

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

9. Januar 2020

Hardwarenahe Programmierung

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

1 Einführung

2 Einführung in C

3 Bibliotheken

4 Hardwarenahe Programmierung

...

4.4 volatile-Variable

4.6 Byte-Reihenfolge – Endianness

4.7 Binärdarstellung negativer Zahlen

4.8 Speicherausrichtung – Alignment

5 Algorithmen

5.1 Differentialgleichungen

5. $\frac{1+i}{\sqrt{2}}$ Quantencomputer

5.2 Rekursion

5.3 Aufwandsabschätzungen

...

...

4.6 Byte-Reihenfolge – Endianness

4.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.
Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\ 0100\ 0000\ 0011_2 = 0403_{16}$$

Speicherzellen:

04	03
----	----

 Big-Endian „großes Ende zuerst“
für Menschen leichter lesbar

03	04
----	----

 Little-Endian „kleines Ende zuerst“
bei Additionen effizienter

→ Geschmackssache ... **außer bei Datenaustausch!**

4.7 Binärdarstellung negativer Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

8-Bit-Zahlen ohne Vorzeichen: `uint8_t`

→ Zahlenwerte von `0x00` bis `0xff` = 0 bis 255

→ $255 + 1 = 0$

8-Bit-Zahlen mit Vorzeichen: `int8_t`

`0xff` = 255 ist die „natürliche“ Schreibweise für -1 .

→ Zweierkomplement

Oberstes Bit = 1: negativ

Oberstes Bit = 0: positiv

→ $127 + 1 = -128$

4.7 Binärdarstellung negativer Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

16-Bit-Zahlen ohne Vorzeichen: `uint16_t`

→ Zahlenwerte von `0x0000` bis `0xffff` = 0 bis 65535

→ $65535 + 1 = 0$

`uint8_t`

0 bis 255

$255 + 1 = 0$

16-Bit-Zahlen mit Vorzeichen: `int16_t`

`0xffff` = 65535 ist die „natürliche“ Schreibweise für -1 .

→ Zweierkomplement

`int8_t`

`0xff` = 255 = -1

Oberstes Bit = 1: negativ

Oberstes Bit = 0: positiv

→ $32767 + 1 = -32768$

Literatur: <http://xkcd.com/571/>

4.7 Binärdarstellung negativer Zahlen

Frage: Für welche Zahl steht der Speicherinhalt

a3	90
----	----

 (hexadezimal)?

Antwort: Das kommt darauf an. ;–)

Little-Endian:

als <code>int8_t</code> :	–93	(nur erstes Byte)
als <code>uint8_t</code> :	163	(nur erstes Byte)
als <code>int16_t</code> :	–28509	
als <code>uint16_t</code> :	37027	
<code>int32_t</code> oder größer:	37027	(zusätzliche Bytes mit Nullen aufgefüllt)

Big-Endian:

als <code>int8_t</code> :	–93	(nur erstes Byte)
als <code>uint8_t</code> :	163	(nur erstes Byte)
als <code>int16_t</code> :	–23664	
als <code>uint16_t</code> :	41872	
als <code>int32_t</code> :	–1550843904	(zusätzliche Bytes mit Nullen aufgefüllt)
als <code>uint32_t</code> :	2744123392	
als <code>int64_t</code> :	–6660823848880963584	
als <code>uint64_t</code> :	11785920224828588032	

4.8 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- ... oder sogar nur dann erlaubt

→ Compiler optimiert Speicherausrichtung

```
uint8_t a;  
uint8_t dummy;  
uint16_t b;  
uint8_t c;  
  
uint8_t a;  
uint8_t c;  
uint16_t b;
```

Fazit:

- **Adressen von Variablen sind systemabhängig**
- Bei Definition von Datenformaten Alignment beachten → effizienter

