcu=atmega328p blink-3.c -E
```

PORTD = 0x01;

→ `(*(volatile uint8_t *) ((0x0B) + 0x20)) = 0x01;`

Umwandlung in Zeiger  
auf **volatile** uint8\_t

Zahl: 0x2B

Dereferenzierung des Zeigers

→ **volatile** uint8\_t-Variable an Speicheradresse 0x2B

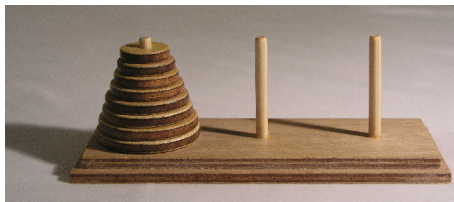
→ `PORTA = PORTB = PORTC = PORTD = 0` ist eine schlechte Idee.

## 5.2 Rekursion

Vollständige Induktion:  $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

### Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.
- $n = 1$  Scheibe: fertig
- Wenn  $n - 1$  Scheiben verschiebbar: schiebe  $n - 1$  Scheiben auf Hilfsplatz, verschiebe die darunterliegende, hole  $n - 1$  Scheiben von Hilfsplatz



## 5.2 Rekursion

Vollständige Induktion:  $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

### Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.
- $n = 1$  Scheibe: fertig
- Wenn  $n - 1$  Scheiben verschiebbar: schiebe  $n - 1$  Scheiben auf Hilfsplatz, verschiebe die darunterliegende, hole  $n - 1$  Scheiben von Hilfsplatz

```
void move (int from, int to, int disks)
{
    if (disks == 1)
        move_one_disk (from, to);
    else
    {
        int help = 0 + 1 + 2 - from - to;
        move (from, help, disks - 1);
        move (from, to, 1);
        move (help, to, disks - 1);
    }
}
```

## 5.2 Rekursion

Vollständige Induktion:  $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

### Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.
- $n = 1$  Scheibe: fertig
- Wenn  $n - 1$  Scheiben verschiebbar: schiebe  $n - 1$  Scheiben auf Hilfsplatz, verschiebe die darunterliegende, hole  $n - 1$  Scheiben von Hilfsplatz

32 Scheiben:

```
$ time ./hanoi-9a
```

```
...
```

```
real      0m32,712s
```

```
user      0m32,708s
```

```
sys       0m0,000s
```

→ etwas über 1 Minute  
für 64 Scheiben

## 5.2 Rekursion

Vollständige Induktion:  $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

### Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.
- $n = 1$  Scheibe: fertig
- Wenn  $n - 1$  Scheiben verschiebbar: schiebe  $n - 1$  Scheiben auf Hilfsplatz, verschiebe die darunterliegende, hole  $n - 1$  Scheiben von Hilfsplatz

32 Scheiben:

