

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

28. November 2022

Hardwarenahe Programmierung

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

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 Unions

6.4 Virtuelle Methoden

6.5 Beispiel: Graphische Benutzeroberfläche (GUI)

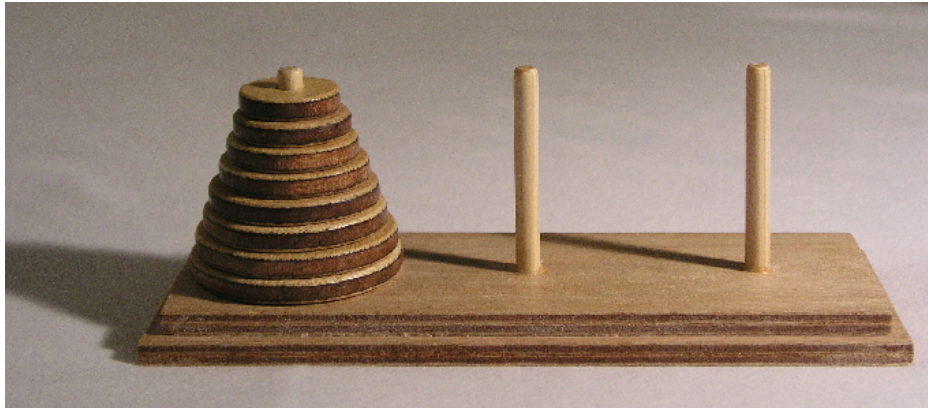
6.6 Ausblick: C++

7 Datenstrukturen

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

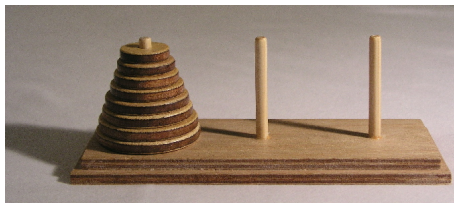


5.2 Rekursion

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.
- $n = 1$ 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-9b
...
real      0m30,672s
user      0m30,662s
sys       0m0,008s
```

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

Für jede zusätzliche Scheibe verdoppelt sich die Rechenzeit!

→ $\frac{30,672 \text{ s} \cdot 2^{32}}{3600 \cdot 24 \cdot 365,25} \approx 4174 \text{ Jahre}$
für 64 Scheiben

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Türme von Hanoi: $\mathcal{O}(2^n)$

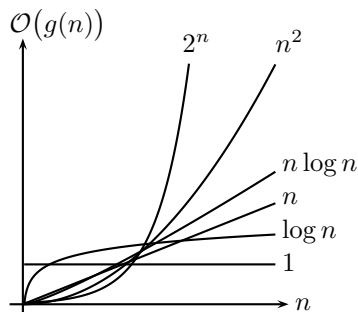
Für jede zusätzliche Scheibe
verdoppelt sich die Rechenzeit!

$$\rightarrow \frac{30,672 \text{ s} \cdot 2^{32}}{3600 \cdot 24 \cdot 365,25} \approx 4174 \text{ Jahre}$$

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“?

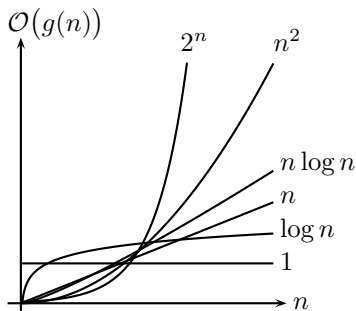
Faustregel:

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“)

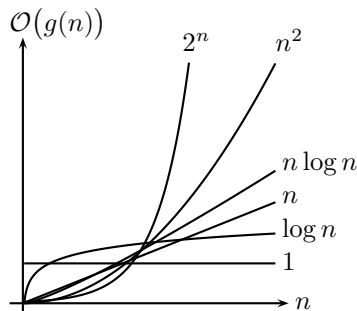
```
int c = 1;  
for (int i = 0; i < e; i++)  
    c = (c * m) % N;
```

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



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

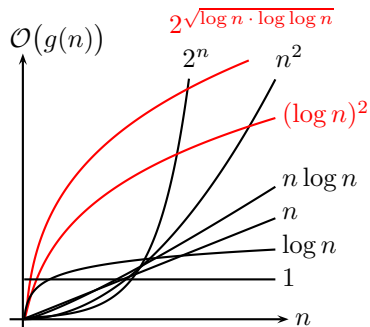
Schachtelung der Schleifen zählen

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

Wie schnell ist RSA?

(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):

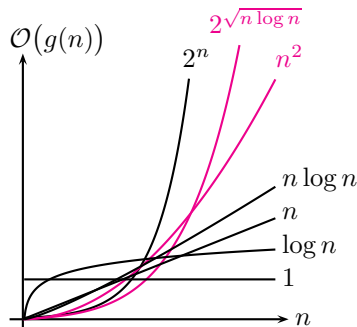
$$O((\log n)^2)$$

$$O(n^2)$$

- 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

6.0 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: Elemente desselben Typs (z. B. 3 ganze Zahlen)
- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern → *Objekt*
- Problem: Die Elemente sind unterschiedlich groß (Speicherplatz).
- Lösung: Im Array nicht die Objekte selbst speichern, sondern Zeiger darauf.
- 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 → *Objekt*
- Problem: Die Elemente sind unterschiedlich groß (Speicherplatz).
- Lösung: Im Array nicht die Objekte selbst speichern, sondern Zeiger darauf.
- 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 gleich**

6 Objektorientierte Programmierung

6.1 Konzepte und Ziele

- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern → *Objekt*
- *Methoden* und *virtuelle Methoden*
- Zeiger auf verschiedene Strukturen mit einem gemeinsamen Anteil von Datenfeldern
→ „verwandte“ *Objekte*, *Klassenhierarchie* 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 Unions

Variable teilen sich denselben Speicherplatz.