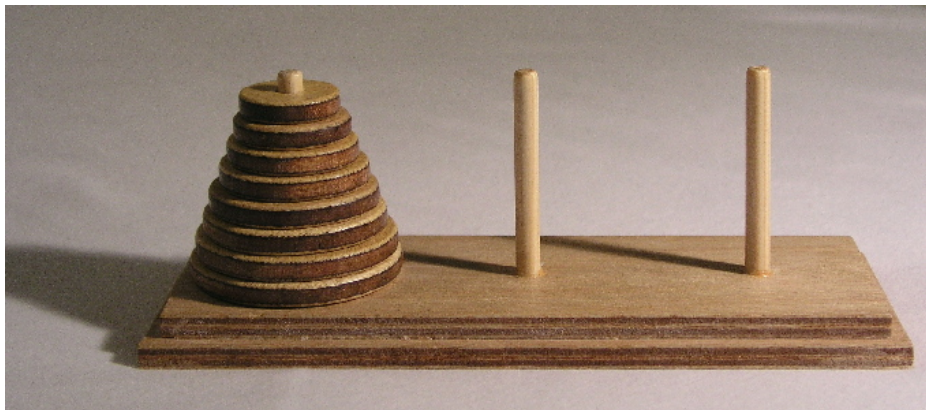
5.2 Rekursion

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

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

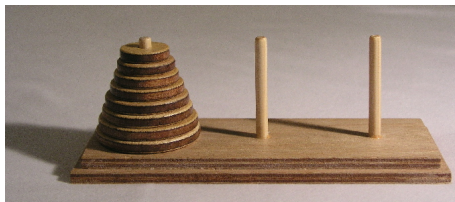


5.2 Rekursion

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.

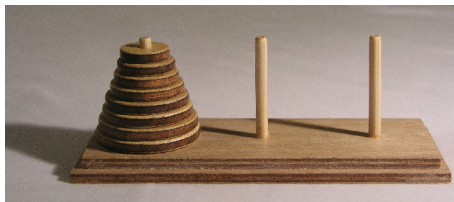


5.2 Rekursion

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
```

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

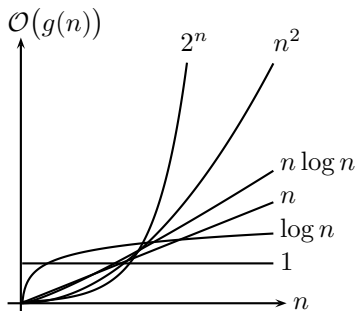
5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

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

Wann ist ein Programm „schnell“?

Türme von Hanoi: $\mathcal{O}(2^n)$

Für jede zusätzliche Scheibe
verdoppelt sich die Rechenzeit!

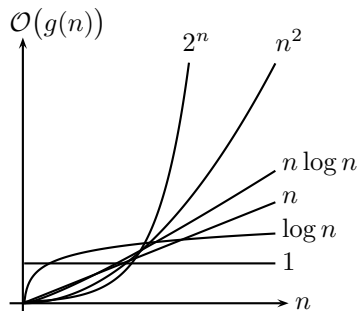
$$\rightarrow \frac{32,712 \text{ s} \cdot 2^{32}}{3600 \cdot 24 \cdot 365,25} \approx 4452 \text{ Jahre}$$

für 64 Scheiben

Faustregel:

Schachtelung der Schleifen zählen

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



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

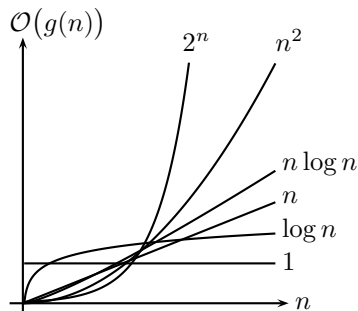
Schachtelung der Schleifen zählen

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen mit Schummeln: $\mathcal{O}(1)$
- Maximum suchen: $\mathcal{O}(n)$
- Selection-Sort: $\mathcal{O}(n^2)$
- Bubble-Sort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$
- Quicksort: $\mathcal{O}(n \log n)$ bis $\mathcal{O}(n^2)$



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

Schachtelung der Schleifen zählen

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

Wie schnell ist RSA-Verschlüsselung?

$c = m^e \% N$ („%“ = „modulo“)

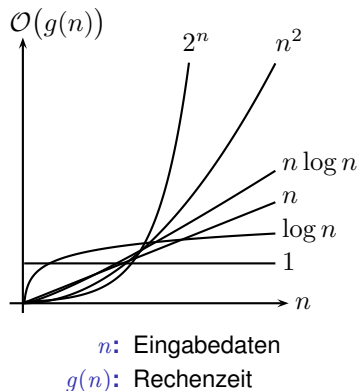
```
int c = 1;
for (int i = 0; i < e; i++)
    c = (c * m) % N;
```

- $\mathcal{O}(e)$ Iterationen
- mit Trick: $\mathcal{O}(\log e)$ Iterationen ($\log e$ = Anzahl der Ziffern von e)

Jede Iteration enthält eine Multiplikation und eine Division.

Aufwand dafür: $\mathcal{O}(\log e)$

\rightarrow Gesamtaufwand: $\mathcal{O}((\log e)^2)$



5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

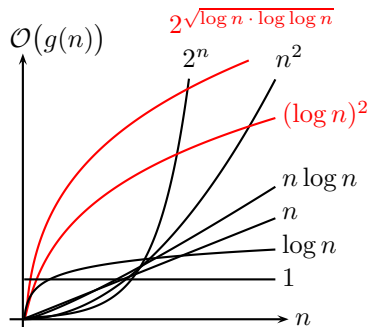
Schachtelung der Schleifen zählen

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

Wie schnell ist RSA?

(n = typische beteiligte Zahl, z. B. e, p, q)

- Ver- und Entschlüsselung (Exponentiation):
 $\mathcal{O}((\log n)^2)$
- Schlüsselerzeugung (Berechnung von d):
 $\mathcal{O}((\log n)^2)$
- Verschlüsselung brechen (Primfaktorzerlegung):
 $\mathcal{O}(2^{\sqrt{\log n \cdot \log \log n}})$



n : Eingabedaten

$g(n)$: Rechenzeit

Die Sicherheit von RSA beruht darauf, daß das Brechen der Verschlüsselung aufwendiger ist als $\mathcal{O}((\log n)^k)$ (für beliebiges k).

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

Schachtelung der Schleifen zählen

k Schleifen ineinander $\rightarrow O(n^k)$

Wie schnell ist RSA?

(n = typische beteiligte Zahl, z. B. e, p, q)

- Ver- und Entschlüsselung (Exponentiation):

$$O((\log n)^2)$$

$$O(n^2)$$

- Schlüsselerzeugung (Berechnung von d):

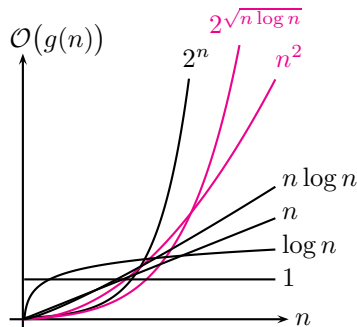
$$O((\log n)^2)$$

$$O(n^2)$$

- Verschlüsselung brechen (Primfaktorzerlegung):

$$O(2^{\sqrt{\log n \cdot \log \log n}})$$

$$O(2^{\sqrt{n \log n}})$$



n : Eingabedaten

$g(n)$: Rechenzeit

Die Sicherheit von RSA beruht darauf, daß das Brechen der Verschlüsselung aufwendiger ist als $O((\log n)^k)$ (für beliebiges k).

6 Objektorientierte Programmierung

6.1 Konzepte und Ziele

- Array: Elemente desselben Typs (z. B. 3 ganze Zahlen)
- 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: Elemente desselben Typs (z. B. 3 ganze Zahlen)
- 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)

→ nächste Woche

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