```
$ time ./hanoi-9a
...
real      0m32,712s
user      0m32,708s
sys       0m0,000s
```

~~→ etwas über 1 Minute  
für 64 Scheiben~~

Für jede zusätzliche Scheibe verdoppelt sich die Rechenzeit!

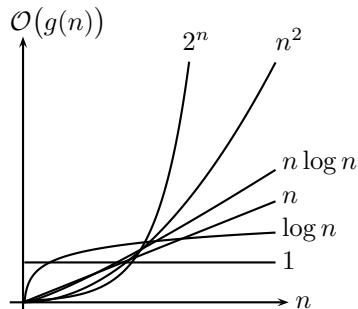
→  $\frac{32,712 \text{ s} \cdot 2^{32}}{3600 \cdot 24 \cdot 365,25} \approx 4452 \text{ Jahre}$   
für 64 Scheiben

## 5.3 Aufwandsabschätzungen – Komplexitätsanalyse

- Türme von Hanoi:  $\mathcal{O}(2^n)$

Für jede zusätzliche Scheibe verdoppelt sich die Rechenzeit!

→  $\frac{32,712 \text{ s} \cdot 2^{32}}{3600 \cdot 24 \cdot 365,25} \approx 4452 \text{ Jahre}$   
für 64 Scheiben

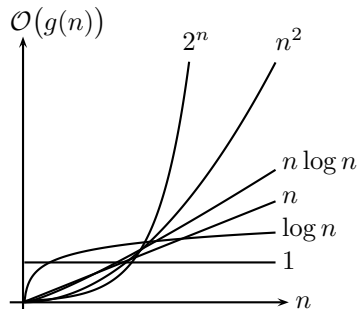


$n$ : Eingabedaten

$g(n)$ : Rechenzeit

## 5.3 Aufwandsabschätzungen – Komplexitätsanalyse

- Türme von Hanoi:  $\mathcal{O}(2^n)$
- Minimum suchen:  $\mathcal{O}(n)$
- ... mit Schummeln:  $\mathcal{O}(1)$
- Minimum an den Anfang tauschen, nächstes Minimum suchen:  
→ Selectionsort



$n$ : Eingabedaten

$g(n)$ : Rechenzeit

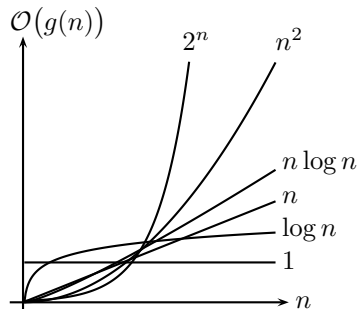
Faustregel:

Schachtelung der Schleifen zählen

$x$  Schleifen →  $\mathcal{O}(n^x)$

## 5.3 Aufwandsabschätzungen – Komplexitätsanalyse

- Türme von Hanoi:  $\mathcal{O}(2^n)$
- Minimum suchen:  $\mathcal{O}(n)$
- ... mit Schummeln:  $\mathcal{O}(1)$
- Minimum an den Anfang tauschen, nächstes Minimum suchen:  
→ Selectionsort:  $\mathcal{O}(n^2)$



$n$ : Eingabedaten

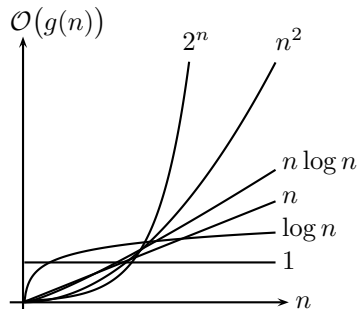
$g(n)$ : Rechenzeit

Faustregel:  
Schachtelung der Schleifen zählen  
 $x$  Schleifen  $\rightarrow \mathcal{O}(n^x)$



## 5.3 Aufwandsabschätzungen – Komplexitätsanalyse

- Türme von Hanoi:  $\mathcal{O}(2^n)$
- Minimum suchen:  $\mathcal{O}(n)$
- ... mit Schummeln:  $\mathcal{O}(1)$
- Minimum an den Anfang tauschen, nächstes Minimum suchen:  
→ Selectionsort:  $\mathcal{O}(n^2)$
- Während Minimumsuche prüfen und abbrechen, falls schon sortiert  
→ Bubblesort:  $\mathcal{O}(n)$  bis  $\mathcal{O}(n^2)$



$n$ : Eingabedaten

$g(n)$ : Rechenzeit

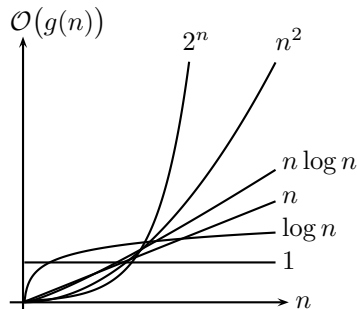
Faustregel:

Schachtelung der Schleifen zählen

$x$  Schleifen  $\rightarrow \mathcal{O}(n^x)$

## 5.3 Aufwandsabschätzungen – Komplexitätsanalyse

- Türme von Hanoi:  $\mathcal{O}(2^n)$
- Minimum suchen:  $\mathcal{O}(n)$
- ... mit Schummeln:  $\mathcal{O}(1)$
- Minimum an den Anfang tauschen, nächstes Minimum suchen:  
→ Selectionsort:  $\mathcal{O}(n^2)$
- Während Minimumsuche prüfen und abbrechen, falls schon sortiert  
→ Bubblesort:  $\mathcal{O}(n)$  bis  $\mathcal{O}(n^2)$
