

# Vertiefung Software-Entwicklung in C++

Prof. Dr. rer. nat. Peter Gerwinski

5. November 2018

# Vertiefung Software-Entwicklung in C++

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

## 1 Einführung

## 2 Wiederholung: Programmieren in C

...

2.11 Arrays und Strings

2.12 Strukturen

2.13 Dateien und Fehlerbehandlung

2.14 Parameter des Hauptprogramms

2.15 String-Operationen

2.16 Bibliotheken

2.17 Algorithmen

2.18 Bit-Operationen

2.19 Dynamische Speicherverwaltung

...

## 3 Einführung in C++

...



Änderungen  
vorbehalten

## 2.17 Algorithmen

### 2.17.1 Differentialgleichungen

$$\varphi'(t) = \omega(t)$$

$$\omega'(t) = -\frac{g}{l} \cdot \sin \varphi(t)$$

- Von Hand (analytisch): Lösung raten (Ansatz), Parameter berechnen
- Mit Computer (numerisch): Eulersches Polygonzugverfahren

```
phi += dt * omega;  
omega += - dt * g / l * sin (phi);
```

Praktikumsaufgabe im 5. Semester Bachelor: Basketball

## 2.17 Algorithmen

### 2.17.2 Rekursion

Vollständige Induktion:

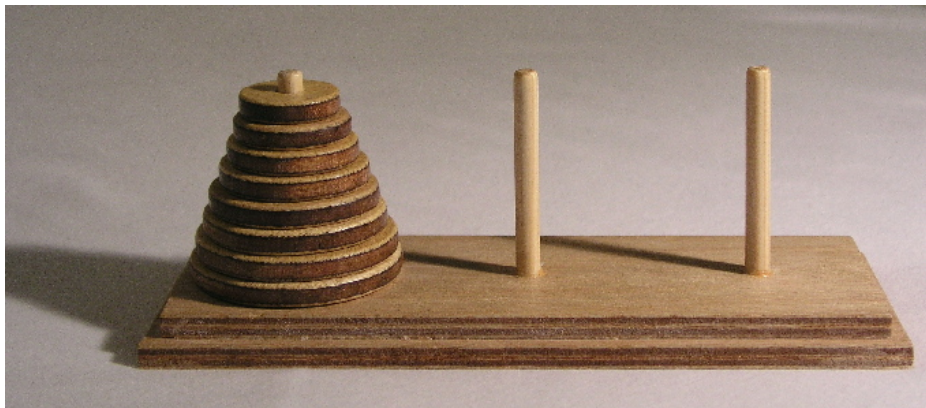
Aussage gilt für  $n = 1$   
Schluß von  $n - 1$  auf  $n$  } Aussage gilt für alle  $n \in \mathbb{N}$

## 2.17 Algorithmen

### 2.17.2 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



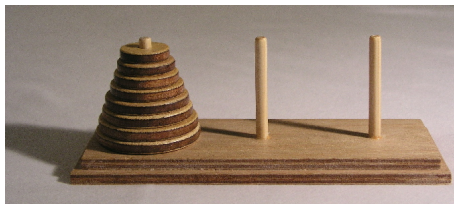
## 2.17 Algorithmen

### 2.17.2 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.



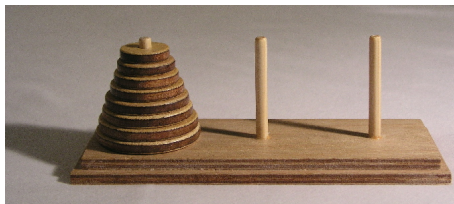
## 2.17 Algorithmen

### 2.17.2 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



## 2.17 Algorithmen

### 2.17.2 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

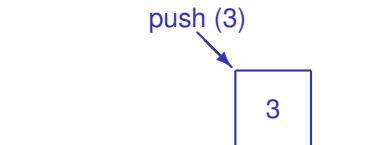
```
void verschiebe (int n, int start, int ziel)
{
    if (n == 1)
        verschiebe_1_scheibe (start, ziel);
    else
    {
        verschiebe (1, start, hilfsplatz);
        verschiebe (n - 1, start, ziel);
        verschiebe (1, hilfsplatz, ziel);
    }
}
```



## 2.17 Algorithmen

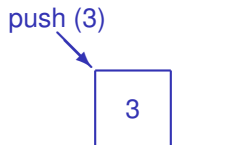
### 2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

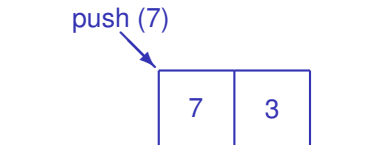


LIFO = Stack = Stapel

## 2.17 Algorithmen

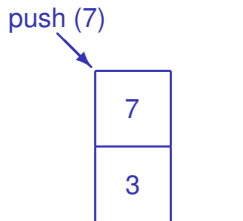
### 2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

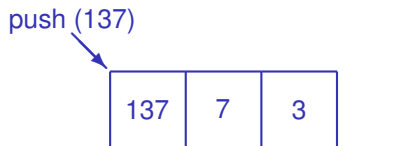


LIFO = Stack = Stapel

## 2.17 Algorithmen

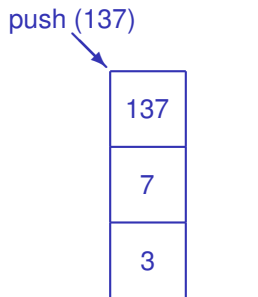
### 2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



LIFO = Stack = Stapel

## 2.17 Algorithmen

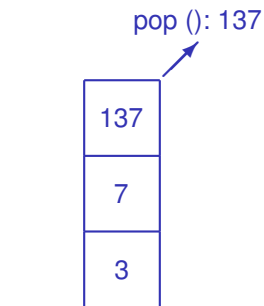
### 2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

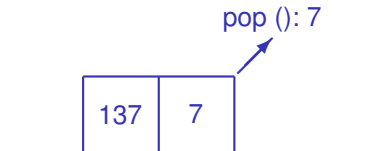


LIFO = Stack = Stapel

## 2.17 Algorithmen

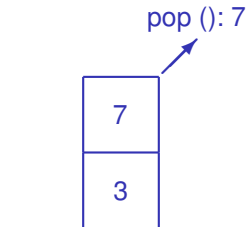
### 2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

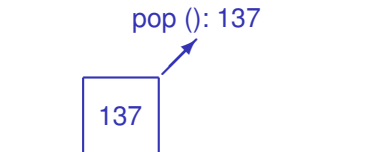


LIFO = Stack = Stapel

## 2.17 Algorithmen

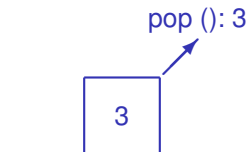
### 2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



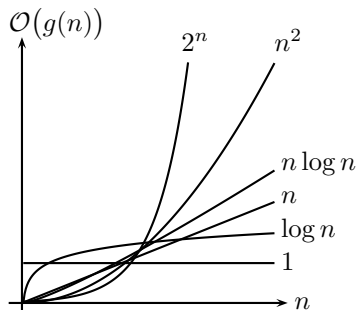
LIFO = Stack = Stapel

## 2.17 Algorithmen

### 2.17.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen



$n$ : Eingabedaten

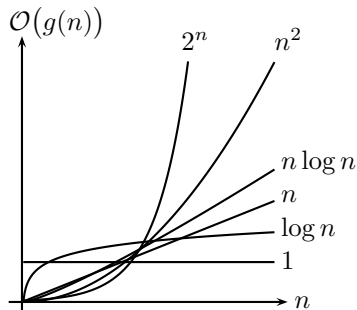
$g(n)$ : Rechenzeit

## 2.17 Algorithmen

### 2.17.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen:  $\mathcal{O}(n)$



$n$ : Eingabedaten

$g(n)$ : Rechenzeit

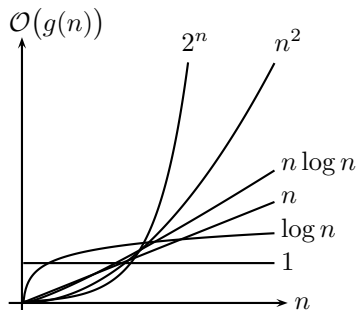


## 2.17 Algorithmen

### 2.17.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen:  $\mathcal{O}(n)$
- Maximum ans Ende tauschen  
→ Selectionsort



$n$ : Eingabedaten

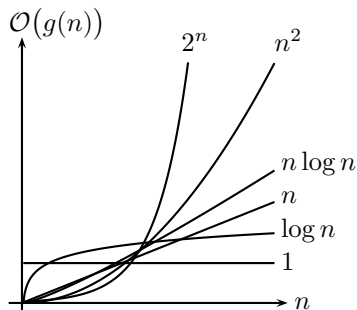
$g(n)$ : Rechenzeit

## 2.17 Algorithmen

### 2.17.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen:  $\mathcal{O}(n)$
- Maximum ans Ende tauschen  
→ Selectionsort:  $\mathcal{O}(n^2)$



$n$ : Eingabedaten

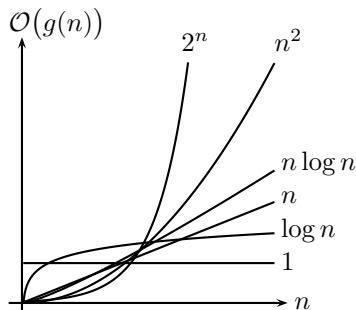
$g(n)$ : Rechenzeit

## 2.17 Algorithmen

### 2.17.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen:  $\mathcal{O}(n)$
- Maximum ans Ende tauschen  
→ Selectionsort:  $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen,  
abbrechen, falls schon sortiert  
→ Bubblesort



$n$ : Eingabedaten

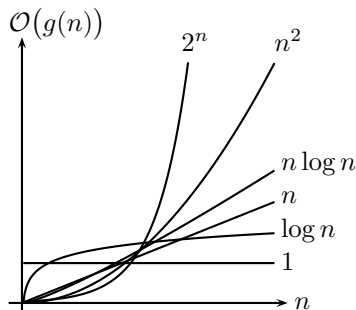
$g(n)$ : Rechenzeit

## 2.17 Algorithmen

### 2.17.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen:  $\mathcal{O}(n)$
- Maximum ans Ende tauschen  
→ Selectionsort:  $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen,  
abbrechen, falls schon sortiert  
→ Bubblesort:  $\mathcal{O}(n)$  bis  $\mathcal{O}(n^2)$



$n$ : Eingabedaten

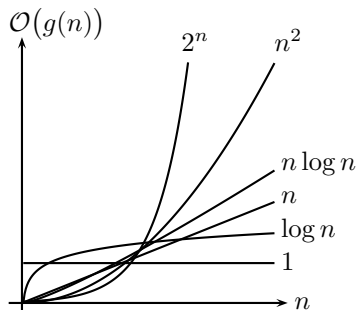
$g(n)$ : Rechenzeit

## 2.17 Algorithmen

### 2.17.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen:  $\mathcal{O}(n)$
- Maximum ans Ende tauschen  
→ Selectionsort:  $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen,  
abbrechen, falls schon sortiert  
→ Bubblesort:  $\mathcal{O}(n)$  bis  $\mathcal{O}(n^2)$
- Rekursiv sortieren  
→ Quicksort



$n$ : Eingabedaten

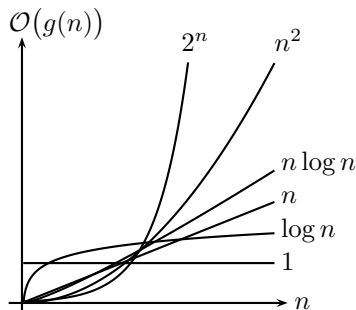
$g(n)$ : Rechenzeit

## 2.17 Algorithmen

### 2.17.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen:  $\mathcal{O}(n)$
- Maximum ans Ende tauschen  
→ Selectionsort:  $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen, 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

## 2.18 Bit-Operationen

### 2.18.1 Zahlensysteme

Basis		Beispiel
2	Binärsystem	1 0000 0011
8	Oktalsystem	0403
10	Dezimalsystem	259
16	Hexadezimalsystem	0x103
256	IP-Adressen (IPv4)	0.0.1.3

## 2.18 Bit-Operationen

### 2.18.1 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



## 2.18 Bit-Operationen

### 2.18.2 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

## 2.19 Dynamische Speicherverwaltung

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

```
...
```

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

## 2.19 Dynamische Speicherverwaltung

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

```
...
```

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

```
...
```

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

Aufgabe: Schreiben Sie eine C-Bibliothek, die ein dynamisches „Array von Bits“ realisiert, z. B. mittels der folgenden Funktionen:

<code>bit_array *create_bit_array (int size);</code>	Bit-Array dynamisch erzeugen
<code>void free_bit_array (bit_array *b);</code>	Bit-Array wieder freigeben
<code>void set_bit (bit_array *b, int i);</code>	Bei Index $i$ auf 1 setzen
<code>void clear_bit (bit_array *b, int i);</code>	Bei Index $i$ auf 0 setzen
<code>int get_bit (bit_array *b, int i);</code>	Bei Index $i$ lesen

Hinweise:

- Der Datentyp `bit_array` ist – sinnvollerweise in der `.h`-Datei – selbst zu definieren, z. B. als ein `struct`, das einen Zeiger auf die eigentlichen Daten, eine Größenangabe und evtl. noch zusätzliche Daten enthält.
- Die Benutzung des Bit-Arrays soll vollständig durch die o. a. Funktionen („Methoden“) erfolgen. Der Benutzer braucht sich insbesondere nicht damit zu beschäftigen, wie das Bit-Array intern aufgebaut ist.
- Sie benötigen ein Array, z. B. von `char`- oder `int`-Variablen, am besten `uint8_t` oder einen größeren Typ (aus `stdint.h`).
- Sie benötigen eine Division (`/`) sowie den Divisionsrest (Modulo: `%`).
- Die Größe des Bit-„Arrays“ wird über den Aufruf der Funktion `create_bit_array()` dynamisch festgelegt.