```
typedef union
```

```
{  
    int8_t i;  
    uint8_t u;  
} num8_t;
```

```
int main (void)
```

```
{  
    num8_t test;  
    test.i = -1;  
    printf ("%d\n", test.u);  
    return 0;  
}
```

6.3 Unions

Variable teilen sich denselben Speicherplatz.

```
typedef union
```

```
{  
    char s[8];  
    uint64_t x;  
} num_char_t;
```

```
int main (void)
```

```
{  
    num_char_t test = { "Hello!" };  
    printf ("%lx\n", test.x);  
    return 0;  
}
```

6.3 Unions

Variable teilen sich denselben Speicherplatz.

typedef union

```
{  
    t_base base;  
    t_integer integer;  
    t_string string;  
} t_object;
```

typedef struct

```
{  
    int type;  
} t_base;
```

typedef struct

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

typedef struct

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

```
if (this->base.type == T_INTEGER)  
    printf ("Integer:_%d\n", this->integer.content);  
else if (this->base.type == T_STRING)  
    printf ("String:_%s\n", this->string.content);
```

6.4 Virtuelle Methoden

```
void print_object (t_object *this)
{
    if (this->base.type == T_INTEGER)
        printf ("Integer:_%d\n", this->integer.content);
    else if (this->base.type == T_STRING)
        printf ("String:_%s\n", this->string.content);
}
```

if-Kette:
wird unübersichtlich



Zeiger auf Funktionen

```
void print_integer (t_object *this)
{
    printf ("Integer:_%d\n", this->integer.content);
}
```

```
void print_string (t_object *this)
{
    printf ("String:_%s\n", this->string.content);
}
```

6.4 Virtuelle Methoden

Zeiger auf Funktionen

```
void (* print) (t_object *this);
```


das, worauf print zeigt,
ist eine Funktion

- Objekt enthält Zeiger auf Funktion

```
typedef struct  
{  
    void (* print) (union t_object *this);  
    int content;  
} t_integer;
```

6.4 Virtuelle Methoden

Zeiger auf Funktionen

```
void (* print) (t_object *this);
```


das, worauf print zeigt,
ist eine Funktion

- Objekt enthält Zeiger auf Funktion
- Konstruktor initialisiert diesen Zeiger

```
t_object *new_integer (int i)
{
    t_object *p = malloc (sizeof (t_integer));
    p->integer.print = print_integer;
    p->integer.content = i;
    return p;
}
```

```
typedef struct
```

```
{
    void (* print) (union t_object *this);
    int content;
} t_integer;
```


6.4 Virtuelle Methoden

Zeiger auf Funktionen

```
void (* print) (t_object *this);
```


das, worauf print zeigt,
ist eine Funktion

- Objekt enthält Zeiger auf Funktion
- Konstruktor initialisiert diesen Zeiger
- Aufruf: „automatisch“ die richtige Funktion

```
for (int i = 0; object[i]; i++)  
    object[i]—>base.print (object[i]);
```

```
typedef struct  
{  
    void (* print) (union t_object *this);  
    int content;  
} t_integer;
```

6.4 Virtuelle Methoden

Zeiger auf Funktionen

```
void (* print) (t_object *this);
```


das, worauf print zeigt,
ist eine Funktion

- Objekt enthält Zeiger auf Funktion
- Konstruktor initialisiert diesen Zeiger
- Aufruf: „automatisch“ die richtige Funktion
- in größeren Projekten:
Objekt enthält Zeiger auf Tabelle von Funktionen

```
typedef struct  
{  
    void (* print) (union t_object *this);  
    int content;  
} t_integer;
```

Hardwarenahe Programmierung

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

- 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 Unions
 - 6.4 Virtuelle Methoden
 - 6.5 Beispiel: Graphische Benutzeroberfläche (GUI)
 - 6.6 Ausblick: C++
- 7 Datenstrukturen**