- Rekursiv sortieren  
→ Quicksort



$n$ : Eingabedaten

$g(n)$ : Rechenzeit

Faustregel:

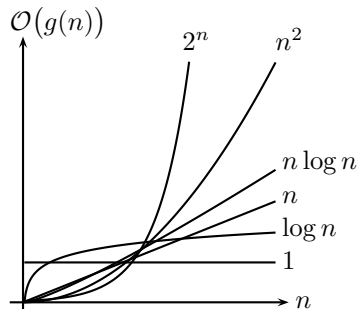
Schachtelung der Schleifen zählen

$x$  Schleifen  $\rightarrow \mathcal{O}(n^x)$

## 5.3 Aufwandsabschätzungen – Komplexitätsanalyse

- Türme von Hanoi:  $\mathcal{O}(2^n)$
- Minimum suchen:  $\mathcal{O}(n)$
- ... mit Schummeln:  $\mathcal{O}(1)$
- Minimum an den Anfang tauschen, nächstes Minimum suchen:  
→ Selectionsort:  $\mathcal{O}(n^2)$
- Während Minimumsuche prüfen und abbrechen, falls schon sortiert  
→ Bubblesort:  $\mathcal{O}(n)$  bis  $\mathcal{O}(n^2)$
- Rekursiv sortieren  
→ Quicksort:  $\mathcal{O}(n \log n)$  bis  $\mathcal{O}(n^2)$

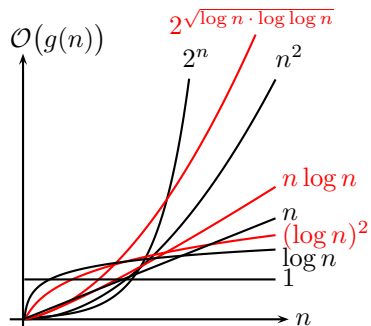
Faustregel:  
Schachtelung der Schleifen zählen  
 $x$  Schleifen →  $\mathcal{O}(n^x)$



$n$ : Eingabedaten  
 $g(n)$ : Rechenzeit

## 5.3 Aufwandsabschätzungen – Komplexitätsanalyse

- Türme von Hanoi:  $\mathcal{O}(2^n)$
- Minimum suchen:  $\mathcal{O}(n)$
- ... mit Schummeln:  $\mathcal{O}(1)$
- Minimum an den Anfang tauschen, nächstes Minimum suchen:  
→ Selectionsort:  $\mathcal{O}(n^2)$
- Während Minimumsuche prüfen und abbrechen, falls schon sortiert  
→ Bubblesort:  $\mathcal{O}(n)$  bis  $\mathcal{O}(n^2)$
- Rekursiv sortieren  
→ Quicksort:  $\mathcal{O}(n \log n)$  bis  $\mathcal{O}(n^2)$



$n$ : Eingabedaten

$g(n)$ : Rechenzeit

Faustregel:

Schachtelung der Schleifen zählen

$x$  Schleifen  $\rightarrow \mathcal{O}(n^x)$

**RSA**: Schlüsselerzeugung (Berechnung von  $d$ ):  $\mathcal{O}((\log n)^2)$ ,

Ver- und Entschlüsselung (Exponentiation):  $\mathcal{O}(n \log n)$ ,

Verschlüsselung brechen (Primfaktorzerlegung):  $\mathcal{O}(2^{\sqrt{\log n \cdot \log \log n}})$

# Angewandte Informatik

## Hardwarenahe Programmierung

<https://gitlab.cvh-server.de/pgerwinski/hp.git>

- 1 Einführung**
- 2 Einführung in C**
- 3 Bibliotheken**
- 4 Hardwarenahe Programmierung**
- 5 Algorithmen**
  - 5.1** Differentialgleichungen
  - 5.2** Rekursion
  - 5.3** Aufwandsabschätzungen
- 6 Objektorientierte Programmierung**
  - 6.0** Dynamische Speicherverwaltung
  - 6.1** Konzepte und Ziele
  - 6.2** Beispiel: Zahlen und Buchstaben
  - 6.3** Beispiel: Graphische Benutzeroberfläche (GUI)
  - ...
- 7 Datenstrukturen**

# 6 Objektorientierte Programmierung

## 6.0 Dynamische Speicherverwaltung

- Array: feste Anzahl von Elementen desselben Typs (z. B. 3 ganze Zahlen)
- Dynamisches Array: variable Anzahl von Elementen desselben Typs

