

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

Wintersemester 2024/25

Wichtiger Hinweis

Diese Vortragsfolien dienen dazu, den Vortrag der/des Lehrenden zu unterstützen. Sie enthalten **nur einen Teil** der Lerninhalte. Wie groß dieser Teil ist, hängt von den konkreten Lerninhalten ab und kann von „praktisch alles“ bis „praktisch gar nichts“ schwanken. Diese Folien alleine sind daher **nicht für ein Selbststudium geeignet!** Hierfür sei auf das Skript verwiesen, in dem allerdings keine tagesaktuellen Änderungen enthalten sind.

Mindestens genauso wichtig wie die Vortragsfolien sind die Beispiel-Programme, die vor Ihren Augen in den Vorlesungen erarbeitet werden. Diese sind im Git-Repository (<https://gitlab.cvh-server.de/pgerwinski/hp.git>) mit allen Zwischenschritten enthalten und befinden sich in den zu den jeweiligen Kalenderdaten gehörenden Verzeichnissen (z. B. für den 10. 10. 2024 unter <https://gitlab.cvh-server.de/pgerwinski/hp/tree/2024ws/20241010/>).

Wenn Sie die Übungsaufgaben bearbeiten, nutzen Sie die Gelegenheit, Ihre Lösungen in den Übungen überprüfen zu lassen. Wer nach Vergleich mit der Musterlösung zu dem Schluß kommt, alles richtig gelöst zu haben, erlebt sonst in der Klausur oft eine unangenehme Überraschung.

In jedem Fall: *Viel Erfolg!*

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

10. Oktober 2024

Hardwarenahe Programmierung

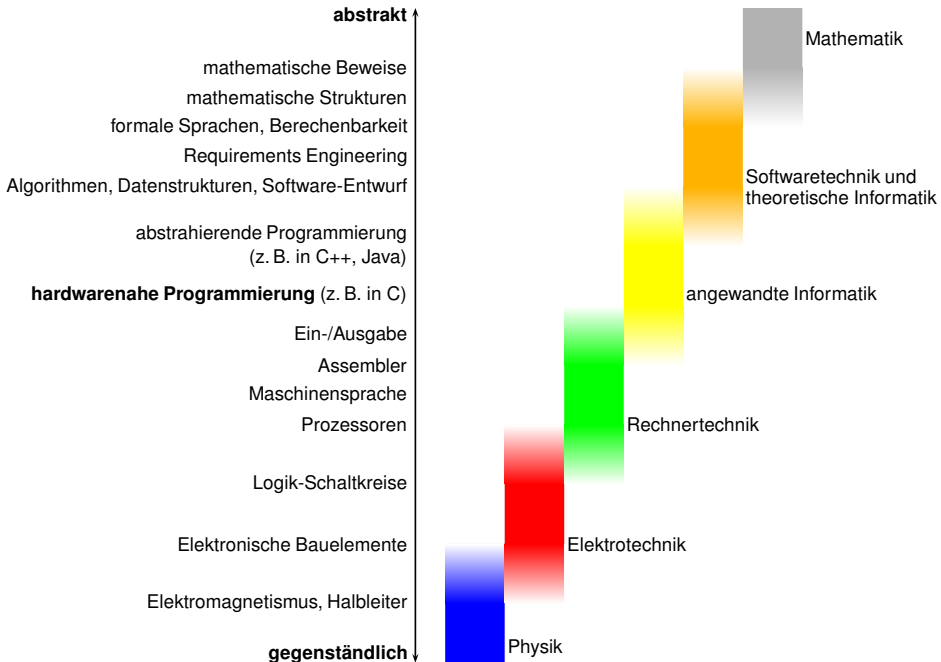
Prof. Dr. rer. nat. Peter Gerwinski

rerum naturalium = der natürlichen Dinge (lat.)

10. Oktober 2024

Vorab: Online-Werkzeuge

- Diese Veranstaltung findet **in Präsenz** statt.
Wir versuchen aber, auch eine Online-Teilnahme zu ermöglichen.
- **Mumble**: Seminarraum 2
Fragen: Mikrofon einschalten oder über den Chat
Umfragen: über den Chat – **auch während der Präsenz-Veranstaltung**
- **VNC**: Kanal 6, Passwort: `testcvh`
Eigenen Bildschirm freigeben: VNC-Software oder Web-Interface *yesVNC*
Eigenes Kamerabild übertragen: Web-Interface *CVH-Camera*
- Allgemeine Informationen: <https://www.cvh-server.de/online-werkzeuge/>
- Notfall-Schnellzugang: <https://www.cvh-server.de/virtuelle-raeume/>
Seminarraum 2, VNC-Passwort: `testcvh`
- **Lehrmaterialien**: <https://gitlab.cvh-server.de/pgerwinski/hp>



Hardwarenahe Programmierung

Man kann Computer vollständig beherrschen.

Hardwarenahe Programmierung

Man kann Computer vollständig beherrschen.

Programmierung in C

- Hardware direkt ansprechen und effizient einsetzen
- ... bis hin zu komplexen Software-Projekten
- Programmierkenntnisse werden nicht vorausgesetzt, aber schnelles Tempo

Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen

Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen

Hardware und/oder Betriebssystem



Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen
- leistungsfähig, aber gefährlich

Hardware und/oder Betriebssystem



Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen
- leistungsfähig, aber gefährlich

„High-Level-Assembler“

Hardware und/oder Betriebssystem



Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen
- leistungsfähig, aber gefährlich

„High-Level-Assembler“

- kein „Fallschirm“

Hardware und/oder Betriebssystem



Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen
- leistungsfähig, aber gefährlich

„High-Level-Assembler“

Hardware und/oder Betriebssystem

- kein „Fallschirm“
- kompakte Schreibweise

Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen
- leistungsfähig, aber gefährlich

„High-Level-Assembler“

Hardware und/oder Betriebssystem

- kein „Fallschirm“
- kompakte Schreibweise

Unix-Hintergrund

Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen
- leistungsfähig, aber gefährlich

„High-Level-Assembler“

Hardware und/oder Betriebssystem

- kein „Fallschirm“
- kompakte Schreibweise

Unix-Hintergrund

- Baukastenprinzip

Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen
- leistungsfähig, aber gefährlich



„High-Level-Assembler“

Hardware und/oder Betriebssystem

- kein „Fallschirm“
- kompakte Schreibweise

Unix-Hintergrund

- Baukastenprinzip
- konsequente Regeln

Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen
- leistungsfähig, aber gefährlich



„High-Level-Assembler“

Hardware und/oder Betriebssystem

- kein „Fallschirm“
- kompakte Schreibweise

Unix-Hintergrund

- Baukastenprinzip
- konsequente Regeln
- kein „Fallschirm“

Zu dieser Lehrveranstaltung



- **Lehrmaterialien:**

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

- **Klausur:**

Zeit: 120 Minuten

Zulässige Hilfsmittel:

- Schreibgerät
- beliebige Unterlagen in Papierform und/oder auf Datenträgern
- elektronische Rechner (Notebook, Taschenrechner o. ä.)
- *kein* Internet-Zugang

- **Übungen**

sind mit der Vorlesung und dem Praktikum integriert.

- **Das Praktikum**

findet jede Woche statt.

Diese Woche: vorbereitende Maßnahmen,
Kennenlernen der verwendeten Werkzeuge.

Im Laufe des Semesters: **4 Praktikumsversuche**

Hardwarenahe Programmierung

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

1 Einführung

- 1.1 Was ist hardwarenahe Programmierung?
- 1.2 Programmierung in C
- 1.3 Zu dieser Lehrveranstaltung

2 Einführung in C

- 2.1 Hello, world!
- 2.2 Programme compilieren und ausführen
- 2.3 Elementare Aus- und Eingabe
- 2.4 Elementares Rechnen
- 2.5 Verzweigungen
- 2.6 Schleifen
- 2.7 Strukturierte Programmierung

...

3 Bibliotheken

...

2 Einführung in C

2.1 Hello, world!

Text ausgeben

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    printf ("Hello,_world!\n");  
    return 0;  
}
```

2 Einführung in C

2.1 Hello, world!

Text ausgeben

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    printf ("Hello, _world!\n");  
    return 0;  
}
```

printf = „print formatted“

2 Einführung in C

2.1 Hello, world!

Text ausgeben

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    printf ("Hello, _world!\n");  
    return 0;  
}
```

printf = „print formatted“

\n: Zeilenschaltung



2.2 Programme compilieren und ausführen

```
$ gcc hello-1.c -o hello-1  
$ ./hello-1  
Hello, world!  
$
```


2.2 Programme compilieren und ausführen

```
$ gcc hello-1.c -o hello-1
```

```
$ ./hello-1
```

```
Hello, world!
```

```
$
```

`-o hello-1`

Name für Ausgabe-Datei („output“)
unter Unix: ohne Endung
unter MS-Windows: Endung `.exe`

2.2 Programme compilieren und ausführen

```
$ gcc hello-1.c -o hello-1
$ ./hello-1
Hello, world!
$
```

Hier: Kommandozeilen-Interface (CLI)

- Der C-Compiler (hier: `gcc`) muß installiert sein und sich im `PATH` befinden.
- Der Quelltext (hier: `hello-1.c`) muß sich im aktuellen Verzeichnis befinden.
- aktuelles Verzeichnis herausfinden: `pwd`
- aktuelles Verzeichnis wechseln: `cd foobar`, `cd ..`
- Inhalt des aktuellen Verzeichnisses ausgeben: `ls`, `ls -l`
- Ausführen des Programms (`hello-1`) im aktuellen Verzeichnis (.):
`./hello-1`

Alternative: Integrierte Entwicklungsumgebung (IDE)
mit graphischer Benutzeroberfläche (GUI)

- Das können Sie bereits.

2.2 Programme compilieren und ausführen

```
$ gcc hello-1.c -o hello-1
$ ./hello-1
Hello, world!
$
```

GNU Compiler Collection (GCC) für verschiedene Plattformen:


- GNU/Linux: [gcc](#)
- Apple Mac OS: [Xcode](#)
- Microsoft Windows: [Cygwin](#)
oder [MinGW](#) mit [MSYS](#)
oder [WSL](#) mit darin installiertem [GNU/Linux](#)
- außerdem: Texteditor
[vi\(m\)](#), [nano](#), [Emacs](#), [Notepad++](#), ...
(Microsoft Notepad ist *nicht* geeignet!)

2.2 Programme compilieren und ausführen

```
$ gcc hello-1.c -o hello-1  
$ ./hello-1  
Hello, world!  
$
```

2.2 Programme compilieren und ausführen

```
$ gcc -Wall -O hello-1.c -o hello-1
$ ./hello-1
Hello, world!
$
```



-Wall	alle Warnungen einschalten
-O	optimieren
-O3	maximal optimieren
-Os	Codegröße optimieren
...	gcc hat <i>sehr viele</i> Optionen.

2.3 Elementare Aus- und Eingabe

Wert ausgeben

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    printf ("Die_Antwort_lautet:_");
```

```
    printf (42);
```

```
    printf ("\n");
```

```
    return 0;
```

```
}
```

2.3 Elementare Aus- und Eingabe

Wert ausgeben

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    printf ("Die_Antwort_lautet:_");
```

```
    printf (42);
```

```
    printf ("\n");
```

```
    return 0;
```

```
}
```

→ Absturz

2.3 Elementare Aus- und Eingabe

Wert ausgeben

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    printf ("Die_Antwort_lautet:_%d\n", 42);
```

```
    return 0;
```

```
}
```

Formatspezifikation „d“: „dezimal“



2.3 Elementare Aus- und Eingabe

Wert ausgeben

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    printf ("Die_Antwort_lautet:_%d\n", 42);
```

```
    return 0;
```

```
}
```

Formatspezifikation „d“: „dezimal“

Weitere Formatspezifikationen:
siehe Dokumentation (z. B. man 3 printf),
Internet-Recherche oder Literatur

2.3 Elementare Aus- und Eingabe

Wert einlesen

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    double a;
```

```
    printf ("Bitte_eine_Zahl_eingeben:_");
```

```
    scanf ("%lf", &a);
```

```
    printf ("Ihre_Antwort_war:_%lf\n", a);
```

```
    return 0;
```

```
}
```

Formatspezifikation „lf“:
„long floating-point“

Das „&“ nicht vergessen!

2.4 Elementares Rechnen

Wert an Variable zuweisen

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    int a;
```

```
    printf ("Bitte_eine_Zahl_eingeben:_");
```

```
    scanf ("%d", &a);
```

```
    a = 2 * a;
```

```
    printf ("Das_Doppelte_ist:_%d\n", a);
```

```
    return 0;
```

```
}
```

2.4 Elementares Rechnen

Variable bei Deklaration initialisieren

```
int a = 42;  
a = 137;
```

Achtung: Initialisierung \neq Zuweisung

Die beiden Gleichheitszeichen haben
subtil unterschiedliche Bedeutungen!

2.5 Verzweigungen

if-Verzweigung

```
if (b != 0)
    printf ("%d\n", a / b);
```

2.5 Verzweigungen

if-Verzweigung

```
if (b != 0)
    printf ("%d\n", a / b);
```

Wahrheitswerte in C: numerisch

0 steht für *falsch* (*false*),
 $\neq 0$ steht für *wahr* (*true*).

```
if (b)
    printf ("%d\n", a / b);
```

2.6 Schleifen

while-Schleife

```
a = 1;  
while (a <= 10)  
{  
    printf ("%d\n", a);  
    a = a + 1;  
}
```

2.6 Schleifen

while-Schleife

```
a = 1;
while (a <= 10)
{
    printf ("%d\n", a);
    a = a + 1;
}
```

for-Schleife

```
for (a = 1; a <= 10; a = a + 1)
    printf ("%d\n", a);
```


2.6 Schleifen

while-Schleife

```
a = 1;
while (a <= 10)
{
    printf ("%d\n", a);
    a = a + 1;
}
```

for-Schleife

```
for (a = 1; a <= 10; a = a + 1)
    printf ("%d\n", a);
```

do-while-Schleife

```
a = 1;
do
{
    printf ("%d\n", a);
    a = a + 1;
}
while (a <= 10);
```

```
int i;
```

```
i = 0;
```

```
while (i < 10)
```

```
{
```

```
    printf ("%d\n", i);
```

```
    i++;
```

```
}
```

```
for (i = 0; i < 10; i++)
```

```
    printf ("%d\n", i);
```

```
i = 0;
```

```
while (i < 10)
```

```
    printf ("%d\n", i++);
```

```
for (i = 0; i < 10; printf ("%d\n", i++));
```

```
i = 0;  
while (1)  
{  
    if (i >= 10)  
        break;  
    printf ("%d\n", i++);  
}
```

```
int i;
```

```
i = 0;  
while (i < 10)  
{  
    printf ("%d\n", i);  
    i++;  
}
```

```
for (i = 0; i < 10; i++)  
    printf ("%d\n", i);
```

```
i = 0;  
while (i < 10)  
    printf ("%d\n", i++);
```

```
for (i = 0; i < 10; printf ("%d\n", i++));
```

```
i = 0;  
while (1)  
{  
    if (i >= 10)  
        break;  
    printf ("%d\n", i++);  
}
```

```
i = 0;  
loop:  
if (i >= 10)  
    goto endloop;  
printf ("%d\n", i++);  
goto loop;  
endloop:
```

```
int i;
```

```
i = 0;  
while (i < 10)  
{  
    printf ("%d\n", i);  
    i++;  
}
```

```
for (i = 0; i < 10; i++)  
    printf ("%d\n", i);
```

```
i = 0;  
while (i < 10)  
    printf ("%d\n", i++);
```

```
for (i = 0; i < 10; printf ("%d\n", i++));
```

2.7 Strukturierte Programmierung

```
i = 0;  
while (1)  
{  
    if (i >= 10)  
        break;  
    printf ("%d\n", i++);  
}
```

```
i = 0;  
loop:  
if (i >= 10)  
    goto endloop;  
printf ("%d\n", i++);  
goto loop;  
endloop:
```

```
int i;  
  
i = 0;  
while (i < 10)  
{  
    printf ("%d\n", i);  
    i++;  
}
```

```
for (i = 0; i < 10; i++)  
    printf ("%d\n", i);
```

```
i = 0;  
while (i < 10)  
    printf ("%d\n", i++);
```

```
for (i = 0; i < 10; printf ("%d\n", i++));
```

2.7 Strukturierte Programmierung

```
i = 0;  
while (1)      fragwürdig  
{  
    if (i >= 10)  
        break;  
    printf ("%d\n", i++);  
}
```

```
i = 0;  
loop:  
if (i >= 10)    sehr fragwürdig  
    goto endloop; (siehe z. B.: http://xkcd.com/292/)  
printf ("%d\n", i++);  
goto loop;  
endloop:
```

```
int i;  
  
i = 0;  
while (i < 10)  
{  
    printf ("%d\n", i);  
    i++;  
}
```

gut

```
for (i = 0; i < 10; i++)  
    printf ("%d\n", i);
```

```
i = 0;  
while (i < 10)  
    printf ("%d\n", i++);
```

nur, wenn
Sie wissen,
was Sie tun

```
for (i = 0; i < 10; printf ("%d\n", i++));
```

Hardwarenahe Programmierung

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

1 Einführung

- 1.1 Was ist hardwarenahe Programmierung?
- 1.2 Programmierung in C
- 1.3 Zu dieser Lehrveranstaltung

2 Einführung in C

- 2.1 Hello, world!
- 2.2 Programme compilieren und ausführen
- 2.3 Elementare Aus- und Eingabe
- 2.4 Elementares Rechnen
- 2.5 Verzweigungen
- 2.6 Schleifen
- 2.7 Strukturierte Programmierung
- 2.8 Seiteneffekte
- 2.9 Funktionen
- 2.10 Zeiger

...

3 Bibliotheken

...

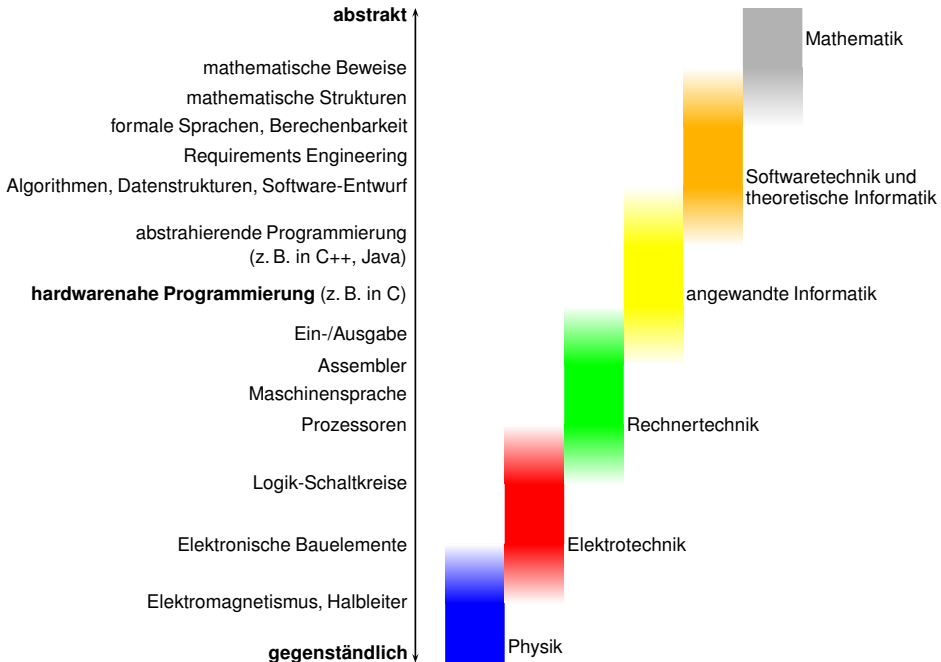
Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

17. Oktober 2024

Vorab: Online-Werkzeuge

- Diese Veranstaltung findet **in Präsenz** statt.
Wir versuchen aber, auch eine Online-Teilnahme zu ermöglichen.
- **Mumble**: Seminarraum 2
Fragen: Mikrofon einschalten oder über den Chat
Umfragen: über den Chat – **auch während der Präsenz-Veranstaltung**
- **VNC**: Kanal 6, Passwort: `testcvh`
Eigenen Bildschirm freigeben: VNC-Software oder Web-Interface *yesVNC*
Eigenes Kamerabild übertragen: Web-Interface *CVH-Camera*
- Allgemeine Informationen: <https://www.cvh-server.de/online-werkzeuge/>
- Notfall-Schnellzugang: <https://www.cvh-server.de/virtuelle-raeume/>
Seminarraum 2, VNC-Passwort: `testcvh`
- **Lehrmaterialien**: <https://gitlab.cvh-server.de/pgerwinski/hp>



Hardwarenahe Programmierung

Man kann Computer vollständig beherrschen.

Programmierung in C

- Hardware direkt ansprechen und effizient einsetzen
- ... bis hin zu komplexen Software-Projekten
- Programmierkenntnisse werden nicht vorausgesetzt, aber schnelles Tempo

Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen
- leistungsfähig, aber gefährlich



„High-Level-Assembler“

Hardware und/oder Betriebssystem

- kein „Fallschirm“
- kompakte Schreibweise

Unix-Hintergrund

- Baukastenprinzip
- konsequente Regeln
- kein „Fallschirm“

Zu dieser Lehrveranstaltung



- **Lehrmaterialien:**

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

- **Klausur:**

Zeit: 120 Minuten

Zulässige Hilfsmittel:

- Schreibgerät
- beliebige Unterlagen in Papierform und/oder auf Datenträgern
- elektronische Rechner (Notebook, Taschenrechner o. ä.)
- *kein* Internet-Zugang

- **Übungen**

sind mit der Vorlesung und dem Praktikum integriert.

- **Das Praktikum**

findet jede Woche statt.

Diese Woche: vorbereitende Maßnahmen,
Kennenlernen der verwendeten Werkzeuge.

Im Laufe des Semesters: **4 Praktikumsversuche**

Hardwarenahe Programmierung

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

1 Einführung

- 1.1 Was ist hardwarenahe Programmierung?
- 1.2 Programmierung in C
- 1.3 Zu dieser Lehrveranstaltung

2 Einführung in C

- 2.1 Hello, world!
- 2.2 Programme compilieren und ausführen
- 2.3 Elementare Aus- und Eingabe
- 2.4 Elementares Rechnen
- 2.5 Verzweigungen
- 2.6 Schleifen
- 2.7 Strukturierte Programmierung
- ...

3 Bibliotheken

...

2 Einführung in C

2.1 Hello, world!

Text ausgeben

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    printf ("Hello, _world!\n");  
    return 0;  
}
```

printf = „print formatted“

\n: Zeilenschaltung



2.2 Programme compilieren und ausführen

```
$ gcc hello-1.c -o hello-1
```

```
$ ./hello-1
```

```
Hello, world!
```

```
$
```


2.2 Programme compilieren und ausführen

```
$ gcc hello-1.c -o hello-1
```

```
$ ./hello-1
```

```
Hello, world!
```

```
$
```

`-o hello-1`

Name für Ausgabe-Datei („output“)
unter Unix: ohne Endung
unter MS-Windows: Endung `.exe`

2.2 Programme compilieren und ausführen

```
$ gcc hello-1.c -o hello-1
$ ./hello-1
Hello, world!
$
```

Hier: Kommandozeilen-Interface (CLI)

- Der C-Compiler (hier: `gcc`) muß installiert sein und sich im `PATH` befinden.
- Der Quelltext (hier: `hello-1.c`) muß sich im aktuellen Verzeichnis befinden.
- aktuelles Verzeichnis herausfinden: `pwd`
- aktuelles Verzeichnis wechseln: `cd foobar`, `cd ..`
- Inhalt des aktuellen Verzeichnisses ausgeben: `ls`, `ls -l`
- Ausführen des Programms (`hello-1`) im aktuellen Verzeichnis (`.`):
`./hello-1`

Alternative: Integrierte Entwicklungsumgebung (IDE)
mit graphischer Benutzeroberfläche (GUI)

- Das können Sie bereits.

2.2 Programme compilieren und ausführen

```
$ gcc hello-1.c -o hello-1
$ ./hello-1
Hello, world!
$
```

GNU Compiler Collection (GCC) für verschiedene Plattformen:


- GNU/Linux: [gcc](#)
- Apple Mac OS: [Xcode](#)
- Microsoft Windows: [Cygwin](#)
oder [MinGW](#) mit [MSYS](#)
oder [WSL](#) mit darin installiertem [GNU/Linux](#)
- außerdem: Texteditor
[vi\(m\)](#), [nano](#), [Emacs](#), [Notepad++](#), ...
(Microsoft Notepad ist *nicht* geeignet!)

2.2 Programme compilieren und ausführen

```
$ gcc hello-1.c -o hello-1  
$ ./hello-1  
Hello, world!  
$
```

2.2 Programme compilieren und ausführen

```
$ gcc -Wall -O hello-1.c -o hello-1
$ ./hello-1
Hello, world!
$
```



-Wall	alle Warnungen einschalten
-O	optimieren
-O3	maximal optimieren
-Os	Codegröße optimieren
...	gcc hat <i>sehr viele</i> Optionen.

2.3 Elementare Aus- und Eingabe

Wert ausgeben

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    printf ("Die_Antwort_lautet:_");
```

```
    printf (42);
```

```
    printf ("\n");
```

```
    return 0;
```

```
}
```

→ Absturz

2.3 Elementare Aus- und Eingabe

Wert ausgeben

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    printf ("Die_Antwort_lautet:_%d\n", 42);
```

```
    return 0;
```

```
}
```

Formatspezifikation „d“: „dezimal“

Weitere Formatspezifikationen:
siehe Dokumentation (z. B. man 3 printf),
Internet-Recherche oder Literatur

2.3 Elementare Aus- und Eingabe

Wert einlesen

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    double a;
```

```
    printf ("Bitte_eine_Zahl_eingeben:_");
```

```
    scanf ("%lf", &a);
```

```
    printf ("Ihre_Antwort_war:_%lf\n", a);
```

```
    return 0;
```

```
}
```

Formatspezifikation „lf“:
„long floating-point“

Das „&“ nicht vergessen!

2.4 Elementares Rechnen

Wert an Variable zuweisen

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    int a;
```

```
    printf ("Bitte_eine_Zahl_eingeben:_");
```

```
    scanf ("%d", &a);
```

```
    a = 2 * a;
```

```
    printf ("Das_Doppelte_ist:_%d\n", a);
```

```
    return 0;
```

```
}
```

2.4 Elementares Rechnen

Variable bei Deklaration initialisieren

```
int a = 42;  
a = 137;
```

Achtung: Initialisierung \neq Zuweisung

Die beiden Gleichheitszeichen haben
subtil unterschiedliche Bedeutungen!

2.5 Verzweigungen

if-Verzweigung

```
if (b != 0)
    printf ("%d\n", a / b);
```

Wahrheitswerte in C: numerisch

0 steht für *falsch* (*false*),
≠ 0 steht für *wahr* (*true*).

```
if (b)
    printf ("%d\n", a / b);
```

2.6 Schleifen

while-Schleife

```
a = 1;
while (a <= 10)
{
    printf ("%d\n", a);
    a = a + 1;
}
```

for-Schleife

```
for (a = 1; a <= 10; a = a + 1)
    printf ("%d\n", a);
```

do-while-Schleife

```
a = 1;
do
{
    printf ("%d\n", a);
    a = a + 1;
}
while (a <= 10);
```

Hardwarenahe Programmierung

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

1 Einführung

- 1.1 Was ist hardwarenahe Programmierung?
- 1.2 Programmierung in C
- 1.3 Zu dieser Lehrveranstaltung

2 Einführung in C

- 2.1 Hello, world!
- 2.2 Programme compilieren und ausführen
- 2.3 Elementare Aus- und Eingabe
- 2.4 Elementares Rechnen
- 2.5 Verzweigungen
- 2.6 Schleifen
- 2.7 Strukturierte Programmierung
- 2.8 Seiteneffekte
- 2.9 Funktionen
- 2.10 Zeiger

...

3 Bibliotheken

...

```
int i;
```

```
i = 0;
```

```
while (i < 10)
```

```
{
```

```
    printf ("%d\n", i);
```

```
    i++;
```

```
}
```

```
for (i = 0; i < 10; i++)
```

```
    printf ("%d\n", i);
```

```
i = 0;
```

```
while (i < 10)
```

```
    printf ("%d\n", i++);
```

```
for (i = 0; i < 10; printf ("%d\n", i++));
```

```
i = 0;  
while (1)  
{  
    if (i >= 10)  
        break;  
    printf ("%d\n", i++);  
}
```

```
int i;
```

```
i = 0;  
while (i < 10)  
{  
    printf ("%d\n", i);  
    i++;  
}
```

```
for (i = 0; i < 10; i++)  
    printf ("%d\n", i);
```

```
i = 0;  
while (i < 10)  
    printf ("%d\n", i++);
```

```
for (i = 0; i < 10; printf ("%d\n", i++));
```

```
i = 0;  
while (1)  
{  
    if (i >= 10)  
        break;  
    printf ("%d\n", i++);  
}
```

```
i = 0;  
loop:  
if (i >= 10)  
    goto endloop;  
printf ("%d\n", i++);  
goto loop;  
endloop:
```

```
int i;
```

```
i = 0;  
while (i < 10)  
{  
    printf ("%d\n", i);  
    i++;  
}
```

```
for (i = 0; i < 10; i++)  
    printf ("%d\n", i);
```

```
i = 0;  
while (i < 10)  
    printf ("%d\n", i++);
```

```
for (i = 0; i < 10; printf ("%d\n", i++));
```


2.7 Strukturierte Programmierung

```
i = 0;
while (1)
{
    if (i >= 10)
        break;
    printf ("%d\n", i++);
}
```

```
i = 0;
loop:
if (i >= 10)
    goto endloop;
printf ("%d\n", i++);
goto loop;
endloop:
```

```
int i;

i = 0;
while (i < 10)
{
    printf ("%d\n", i);
    i++;
}
```

```
for (i = 0; i < 10; i++)
    printf ("%d\n", i);
```

```
i = 0;
while (i < 10)
    printf ("%d\n", i++);
```

```
for (i = 0; i < 10; printf ("%d\n", i++));
```

2.7 Strukturierte Programmierung

```
i = 0;  
while (1) fragwürdig  
{  
    if (i >= 10)  
        break;  
    printf ("%d\n", i++);  
}
```

```
i = 0;  
loop:  
if (i >= 10) sehr fragwürdig  
    goto endloop; (siehe z. B.: http://xkcd.com/292/)  
printf ("%d\n", i++);  
goto loop;  
endloop:
```

```
int i;  
  
i = 0;  
while (i < 10)  
{  
    printf ("%d\n", i);  
    i++;  
}
```

gut

```
for (i = 0; i < 10; i++)  
    printf ("%d\n", i);
```

```
i = 0;  
while (i < 10)  
    printf ("%d\n", i++);
```

nur, wenn
Sie wissen,
was Sie tun

```
for (i = 0; i < 10; printf ("%d\n", i++));
```

2.8 Seiteneffekte

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    printf ("Hello, _world!\n");  
    "Hello, _world!\n";  
    return 0;  
}
```

2.8 Seiteneffekte

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    printf ("Hello, _world!\n");
```

```
    "Hello, _world!\n"; ← Ausdruck als Anweisung: Wert wird ignoriert
```

```
    return 0;
```

```
}
```

2.8 Seiteneffekte

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    printf ("Hello, _world!\n");
```

```
    "Hello, _world!\n";
```

```
    return 0;
```

```
}
```

← Ausdruck als Anweisung: Wert wird ignoriert

2.8 Seiteneffekte

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    int a = printf ("Hello, _world!\n");
```

```
    printf ("%d\n", a);
```

```
    return 0;
```

```
}
```

2.8 Seiteneffekte

```
#include <stdio.h>
```

```
int main (void)
{
    int a = printf ("Hello, world!\n");
    printf ("%d\n", a);
    return 0;
}
```

```
$ gcc -Wall -O side-effects-1.c -o side-effects-1
$ ./side-effects-1
Hello, world!
14
$
```

2.8 Seiteneffekte

```
#include <stdio.h>
```

```
int main (void)
{
    int a = printf ("Hello, \_world!\n");
    printf ("%d\n", a);
    return 0;
}
```

- `printf()` ist eine Funktion.

2.8 Seiteneffekte

```
#include <stdio.h>
```

```
int main (void)
{
    int a = printf ("Hello, _world!\n");
    printf ("%d\n", a);
    return 0;
}
```

- `printf()` ist eine Funktion.
- „Haupteffekt“: Wert zurückliefern
(hier: Anzahl der ausgegebenen Zeichen)

2.8 Seiteneffekte

```
#include <stdio.h>
```

```
int main (void)
{
    int a = printf ("Hello, _world!\n");
    printf ("%d\n", a);
    return 0;
}
```

- `printf()` ist eine Funktion.
- „Haupteffekt“: Wert zurückliefern
(hier: Anzahl der ausgegebenen Zeichen)
- *Seiteneffekt*: Ausgabe

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `-foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo--`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `--foo`

Binäre Operatoren:

- Rechnen: `+ - * / %`
- Vergleich: `== != < > <= >=`
- Zuweisung: `= += -= *= /= %=`
- Ignorieren: `, a, b`: berechne `a`,
ignore es, nimm stattdessen `b`

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `-foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo--`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `--foo`

Binäre Operatoren:

- Rechnen: `+ - * / %`
- Vergleich: `== != < > <= >=`
- Zuweisung: `= += -= *= /= %=`
- Ignorieren: `, a, b`: berechne `a`,
ignoreiere es, nimm stattdessen `b`

rot = mit Seiteneffekt

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `-foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo--`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `--foo`

```
int i;
```

```
i = 0;
```

```
while (i < 10)
```

```
{
```

```
    printf ("%d\n", i);
```

```
    i++;
```

```
}
```

Binäre Operatoren:

- Rechnen: `+` `-` `*` `/` `%`
- Vergleich: `==` `!=` `<` `>` `<=` `>=`
- Zuweisung: `=` `+=` `-=` `*=` `/=` `%=`
- Ignorieren: `,` `a`, `b`: berechne `a`,
ignoriere es, nimm stattdessen `b`

rot = mit Seiteneffekt

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `-foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo--`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `--foo`

Binäre Operatoren:

- Rechnen: `+ - * / %`
- Vergleich: `== != < > <= >=`
- Zuweisung: `= += -= *= /= %=`
- Ignorieren: `, a, b`: berechne `a`,
ignore es, nimm stattdessen `b`

rot = mit Seiteneffekt

```
int i;
```

```
i = 0;
```

```
while (i < 10)
```

```
{
```

```
    printf ("%d\n", i);
```

```
    i++;
```

```
}
```

```
for (i = 0; i < 10; i++)
```

```
    printf ("%d\n", i);
```

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `-foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo--`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `--foo`

Binäre Operatoren:

- Rechnen: `+ - * / %`
- Vergleich: `== != < > <= >=`
- Zuweisung: `= += -= *= /= %=`
- Ignorieren: `, a, b`: berechne `a`,
ignore es, nimm stattdessen `b`

rot = mit Seiteneffekt

```
int i;
```

```
i = 0;
```

```
while (i < 10)
```

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

```
for (i = 0; i < 10; i++)
```

```
    printf ("%d\n", i);
```

```
i = 0;
```

```
while (i < 10)
```

```
    printf ("%d\n", i++);
```

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `-foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo--`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `--foo`

Binäre Operatoren:

- Rechnen: `+ - * / %`
- Vergleich: `== != < > <= >=`
- Zuweisung: `= += -= *= /= %=`
- Ignorieren: `, a, b`: berechne `a`,
ignore es, nimm stattdessen `b`

rot = mit Seiteneffekt

```
int i;
```

```
i = 0;
```

```
while (i < 10)
```

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

```
for (i = 0; i < 10; i++)
```

```
    printf ("%d\n", i);
```

```
i = 0;
```

```
while (i < 10)
```

```
    printf ("%d\n", i++);
```

```
for (i = 0; i < 10; printf ("%d\n", i++));
```


Hardwarenahe Programmierung

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

1 Einführung

2 Einführung in C

2.1 Hello, world!

2.2 Programme compilieren und ausführen

2.3 Elementare Aus- und Eingabe

2.4 Elementares Rechnen

2.5 Verzweigungen

2.6 Schleifen

2.7 Strukturierte Programmierung

2.8 Seiteneffekte

2.9 Funktionen

2.10 Zeiger

2.11 Arrays und Strings

...

3 Bibliotheken

...

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

31. Oktober 2024

Hardwarenahe Programmierung

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

1 Einführung

2 Einführung in C

2.1 Hello, world!

2.2 Programme compilieren und ausführen

2.3 Elementare Aus- und Eingabe

2.4 Elementares Rechnen

2.5 Verzweigungen

2.6 Schleifen

2.7 Strukturierte Programmierung

2.8 Seiteneffekte

2.9 Funktionen

2.10 Zeiger

2.11 Arrays und Strings

2.12 Strukturen

...

3 Bibliotheken

...

2.7 Strukturierte Programmierung

```
i = 0;  
while (1)      fragwürdig  
{  
    if (i >= 10)  
        break;  
    printf ("%d\n", i++);  
}
```

```
i = 0;  
loop:  
if (i >= 10)   sehr fragwürdig  
    goto endloop; (siehe z. B.: http://xkcd.com/292/)  
printf ("%d\n", i++);  
goto loop;  
endloop:
```

```
int i;  
  
i = 0;  
while (i < 10)  
{  
    printf ("%d\n", i);  
    i++;  
}
```

gut

```
for (i = 0; i < 10; i++)  
    printf ("%d\n", i);
```

```
i = 0;  
while (i < 10)  
    printf ("%d\n", i++);
```

nur, wenn
Sie wissen,
was Sie tun

```
for (i = 0; i < 10; printf ("%d\n", i++));
```

2.8 Seiteneffekte

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    printf ("Hello, _world!\n");  
    "Hello, _world!\n";  
    return 0;  
}
```

2.8 Seiteneffekte

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    printf ("Hello, _world!\n");
```

```
    "Hello, _world!\n"; ← Ausdruck als Anweisung: Wert wird ignoriert
```

```
    return 0;
```

```
}
```

2.8 Seiteneffekte

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    printf ("Hello, _world!\n");
```

```
    "Hello, _world!\n";
```

```
    return 0;
```

```
}
```

Ausdruck als Anweisung: Wert wird ignoriert

2.8 Seiteneffekte

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    int a = printf ("Hello, \uworld!\n");
```

```
    printf ("%d\n", a);
```

```
    return 0;
```

```
}
```


2.8 Seiteneffekte

```
#include <stdio.h>
```

```
int main (void)
{
    int a = printf ("Hello, _world!\n");
    printf ("%d\n", a);
    return 0;
}
```

```
$ gcc -Wall -O side-effects-1.c -o side-effects-1
$ ./side-effects-1
Hello, world!
14
$
```

2.8 Seiteneffekte

```
#include <stdio.h>
```

```
int main (void)
{
    int a = printf ("Hello, \_world!\n");
    printf ("%d\n", a);
    return 0;
}
```

- `printf()` ist eine Funktion.

2.8 Seiteneffekte

```
#include <stdio.h>
```

```
int main (void)
{
    int a = printf ("Hello, _world!\n");
    printf ("%d\n", a);
    return 0;
}
```

- `printf()` ist eine Funktion.
- „Haupteffekt“: Wert zurückliefern
(hier: Anzahl der ausgegebenen Zeichen)

2.8 Seiteneffekte

```
#include <stdio.h>
```

```
int main (void)
{
    int a = printf ("Hello, _world!\n");
    printf ("%d\n", a);
    return 0;
}
```

- `printf()` ist eine Funktion.
- „Haupteffekt“: Wert zurückliefern
(hier: Anzahl der ausgegebenen Zeichen)
- *Seiteneffekt*: Ausgabe

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `-foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo--`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `--foo`

Binäre Operatoren:

- Rechnen: `+ - * / %`
- Vergleich: `== != < > <= >=`
- Zuweisung: `= += -= *= /= %=`
- Ignorieren: `, a, b`: berechne `a`,
ignore es, nimm stattdessen `b`

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `-foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo--`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `--foo`

Binäre Operatoren:

- Rechnen: `+ - * / %`
- Vergleich: `== != < > <= >=`
- Zuweisung: `= += -= *= /= %=`
- Ignorieren: `, a, b`: berechne `a`,
ignoriere es, nimm stattdessen `b`

rot = mit Seiteneffekt

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `-foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo--`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `--foo`

```
int i;
```

```
i = 0;
```

```
while (i < 10)
```

```
{
```

```
    printf ("%d\n", i);
```

```
    i++;
```

```
}
```

Binäre Operatoren:

- Rechnen: `+` `-` `*` `/` `%`
- Vergleich: `==` `!=` `<` `>` `<=` `>=`
- Zuweisung: `=` `+=` `-=` `*=` `/=` `%=`
- Ignorieren: `,` `a`, `b`: berechne `a`,
 ignoriere es, nimm stattdessen `b`

rot = mit Seiteneffekt

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `-foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo--`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `--foo`

Binäre Operatoren:

- Rechnen: `+ - * / %`
- Vergleich: `== != < > <= >=`
- Zuweisung: `= += -= *= /= %=`
- Ignorieren: `, a, b`: berechne `a`,
ignore es, nimm stattdessen `b`

```
int i;
```

```
i = 0;
```

```
while (i < 10)
```

```
{
```

```
    printf ("%d\n", i);
```

```
    i++;
```

```
}
```

```
for (i = 0; i < 10; i++)
```

```
    printf ("%d\n", i);
```

rot = mit Seiteneffekt

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `-foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo--`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `--foo`

Binäre Operatoren:

- Rechnen: `+ - * / %`
- Vergleich: `== != < > <= >=`
- Zuweisung: `= += -= *= /= %=`
- Ignorieren: `, a, b`: berechne `a`,
ignore es, nimm stattdessen `b`

rot = mit Seiteneffekt

```
int i;
```

```
i = 0;
```

```
while (i < 10)
```

```
{
```

```
    printf ("%d\n", i);
```

```
    i++;
```

```
}
```

```
for (i = 0; i < 10; i++)
```

```
    printf ("%d\n", i);
```

```
i = 0;
```

```
while (i < 10)
```

```
    printf ("%d\n", i++);
```

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `-foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo--`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `--foo`

Binäre Operatoren:

- Rechnen: `+ - * / %`
- Vergleich: `== != < > <= >=`
- Zuweisung: `= += -= *= /= %=`
- Ignorieren: `, a, b`: berechne `a`,
ignore es, nimm stattdessen `b`

rot = mit Seiteneffekt

```
int i;
```

```
i = 0;
```

```
while (i < 10)
```

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

```
for (i = 0; i < 10; i++)
```

```
    printf ("%d\n", i);
```

```
i = 0;
```

```
while (i < 10)
```

```
    printf ("%d\n", i++);
```

```
for (i = 0; i < 10; printf ("%d\n", i++));
```

2.9 Funktionen

```
#include <stdio.h>
```

```
int answer (void)
```

```
{  
    return 42;  
}
```

```
void foo (void)
```

```
{  
    printf ("%d\n", answer ());  
}
```

```
int main (void)
```

```
{  
    foo ();  
    return 0;  
}
```

2.9 Funktionen

```
#include <stdio.h>
```

```
int answer (void)
```

```
{  
    return 42;  
}
```

```
void foo (void)
```

```
{  
    printf ("%d\n", answer ());  
}
```

```
int main (void)
```

```
{  
    foo ();  
    return 0;  
}
```

- Funktionsdeklaration:
Typ Name (Parameterliste)
{
 Anweisungen
}

2.9 Funktionen

```
#include <stdio.h>
```

```
void add_verbose (int a, int b)
{
    printf ("%d_+_%d=_%d\n", a, b, a + b);
}
```

```
int main (void)
{
    add_verbose (3, 7);
    return 0;
}
```

- Funktionsdeklaration:
Typ Name (Parameterliste)
{
 Anweisungen
}

2.9 Funktionen

```
#include <stdio.h>
```

```
void add_verbose (int a, int b)
{
    printf ("%d_+_d=_d\n", a, b, a + b);
}
```

```
int main (void)
{
    add_verbose (3, 7);
    return 0;
}
```

- Funktionsdeklaration:
Typ Name (Parameterliste)
{
 Anweisungen
}
- Der Datentyp **void** steht für „nichts“ und kann ignoriert werden.

2.9 Funktionen

```
#include <stdio.h>
```

```
void add_verbose (int a, int b)
{
    printf ("%d_+_%d=_%d\n", a, b, a + b);
}
```

```
int main (void)
{
    add_verbose (3, 7);
    return 0;
}
```

- Funktionsdeklaration:
Typ Name (Parameterliste)
{
 Anweisungen
}
- Der Datentyp **void**
steht für „nichts“
und muß ignoriert werden.

2.9 Funktionen

```
#include <stdio.h>
```

```
int a, b = 3;
```

```
void foo (void)
```

```
{  
    b++;  
    static int a = 5;  
    int b = 7;  
    printf ("foo():_"  
           "a=_%d,_b=_%d\n",  
           a, b);  
    a++;  
}
```

```
int main (void)
```

```
{  
    printf ("main():_"  
           "a=_%d,_b=_%d\n",  
           a, b);  
    foo ();  
    printf ("main():_"  
           "a=_%d,_b=_%d\n",  
           a, b);  
    a = b = 12;  
    printf ("main():_"  
           "a=_%d,_b=_%d\n",  
           a, b);  
    foo ();  
    printf ("main():_"  
           "a=_%d,_b=_%d\n",  
           a, b);  
    return 0;  
}
```


2.10 Zeiger

```
#include <stdio.h>
```

```
void calc_answer (int *a)
```

```
{
```

```
    *a = 42;
```

```
}
```

```
int main (void)
```

```
{
```

```
    int answer;
```

```
    calc_answer (&answer);
```

```
    printf ("The_answer_is_%d.\n", answer);
```

```
    return 0;
```

```
}
```

2.10 Zeiger

```
#include <stdio.h>
```

```
void calc_answer (int *a)
```

```
{  
    *a = 42;  
}
```

- `*a` ist eine `int`.

```
int main (void)
```

```
{  
    int answer;  
    calc_answer (&answer);  
    printf ("The_answer_is_%d.\n", answer);  
    return 0;  
}
```

2.10 Zeiger

```
#include <stdio.h>
```

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

- `*a` ist eine **int**.
- unärer Operator `*`:
Pointer-Derferenzierung

```
int main (void)
{
    int answer;
    calc_answer (&answer);
    printf ("The_answer_is_%d.\n", answer);
    return 0;
}
```

2.10 Zeiger

```
#include <stdio.h>
```

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

- `*a` ist eine **int**.
- unärer Operator `*`:
Pointer-Derferenzierung

→ `a` ist ein Zeiger (Pointer) auf eine **int**.

```
int main (void)
{
    int answer;
    calc_answer (&answer);
    printf ("The_answer_is_%d.\n", answer);
    return 0;
}
```

2.10 Zeiger

```
#include <stdio.h>
```

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

- `*a` ist eine **int**.
- unärer Operator `*`:
Pointer-Derferenzierung

→ `a` ist ein Zeiger (Pointer) auf eine **int**.

```
int main (void)
{
    int answer;
    calc_answer (&answer);
    printf ("The_answer_is_%d.\n", answer);
    return 0;
}
```

- unärer Operator `&`: Adresse

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
{
    int prime[5] = { 2, 3, 5, 7, 11 };
    int *p = prime;
    for (int i = 0; i < 5; i++)
        printf ("%d\n", *(p + i));
    return 0;
}
```


2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

int main (**void**)

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int *p = prime;  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", *(p + i));  
    return 0;  
}
```

- **prime** ist eine Ansammlung von fünf ganzen Zahlen.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int *p = prime;  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", *(p + i));  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

int main (**void**)

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int *p = prime;  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", *(p + i));  
    return 0;  
}
```

- **prime** ist ein Array von fünf ganzen Zahlen.
- **prime** ist ein Zeiger auf eine **int**.



2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

int main (**void**)

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int *p = prime;  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", *(p + i));  
    return 0;  
}
```

- **prime** ist ein Array von fünf ganzen Zahlen.
- **prime** ist ein Zeiger auf eine **int**.
- **p + i** ist ein Zeiger auf den **i**-ten Nachbarn von ***p**.



2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int *p = prime;  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", *(p + i));  
    return 0;  
}
```



- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

int main (**void**)

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int *p = prime;  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", p[i]);  
    return 0;  
}
```



- **prime** ist ein Array von fünf ganzen Zahlen.
- **prime** ist ein Zeiger auf eine **int**.
- **p + i** ist ein Zeiger auf den **i**-ten Nachbarn von ***p**.
- ***(p + i)** ist der **i**-te Nachbar von ***p**.
- Andere Schreibweise:
p[i] statt ***(p + i)**

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

int main (**void**)

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", prime[i]);  
    return 0;  
}
```

- **prime** ist ein Array von fünf ganzen Zahlen.
- **prime** ist ein Zeiger auf eine **int**.
- **p + i** ist ein Zeiger auf den **i**-ten Nachbarn von ***p**.
- ***(p + i)** ist der **i**-te Nachbar von ***p**.
- Andere Schreibweise:
p[i] statt ***(p + i)**

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

```
int main (void)
{
    int prime[5] = { 2, 3, 5, 7, 11 };
    for (int *p = prime;
         p < prime + 5; p++)
        printf ("%d\n", *p);
    return 0;
}
```

- **prime** ist ein Array von fünf ganzen Zahlen.
- **prime** ist ein Zeiger auf eine **int**.
- **p + i** ist ein Zeiger auf den **i**-ten Nachbarn von ***p**.
- ***(p + i)** ist der **i**-te Nachbar von ***p**.
- Andere Schreibweise:
p[i] statt ***(p + i)**
- Zeiger-Arithmetik:
p++ rückt den Zeiger **p** um eine **int** weiter.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

int main (**void**)

```
{  
    int prime[6] = { 2, 3, 5, 7, 11, 0 };  
    for (int *p = prime; *p; p++)  
        printf ("%d\n", *p);  
    return 0;  
}
```

- **prime** ist ein Array von fünf ganzen Zahlen.
- **prime** ist ein Zeiger auf eine **int**.
- **p + i** ist ein Zeiger auf den **i**-ten Nachbarn von ***p**.
- ***(p + i)** ist der **i**-te Nachbar von ***p**.
- Andere Schreibweise:
p[i] statt ***(p + i)**
- Zeiger-Arithmetik:
p++ rückt den Zeiger **p** um eine **int** weiter.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

int main (**void**)

```
{  
    int prime[] = { 2, 3, 5, 7, 11, 0 };  
    for (int *p = prime; *p; p++)  
        printf ("%d\n", *p);  
    return 0;  
}
```

- **prime** ist ein Array von fünf ganzen Zahlen.
- **prime** ist ein Zeiger auf eine **int**.
- **p + i** ist ein Zeiger auf den **i**-ten Nachbarn von ***p**.
- ***(p + i)** ist der **i**-te Nachbar von ***p**.
- Andere Schreibweise:
p[i] statt ***(p + i)**
- Zeiger-Arithmetik:
p++ rückt den Zeiger **p** um eine **int** weiter.
- Array ohne Längenangabe:
Compiler zählt selbst

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

```
int main (void)
{
    int prime[] = { 2, 3, 5, 7, 11, 0 };
    for (int *p = prime; *p; p++)
        printf ("%d\n", *p);
    return 0;
}
```

**Die Länge des Arrays
ist *nicht* veränderlich!**

- **prime** ist ein Array von fünf ganzen Zahlen.
- **prime** ist ein Zeiger auf eine **int**.
- **p + i** ist ein Zeiger auf den **i**-ten Nachbarn von ***p**.
- ***(p + i)** ist der **i**-te Nachbar von ***p**.
- Andere Schreibweise:
p[i] statt ***(p + i)**
- Zeiger-Arithmetik:
p++ rückt den Zeiger **p** um eine **int** weiter.
- Array ohne explizite Längenangabe:
Compiler zählt selbst

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, _world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, _world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**.

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, _world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars**.

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, _world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.
- Der letzte **char** muß 0 sein. Er kennzeichnet das Ende des Strings.

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
{
    char hello[] = "Hello, world!\n";
    for (char *p = hello; *p; p++)
        printf ("%c", *p);
    return 0;
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.
- Der letzte **char** muß 0 sein. Er kennzeichnet das Ende des Strings.
- Die Formatspezifikation entscheidet über die Ausgabe:
 - %d** dezimal
 - %c** Zeichen
 - %x** hexadezimal

2.11 Arrays und Strings und Zeichen

„Alles ist Zahl.“ – Schule der Pythagoreer, 6. Jh. v. Chr.

"Hello"		{ 72, 101, 108, 108, 111, 0 }
'H'	ist nur eine andere	72
	Schreibweise für	
'a' + 4		'e'

- Welchen Zahlenwert hat '*' im Zeichensatz?

```
printf ("%d\n", '*');
```

(normalerweise: ASCII)

- Ist **char** **ch** ein Großbuchstabe?

```
if (ch >= 'A' && ch <= 'Z')
```

...

- Groß- in Kleinbuchstaben umwandeln

```
ch += 'a' - 'A';
```

Hardwarenahe Programmierung

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

1 Einführung

2 Einführung in C

2.1 Hello, world!

2.2 Programme compilieren und ausführen

2.3 Elementare Aus- und Eingabe

2.4 Elementares Rechnen

2.5 Verzweigungen

2.6 Schleifen

2.7 Strukturierte Programmierung

2.8 Seiteneffekte

2.9 Funktionen

2.10 Zeiger

2.11 Arrays und Strings

2.12 Strukturen

...

3 Bibliotheken

...

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

7. November 2024

Hardwarenahe Programmierung

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

1 Einführung

2 Einführung in C

...

2.5 Verzweigungen

2.6 Schleifen

2.7 Strukturierte Programmierung

2.8 Seiteneffekte

2.9 Funktionen

2.10 Zeiger

2.11 Arrays und Strings

2.12 Strukturen

2.13 Dateien und Fehlerbehandlung

2.14 Parameter des Hauptprogramms

2.15 String-Operationen

3 Bibliotheken

...

2.10 Zeiger

```
#include <stdio.h>
```

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

- `*a` ist eine **int**.
- unärer Operator `*`:
Pointer-Derferenzierung

→ `a` ist ein Zeiger (Pointer) auf eine **int**.

```
int main (void)
{
    int answer;
    calc_answer (&answer);
    printf ("The_answer_is_%d.\n", answer);
    return 0;
}
```

- unärer Operator `&`: Adresse

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
{
    int prime[5] = { 2, 3, 5, 7, 11 };
    int *p = prime;
    for (int i = 0; i < 5; i++)
        printf ("%d\n", *(p + i));
    return 0;
}
```

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    int prime[5] = { 2, 3, 5, 7, 11 };
```

```
    int *p = prime;
```

```
    for (int i = 0; i < 5; i++)
```

```
        printf ("%d\n", *(p + i));
```

```
    return 0;
```

```
}
```

- `prime` ist eine Ansammlung von fünf ganzen Zahlen.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    int prime[5] = { 2, 3, 5, 7, 11 };
```

```
    int *p = prime;
```

```
    for (int i = 0; i < 5; i++)
```

```
        printf ("%d\n", *(p + i));
```

```
    return 0;
```

```
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    int prime[5] = { 2, 3, 5, 7, 11 };
```

```
    int *p = prime;
```

```
    for (int i = 0; i < 5; i++)
```

```
        printf ("%d\n", *(p + i));
```

```
    return 0;
```

```
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.



- `prime` ist ein Zeiger auf eine `int`.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int *p = prime;  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", *(p + i));  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int *p = prime;  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", *(p + i));  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.


2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int *p = prime;  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", p[i]);  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`. 
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", prime[i]);  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    for (int *p = prime;  
        p < prime + 5; p++)  
        printf ("%d\n", *p);  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`
- Zeiger-Arithmetik:
`p++` rückt den Zeiger `p` um eine `int` weiter.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    int prime[6] = { 2, 3, 5, 7, 11, 0 };  
    for (int *p = prime; *p; p++)  
        printf ("%d\n", *p);  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`
- Zeiger-Arithmetik:
`p++` rückt den Zeiger `p` um eine `int` weiter.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    int prime[] = { 2, 3, 5, 7, 11, 0 };  
    for (int *p = prime; *p; p++)  
        printf ("%d\n", *p);  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`
- Zeiger-Arithmetik:
`p++` rückt den Zeiger `p` um eine `int` weiter.
- Array ohne Längenangabe:
Compiler zählt selbst

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    int prime[] = { 2, 3, 5, 7, 11, 0 };  
    for (int *p = prime; *p; p++)  
        printf ("%d\n", *p);  
    return 0;  
}
```

**Die Länge des Arrays
ist *nicht* veränderlich!**

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`
- Zeiger-Arithmetik:
`p++` rückt den Zeiger `p` um eine `int` weiter.
- Array ohne explizite Längenangabe:
Compiler zählt selbst

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, _world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, _world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**.

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, _world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars**.

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.
- Der letzte **char** muß 0 sein. Er kennzeichnet das Ende des Strings.

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, _world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%c", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.
- Der letzte **char** muß 0 sein. Er kennzeichnet das Ende des Strings.
- Die Formatspezifikation entscheidet über die Ausgabe:
 - %d** dezimal
 - %c** Zeichen
 - %x** hexadezimal

2.11 Arrays und Strings und Zeichen

„Alles ist Zahl.“ – Schule der Pythagoreer, 6. Jh. v. Chr.

"Hello"		{ 72, 101, 108, 108, 111, 0 }
'H'	ist nur eine andere	72
	Schreibweise für	
'a' + 4		'e'

- Welchen Zahlenwert hat '*' im Zeichensatz?

```
printf ("%d\n", '*');
```

(normalerweise: ASCII)

- Ist **char** **ch** ein Großbuchstabe?

```
if (ch >= 'A' && ch <= 'Z')
```

...

- Groß- in Kleinbuchstaben umwandeln

```
ch += 'a' - 'A';
```

2.12 Strukturen

```
#include <stdio.h>
```

```
typedef struct
```

```
{
```

```
    char day, month;
```

```
    int year;
```

```
}
```

```
date;
```

```
int main (void)
```

```
{
```

```
    date today = { 7, 11, 2024 };
```

```
    printf ("%d.%d.%d\n", today.day, today.month, today.year);
```

```
    return 0;
```

```
}
```

2.12 Strukturen

```
#include <stdio.h>
```

```
typedef struct
```

```
{
```

```
    char day, month;
```

```
    int year;
```

```
}
```

```
date;
```

```
void set_date (date *d)
```

```
{
```

```
    (*d).day = 7;
```

```
    (*d).month = 11;
```

```
    (*d).year = 2024;
```

```
}
```

```
int main (void)
```

```
{
```

```
    date today;
```

```
    set_date (&today);
```

```
    printf ("%d.%d.%d\n", today.day,  
            today.month, today.year);
```

```
    return 0;
```

```
}
```

2.12 Strukturen

```
#include <stdio.h>
```

```
typedef struct
```

```
{
```

```
    char day, month;
```

```
    int year;
```

```
}
```

```
date;
```

```
void set_date (date *d)
```

```
{
```

```
    d->day = 7;
```

```
    d->month = 11;
```

```
    d->year = 2024;
```

```
}
```

foo->bar ist Abkürzung für (*foo).bar

```
int main (void)
```

```
{
```

```
    date today;
```

```
    set_date (&today);
```

```
    printf ("%d.%d.%d\n", today.day,  
            today.month, today.year);
```

```
    return 0;
```

```
}
```


2.12 Strukturen

```
#include <stdio.h>
```

```
typedef struct
```

```
{  
    char day, month;  
    int year;  
}  
date;
```

```
void set_date (date *d)
```

```
{  
    d->day = 7;  
    d->month = 11;  
    d->year = 2024;  
}
```

`foo->bar` ist Abkürzung für `(*foo).bar`

Eine Funktion, die mit einem **struct** arbeitet, kann man eine *Methode* des **struct** nennen.

```
int main (void)
```

```
{  
    date today;  
    set_date (&today);  
    printf ("%d.%d.%d\n", today.day,  
            today.month, today.year);  
    return 0;  
}
```

2.13 Dateien und Fehlerbehandlung

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    FILE *f = fopen ("fhello.txt", "w");
```

```
    fprintf (f, "Hello, world!\n");
```

```
    fclose (f);
```

```
    return 0;
```

```
}
```

2.13 Dateien und Fehlerbehandlung

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    FILE *f = fopen ("fhello.txt", "w");
```

```
    if (f)
```

```
    {
```

```
        fprintf (f, "Hello,_world!\n");
```

```
        fclose (f);
```

```
    }
```

```
    return 0;
```

```
}
```

- Wenn die Datei nicht geöffnet werden kann, gibt `fopen()` den Wert `NULL` zurück.

2.13 Dateien und Fehlerbehandlung

```
#include <stdio.h>
```

```
#include <errno.h>
```

```
int main (void)
```

```
{
```

```
    FILE *f = fopen ("fhello.txt", "w");
```

```
    if (f)
```

```
    {
```

```
        fprintf (f, "Hello,_world!\n");
```

```
        fclose (f);
```

```
    }
```

```
    else
```

```
        fprintf (stderr, "error_#%d\n", errno);
```

```
    return 0;
```

```
}
```

- Wenn die Datei nicht geöffnet werden kann, gibt `fopen()` den Wert `NULL` zurück.
- Die globale Variable `int errno` enthält dann die Nummer des Fehlers. Benötigt: `#include <errno.h>`

2.13 Dateien und Fehlerbehandlung

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
```

```
int main (void)
{
    FILE *f = fopen ("fhello.txt", "w");
    if (f)
    {
        fprintf (f, "Hello, _world!\n");
        fclose (f);
    }
    else
    {
        char *msg = strerror (errno);
        fprintf (stderr, "%s\n", msg);
    }
    return 0;
}
```

- Wenn die Datei nicht geöffnet werden kann, gibt `fopen()` den Wert `NULL` zurück.
- Die globale Variable `int errno` enthält dann die Nummer des Fehlers. Benötigt: `#include <errno.h>`
- Die Funktion `strerror()` wandelt `errno` in einen Fehlermeldungstext um. Benötigt: `#include <string.h>`

2.13 Dateien und Fehlerbehandlung

```
#include <stdio.h>
```

```
#include <errno.h>
```

```
#include <error.h>
```

```
int main (void)
```

```
{
```

```
    FILE *f = fopen ("fhello.txt", "w");
```

```
    if (!f)
```

```
        error (errno, errno, "cannot_open_file");
```

```
    fprintf (f, "Hello,_world!\n");
```

```
    fclose (f);
```

```
    return 0;
```

```
}
```

- Wenn die Datei nicht geöffnet werden kann, gibt `fopen()` den Wert `NULL` zurück.
- Die globale Variable `int errno` enthält dann die Nummer des Fehlers. Benötigt: `#include <errno.h>`
- Die Funktion `strerror()` wandelt `errno` in einen Fehlermeldungstext um. Benötigt: `#include <string.h>`
- Die Funktion `error()` gibt eine Fehlermeldung aus und beendet das Programm. Benötigt: `#include <error.h>`

2.13 Dateien und Fehlerbehandlung

```
#include <stdio.h>
```

```
#include <errno.h>
```

```
#include <error.h>
```

```
int main (void)
```

```
{
```

```
    FILE *f = fopen ("fhello.txt", "w");
```

```
    if (!f)
```

```
        error (errno, errno, "cannot_open_file");
```

```
    fprintf (f, "Hello,_world!\n");
```

```
    fclose (f);
```

```
    return 0;
```

```
}
```

- Wenn die Datei nicht geöffnet werden kann, gibt `fopen()` den Wert `NULL` zurück.
- Die globale Variable `int errno` enthält dann die Nummer des Fehlers. Benötigt: `#include <errno.h>`
- Die Funktion `strerror()` wandelt `errno` in einen Fehlermeldungstext um. Benötigt: `#include <string.h>`
- Die Funktion `error()` gibt eine Fehlermeldung aus und beendet das Programm. Benötigt: `#include <error.h>`
- **Niemals Fehler einfach ignorieren!**

2.14 Parameter des Hauptprogramms

```
#include <stdio.h>
```

```
int main (int argc, char **argv)
{
    printf ("argc=%d\n", argc);
    for (int i = 0; i < argc; i++)
        printf ("argv[%d]=\n", i, argv[i]);
    return 0;
}
```


2.14 Parameter des Hauptprogramms

```
#include <stdio.h>
```

```
int main (int argc, char **argv)
{
    printf ("argc=%d\n", argc);
    for (int i = 0; *argv; i++, argv++)
        printf ("argv[%d]=\n", i, *argv);
    return 0;
}
```

2.15 String-Operationen

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello,_world!\n";
```

```
    printf ("%s\n", hello);
```

```
    printf ("%zd\n", strlen (hello));
```

```
    printf ("%s\n", hello + 7);
```

```
    printf ("%zd\n", strlen (hello + 7));
```

```
    hello[5] = 0;
```

```
    printf ("%s\n", hello);
```

```
    printf ("%zd\n", strlen (hello));
```

```
    return 0;
```

```
}
```

2.15 String-Operationen

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main (void)
```

```
{
```

```
    char *anton = "Anton";
```

```
    char *zacharias = "Zacharias";
```

```
    printf ("%d\n", strcmp (anton, zacharias));
```

```
    printf ("%d\n", strcmp (zacharias, anton));
```

```
    printf ("%d\n", strcmp (anton, anton));
```

```
    char buffer[100] = "Huber_";
```

```
    strcat (buffer, anton);
```

```
    printf ("%s\n", buffer);
```

```
    return 0;
```

```
}
```

2.15 String-Operationen

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main (void)
```

```
{
```

```
    char buffer[100] = "";
```

```
    sprintf (buffer, "Die_Antwort_lautet:%d", 42);
```

```
    printf ("%s\n", buffer);
```

```
    char *answer = strstr (buffer, "Antwort");
```

```
    printf ("%s\n", answer);
```

```
    printf ("found_at:%zd\n", answer - buffer);
```

```
    return 0;
```

```
}
```

2.15 String-Operationen

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main (void)
```

```
{
```

```
    char buffer[100] = "";
```

```
    snprintf (buffer, 100, "Die_Antwort_lautet:_%d", 42);
```

```
    printf ("%s\n", buffer);
```

```
    char *answer = strstr (buffer, "Antwort");
```

```
    printf ("%s\n", answer);
```

```
    printf ("found_at:_%zd\n", answer - buffer);
```

```
    return 0;
```

```
}
```

2 Einführung in C

Sprachelemente weitgehend komplett

Es fehlen:

- Ergänzungen (z. B. ternärer Operator, **union**, **unsigned**, **volatile**)
- Bibliotheksfunktionen (z. B. **malloc()**)

—> werden eingeführt, wenn wir sie brauchen

- Konzepte (z. B. rekursive Datenstrukturen, Klassen selbst bauen)

—> werden eingeführt, wenn wir sie brauchen, oder:

—> Literatur

(z. B. Wikibooks: C-Programmierung,
Dokumentation zu Compiler und Bibliotheken)

- Praxiserfahrung

—> Übung und Praktikum: nur Einstieg

—> selbständig arbeiten

Hardwarenahe Programmierung

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

1 Einführung

2 Einführung in C

...

2.5 Verzweigungen

2.6 Schleifen

2.7 Strukturierte Programmierung

2.8 Seiteneffekte

2.9 Funktionen

2.10 Zeiger

2.11 Arrays und Strings

2.12 Strukturen

2.13 Dateien und Fehlerbehandlung

2.14 Parameter des Hauptprogramms

2.15 String-Operationen

3 Bibliotheken

...

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

14. November 2024

Hardwarenahe Programmierung

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

1 Einführung

2 Einführung in C

...

2.10 Zeiger

2.11 Arrays und Strings

2.12 Strukturen

2.13 Dateien und Fehlerbehandlung

2.14 Parameter des Hauptprogramms

2.15 String-Operationen

3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliotheken verwenden

3.4 Callbacks

3.5 Projekt organisieren: make

4 Hardwarenahe Programmierung

...

2.10 Zeiger

```
#include <stdio.h>
```

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

- `*a` ist eine **int**.
- unärer Operator `*`:
Pointer-Derferenzierung

→ `a` ist ein Zeiger (Pointer) auf eine **int**.

```
int main (void)
{
    int answer;
    calc_answer (&answer);
    printf ("The_answer_is_%d.\n", answer);
    return 0;
}
```

- unärer Operator `&`: Adresse

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    int prime[5] = { 2, 3, 5, 7, 11 };
```

```
    int *p = prime;
```

```
    for (int i = 0; i < 5; i++)
```

```
        printf ("%d\n", *(p + i));
```

```
    return 0;
```

```
}
```

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    int prime[5] = { 2, 3, 5, 7, 11 };
```

```
    int *p = prime;
```

```
    for (int i = 0; i < 5; i++)
```

```
        printf ("%d\n", *(p + i));
```

```
    return 0;
```

```
}
```

- `prime` ist eine Ansammlung von fünf ganzen Zahlen.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    int prime[5] = { 2, 3, 5, 7, 11 };
```

```
    int *p = prime;
```

```
    for (int i = 0; i < 5; i++)
```

```
        printf ("%d\n", *(p + i));
```

```
    return 0;
```

```
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    int prime[5] = { 2, 3, 5, 7, 11 };
```

```
    int *p = prime;
```

```
    for (int i = 0; i < 5; i++)
```

```
        printf ("%d\n", *(p + i));
```

```
    return 0;
```

```
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.



- `prime` ist ein Zeiger auf eine `int`.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int *p = prime;  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", *(p + i));  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.



2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int *p = prime;  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", *(p + i));  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.


2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int *p = prime;  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", p[i]);  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`. 
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", prime[i]);  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    for (int *p = prime;  
         p < prime + 5; p++)  
        printf ("%d\n", *p);  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`
- Zeiger-Arithmetik:
`p++` rückt den Zeiger `p` um eine `int` weiter.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    int prime[6] = { 2, 3, 5, 7, 11, 0 };
```

```
    for (int *p = prime; *p; p++)
```

```
        printf ("%d\n", *p);
```

```
    return 0;
```

```
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`
- Zeiger-Arithmetik:
`p++` rückt den Zeiger `p` um eine `int` weiter.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    int prime[] = { 2, 3, 5, 7, 11, 0 };  
    for (int *p = prime; *p; p++)  
        printf ("%d\n", *p);  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`
- Zeiger-Arithmetik:
`p++` rückt den Zeiger `p` um eine `int` weiter.
- Array ohne Längenangabe:
Compiler zählt selbst

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    int prime[] = { 2, 3, 5, 7, 11, 0 };  
    for (int *p = prime; *p; p++)  
        printf ("%d\n", *p);  
    return 0;  
}
```

**Die Länge des Arrays
ist *nicht* veränderlich!**

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`
- Zeiger-Arithmetik:
`p++` rückt den Zeiger `p` um eine `int` weiter.
- Array ohne explizite Längenangabe:
Compiler zählt selbst

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, _world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, _world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**.

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars**.

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.
- Der letzte **char** muß 0 sein. Er kennzeichnet das Ende des Strings.

2.11 Arrays und Strings

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, _world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%c", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.
- Der letzte **char** muß 0 sein. Er kennzeichnet das Ende des Strings.
- Die Formatspezifikation entscheidet über die Ausgabe:
 - %d** dezimal
 - %c** Zeichen
 - %x** hexadezimal

2.11 Arrays und Strings und Zeichen

„Alles ist Zahl.“ – Schule der Pythagoreer, 6. Jh. v. Chr.

"Hello"		{ 72, 101, 108, 108, 111, 0 }
'H'	ist nur eine andere	72
	Schreibweise für	
'a' + 4		'e'

- Welchen Zahlenwert hat '*' im Zeichensatz?

```
printf ("%d\n", '*');
```

(normalerweise: ASCII)

- