

# Algorithmen und Datenstrukturen in C/C++

Prof. Dr. rer. nat. Peter Gerwinski

23. März 2023

# 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
- 7 **Datenstrukturen**
  - 7.1 Stack und FIFO
  - 7.2 Verkettete Listen
  - 7.3 Bäume

# Algorithmen

## Differentialgleichungen

- Pendel
- Planetenbahnen

## Rekursion

- Türme von Hanoi

→ *Wie rechnet man das überhaupt aus?*

## Aufwandsabschätzungen

- Selectionsort
- Bubblesort
- Quicksort

→ *Wie rechnet man das möglichst effizient aus?*

- ? möglichst schnell
- ? mit möglichst wenig Speicherplatzverbrauch
- ? unter Berücksichtigung gegebener Randbedingungen

# Datenstrukturen

## Structs und Objekte

- zusammengehörige Daten gemeinsam speichern

## Stack und FIFO – Arrays

- effizientes Anfügen und Entfernen vorne und hinten
- effizienter direkter Zugriff auf Elemente in der Mitte
- ineffizientes Einfügen und Entfernen in der Mitte

## Verkettete Listen – Structs mit Pointern

- effizientes Einfügen und Entfernen in der Mitte
- ineffizienter direkter Zugriff auf Elemente in der Mitte

## Bäume – Structs mit Pointern

- Kompromiß

→ *Wie speichert man das möglichst effizient?*

- ? möglichst schnell
- ? mit möglichst wenig Speicherplatzverbrauch
- ? unter Berücksichtigung gegebener Randbedingungen

# Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Türme von Hanoi:  $\mathcal{O}(2^n)$

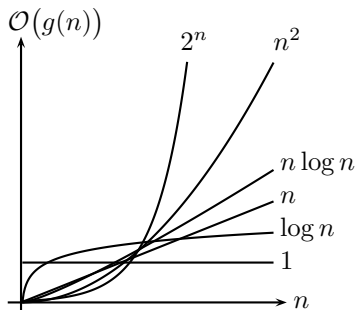
Für jede zusätzliche Scheibe  
verdoppelt sich die Rechenzeit!

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

für 64 Scheiben

Faustregel:

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



$n$ : Eingabedaten

$g(n)$ : Rechenzeit

# Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

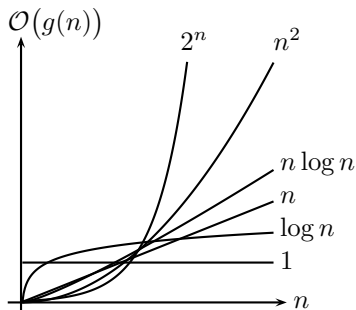
Faustregel:

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

# 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):

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

$$\mathcal{O}(n^2)$$

- Schlüsselerzeugung (Berechnung von  $d$ ):

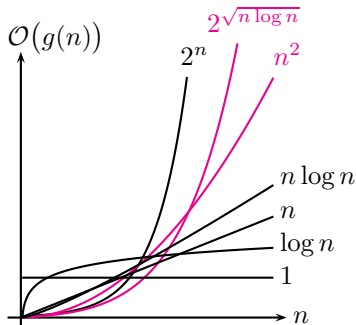
$$\mathcal{O}((\log n)^2)$$

$$\mathcal{O}(n^2)$$

- Verschlüsselung brechen (Primfaktorzerlegung):

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

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



$n$ : Eingabedaten

$g(n)$ : Rechenzeit

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

# Algorithmen und Datenstrukturen in C/C++

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

1 Einführung

2 ...

$\mathcal{O}(n \log n)$  Rasterung array  
struct Wegfindung CORDIC FFT  
**B-Baum** Verschlüsseln pointer  
*verkettete Liste* Datenbanken *Datenkompression*  
digitale Signatur Hash-Tabelle  
kryptographische Hash-Funktion Prüfsumme



# Algorithmen und Datenstrukturen in C/C++

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

- 1 Einführung
- 2 Datenorganisation
- 3 Optimierung
- 4 Hardwarenahe Algorithmen
- 5 Datenkodierung
- 6 Numerik



Änderungen  
vorbehalten

## C: Arrays und Zeiger für Fortgeschrittene

Array:

```
char a[] = "Test";
```

Zeiger:

```
char *p = "Test";
```

- In beiden Fällen wird ein Array von ganzen Zahlen (5 **char**-Variable mit den Werten 84, 101, 115, 116 und 0) im Speicher angelegt.
- Links heißt das Array **a**; rechts ist es „anonym“.
- Rechts wird zusätzlich ein Zeiger **p** im Speicher angelegt, der auf das (anonyme) Array zeigt.
- **&a** ist dasselbe wie **a**, nämlich die Adresse des Arrays.
- **&p** ist die Adresse des Zeigers.
- **p** ist der Wert des Zeigers, momentan also die Adresse des (anonymen) Arrays.

## C: Arrays und Zeiger für Fortgeschrittene

Array:

```
char a[] = "Test";
```

Zeiger:

```
char *p = "Test";
```

- In beiden Fällen wird ein Array von ganzen Zahlen (5 **char**-Variable mit den Werten 84, 101, 115, 116 und 0) im Speicher angelegt.
- Links heißt das Array **a**; rechts ist es „anonym“.
- Rechts wird zusätzlich ein Zeiger **p** im Speicher angelegt, der auf das (anonyme) Array zeigt.
- **&a** ist **fast** dasselbe wie **a**, nämlich die Adresse des Arrays **bzw. das Array selbst, das zuweisungskompatibel zu einem Zeiger auf Elemente des Arrays ist. & bewirkt hier eine (nicht explizite!) Typumwandlung.**
- **&p** ist die Adresse des Zeigers.
- **p** ist der Wert des Zeigers, momentan also die Adresse des (anonymen) Arrays.

## C: Arrays und Zeiger für Fortgeschrittene

Array:

```
char *a[] = { "Dies", "ist", "ein", "Test" };
```

Zeiger:

```
char **p = a;
```

- Array von Zeigern auf **char**-Variable
- Zeiger auf das Array = Zeiger auf Zeiger auf **char**-Variable
- Schleife durch äußeres Array mit **p++** möglich

## C: Arrays und Zeiger für Fortgeschrittene

Array:

```
char a[][5] = { "Dies", "ist", "ein", "Test" };
```

Zeiger:

```
char *p = a[0];
```

- zweidimensionales Array von **char**-Variablen
- Zeiger auf Array-Komponente
  - = Zeiger auf eindimensionales Array
  - = Zeiger auf **char**-Variable
- Schleife durch äußeres Array mit Zeiger-Arithmetik ~~nicht~~ möglich

  
nur mit Trick: p += 5

## C: Arrays und Zeiger für Fortgeschrittene

```
typedef char string5[5];  
string5 a[] = { "Dies", "ist", "ein", "Test" };  
string5 *p = a;
```

- Array von Array von **char**-Variablen  
= zweidimensionales Array von **char**-Variablen
- Zeiger auf zweidimensionales Array
- Schleife durch äußeres Array mit **p++** möglich

→ Fazit:  
Ein Hoch auf **typedef**!

→ Trotzdem: **Vorsicht!**  
Ein **p++** auf einen Zeiger vom Typ **string5 \*p**  
ergibt anscheinend undefiniertes Verhalten!

## C: Arrays und Zeiger für Fortgeschrittene

```
typedef char string5[5];  
string5 *p = { "Dies", "ist", "ein", "Test" };
```

- ~~anonymes Array von Array von **char**-Variablen  
= anonymes zweidimensionales Array von **char**-Variablen~~
- ~~Zeiger auf zweidimensionales Array~~
- ~~Schleife durch äußeres Array mit **p++** möglich~~

Das Konstrukt { "Dies", "ist", "ein", "Test" }  
steht für ein Array von 4 Zeigern auf **char**-Variable.

**string5 \*p** hingegen erwartet einen Zeiger auf ein Array von 5 **char**-Variablen.

Es bekommt die Adresse von "Dies" zugewiesen.

Durch das Erhöhen von **p** (um 5) zeigt es danach *zufällig* auf das "ist".

Bei nochmaligem Erhöhen zeigt es auf das "in" von "ein".

(Auch ohne Optimierung werden die Strings "ist", "ein" und "Test"  
u. U. wegoptimiert.)

## Übungsaufgabe: Verkettete Listen

- (a) Schreiben Sie eine Funktion, die eine verkettete Liste in die umgekehrte Reihenfolge bringt.
- (b) Schreiben Sie eine Funktion, die eine verkettete Liste in eine doppelt verkettete Liste umwandelt.
- (c) Wieviel Rechenzeit (Landau-Symbol) benötigen Ihre Funktionen?
- (d) Wieviel Speicherplatz (Landau-Symbol) benötigen Ihre Funktionen?



# Algorithmen und Datenstrukturen in C/C++

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

- 1 Einführung
- 2 Datenorganisation
- 3 Optimierung
- 4 Hardwarenahe Algorithmen
- 5 Datenkodierung
- 6 Numerik



Änderungen  
vorbehalten

# 1 Einführung in C++

## 1.0 Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen



Hardware und/oder Betriebssystem

- Hardware direkt ansprechen und effizient einsetzen
- ... bis hin zu komplexen Software-Projekten

→ Man kann Computer vollständig beherrschen.

# 1 Einführung in C++

## 1.0 Was ist C?

### Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen
- Hardware direkt ansprechen und effizient einsetzen
- ... bis hin zu komplexen Software-Projekten
- leistungsfähig, aber gefährlich

### „High-Level-Assembler“

*C makes it easy to shoot yourself in the foot.*

- kein „Fallschirm“
- kompakte Schreibweise

Bjarne Stroustrup, ca. 1986

[http://www.stroustrup.com/bs\\_faq.html#really-say-that](http://www.stroustrup.com/bs_faq.html#really-say-that)

### Unix-Hintergrund

- Baukastenprinzip
- konsequente Regeln
- kein „Fallschirm“

# 1 Einführung in C++

## 1.1 Was ist C++?

Etabliertes Profi-Werkzeug

- kompatibel zu C

*C++ is a better C.*

C++ unterstützt

- *objektorientierte Programmierung*
- *Datenabstraktion*
- *generische Programmierung*

Bjarne Stroustrup, Autor von C++  
<http://www.stroustrup.com/C++.html>

*C makes it easy to shoot yourself in the foot;  
C++ makes it harder, but when you do  
it blows your whole leg off.*

Bjarne Stroustrup, Autor von C++, ca. 1986  
[http://www.stroustrup.com/bs\\_faq.html](http://www.stroustrup.com/bs_faq.html)  
[#really-say-that](#)

### Motivation:

Vermeidung unsicherer Techniken,  
insbesondere von Präprozessor-Konstruktionen und Zeigern,  
unter Beibehaltung der Effizienz

## 1.2 Elementare Neuerungen in C++ gegenüber C

## 1.2 Elementare Neuerungen in C++ gegenüber C

- Kommentare mit //

## 1.2 Elementare Neuerungen in C++ gegenüber C

- Kommentare mit //
- Konstante:  
**const int** n = 5;  
**int** prime[n] = { 2, 3, 5, 7, 11 };

## 1.2 Elementare Neuerungen in C++ gegenüber C

- Kommentare mit //
- Konstante:  
`const int n = 5;`  
`int prime[n] = { 2, 3, 5, 7, 11 };`
- Ab C++11: `constexpr`-Funktionen  
C++11: darf nur aus einem einzigen `return`-Statement bestehen  
→ `?:` statt `if`, Rekursion statt Schleife  
C++-14: auch Verzweigungen und Schleifen erlaubt



## 1.2 Elementare Neuerungen in C++ gegenüber C

- Kommentare mit //
- Konstante:  
`const int n = 5;`  
`int prime[n] = { 2, 3, 5, 7, 11 };`
- Ab C++11: `constexpr`-Funktionen  
C++11: darf nur aus einem einzigen `return`-Statement bestehen  
→ `?:` statt `if`, Rekursion statt Schleife  
C++14: auch Verzweigungen und Schleifen erlaubt
- leere Parameterliste: `void` optional  
in C: ohne `void` = Parameterliste wird nicht geprüft

## 1.2 Elementare Neuerungen in C++ gegenüber C

- Kommentare mit //
- Konstante:  
`const int n = 5;`  
`int prime[n] = { 2, 3, 5, 7, 11 };`
- Ab C++11: `constexpr`-Funktionen  
C++11: darf nur aus einem einzigen `return`-Statement bestehen  
→ `?:` statt `if`, Rekursion statt Schleife  
C++-14: auch Verzweigungen und Schleifen erlaubt
- leere Parameterliste: `void` optional  
in C: ohne `void` = Parameterliste wird nicht geprüft
- Operatoren `new` und `delete`  
als Alternative zu den Funktionen `malloc()` und `free()`

## 1.3 Referenz-Typen

```
void calc_answer (int &answer)
{
    answer = 42;
}
```

... als Alternative zu ...

```
void calc_answer (int *answer)
{
    *answer = 42;
}
```

- Zeiger „verborgen“, übersichtlicher und sicherer
- Es gibt keinen **NULL**-Wert.  
→ Für verkettete Listen u. ä.: Tricks erforderlich

## 1.4 Überladbare Operatoren und Funktionen

```
#include <iostream>
```

```
int main ()
```

```
{
```

```
    std::cout << "Hello, world!" << std::endl;
```

```
    return 0;
```

```
}
```

## 1.4 Überladbare Operatoren und Funktionen

```
#include <iostream>
```

```
int main ()  
{  
    std::cout << "Hello, world!" << std::endl;  
    return 0;  
}
```

Bemerkungen:

- Compilieren mit `g++` statt `gcc`:  
C++-Bibliotheken mit einbinden
- Der Operator `<<` hat normalerweise keinen Seiteneffekt, hier schon.

## 1.4 Überladbare Operatoren und Funktionen

```
#include <iostream>
```

```
struct vector  
{  
    double x, y, z;  
};
```

```
vector operator + (vector u, vector v)  
{  
    vector w = { u.x + v.x, u.y + v.y, u.z + v.z };  
    return w;  
}
```

- ++ wird zum Präfix-Operator.
- ++ mit zusätzlichem (ungenutzten) **int**-Parameter wird zum Postfix-Operator.

## 1.4 Überladbare Operatoren und Funktionen

```
void print (const char *s)
```

```
{  
    printf ("%s", s);  
}
```

```
void print (int i)
```

```
{  
    printf ("%d", i);  
}
```

## 1.4 Überladbare Operatoren und Funktionen

```
void print (const char *s)
{
    printf ("%s", s);
}
```

```
void print (int i)
{
    printf ("%d", i);
}
```

Optionale Parameter:

```
void print (const char *s = "\n")
{
    printf ("%s", s);
}
```



## 1.4 Überladbare Operatoren und Funktionen

```
void print (const char *s)
```

```
{  
    printf ("%s", s);  
}
```

Für den Linker:  
veränderte, eindeutige Namen

```
void print (int i)
```

```
{  
    printf ("%d", i);  
}
```

Optionale Parameter:

```
void print (const char *s = "\n")
```

```
{  
    printf ("%s", s);  
}
```

wird vom Compiler erledigt

## 1.4 Überladbare Operatoren und Funktionen

```
void print (const char *s)
```

```
{  
    printf ("%s", s);  
}
```

```
void print (int i)
```

```
{  
    printf ("%d", i);  
}
```

Für den Linker:  
veränderte, eindeutige Namen

Wenn man das nicht will:  
extern "C" { ... }

Optionale Parameter:

```
void print (const char *s = "\n")
```

```
{  
    printf ("%s", s);  
}
```

wird vom Compiler erledigt

## 1.5 Namensräume

```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    cout << "Hello, _world!" << endl;
```

```
    return 0;
```

```
}
```

## 1.5 Namensräume

```
#include <iostream>
```

```
using namespace std;
```

```
int main ()  
{  
    cout << "Hello,_world!" << endl;  
    return 0;  
}
```

```
namespace my_output  
{  
    ...  
}
```

```
using namespace my_output;
```