```
char *name[] = { "Anna", "Berthold", "Caesar" };
```

...

~~name[3] = "Dieter";~~

# 6 Objektorientierte Programmierung

## 6.0 Dynamische Speicherverwaltung

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

```
...
```

```
char **name = malloc (3 * sizeof (char *));  
    /* Speicherplatz für 3 Zeiger anfordern */
```

```
...
```

```
free (name)  
    /* Speicherplatz freigeben */
```

# 6 Objektorientierte Programmierung

## 6.1 Konzepte und Ziele

- Array: feste Anzahl von Elementen desselben Typs (z. B. 3 ganze Zahlen)
- Dynamisches Array: variable Anzahl von Elementen desselben Typs
- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern
- Funktionen, die mit dem Objekt arbeiten: *Methoden*
- Was die Funktion bewirkt, hängt vom Typ des Objekts ab
- Realisierung über endlose **if**-Ketten



# 6 Objektorientierte Programmierung

## 6.1 Konzepte und Ziele

- Array: feste Anzahl von Elementen desselben Typs (z. B. 3 ganze Zahlen)
- Dynamisches Array: variable Anzahl von Elementen desselben Typs
- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern
- Funktionen, die mit dem Objekt arbeiten: *Methoden*
- ~~Was die Funktion bewirkt~~ Welche Funktion aufgerufen wird, hängt vom Typ des Objekts ab: *virtuelle Methode*
- Realisierung über ~~endlose if-Ketten~~ Zeiger, die im Objekt gespeichert sind (Genaugenommen: Tabelle von Zeigern)

→ **kommt demnächst**

# 6 Objektorientierte Programmierung

## 6.1 Konzepte und Ziele

- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern
- *Methoden* und *virtuelle Methoden*
- Zeiger auf verschiedene Strukturen mit einem gemeinsamen Anteil von Datenfeldern  
→ „verwandte“ *Objekte*, *Klassen* von Objekten
- Struktur, die *nur* den gemeinsamen Anteil enthält  
→ „Vorfahr“, *Basisklasse*, *Vererbung*
- Zeiger auf die Basisklasse dürfen auf Objekte der *abgeleiteten Klasse* zeigen  
→ *Polymorphie*

# 6 Objektorientierte Programmierung

## 6.2 Beispiel: Zahlen und Buchstaben

```
typedef struct
{
    int type;
} t_base;
```

```
typedef struct
{
    int type;
    int content;
} t_integer;
```

```
typedef struct
{
    int type;
    char *content;
} t_string;
```

```
typedef struct
{
    int type;
} t_base;
```

```
typedef struct
{
    int type;
    int content;
} t_integer;
```

```
typedef struct
{
    int type;
    char *content;
} t_string;
```

```
t_integer i = { 1, 42 };
t_string s = { 2, "Hello,_world!" };
```

```
t_base *object[] = { (t_base *) &i, (t_base *) &s };
```



explizite

Typumwandlung

## 6.3 Beispiel: Graphische Benutzeroberfläche (GUI)

```
#include <gtk/gtk.h>
```

```
int main (int argc, char **argv)
```

```
{
    gtk_init (&argc, &argv);
    GtkWidget *window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Hello");
    g_signal_connect (window, "destroy", G_CALLBACK (gtk_main_quit), NULL);
    GtkWidget *vbox = gtk_box_new (GTK_ORIENTATION_VERTICAL, 5);
    gtk_container_add (GTK_CONTAINER (window), vbox);
    gtk_container_set_border_width (GTK_CONTAINER (vbox), 10);
    GtkWidget *label = gtk_label_new ("Hello, world!");
    gtk_container_add (GTK_CONTAINER (vbox), label);
    GtkWidget *button = gtk_button_new_with_label ("Quit");
    g_signal_connect (button, "clicked", G_CALLBACK (gtk_main_quit), NULL);
    gtk_container_add (GTK_CONTAINER (vbox), button);
    gtk_widget_show (button);
    gtk_widget_show (label);
    gtk_widget_show (vbox);
    gtk_widget_show (window);
    gtk_main ();
    return 0;
}
```



**Praktikumsversuch:  
Objektorientiertes Zeichenprogramm**

# Angewandte Informatik

## Hardwarenahe Programmierung

<https://gitlab.cvh-server.de/pgerwinski/hp.git>

- 1 Einführung**
- 2 Einführung in C**
- 3 Bibliotheken**
- 4 Hardwarenahe Programmierung**
- 5 Algorithmen**
- 6 Objektorientierte Programmierung**
  - 6.0 Dynamische Speicherverwaltung
  - 6.1 Konzepte und Ziele
  - 6.2 Beispiel: Zahlen und Buchstaben
  - 6.3 Beispiel: Graphische Benutzeroberfläche (GUI)
  - 6.4 Unions
  - 6.5 Virtuelle Methoden
  - 6.6 Einführung in C++
- 7 Datenstrukturen**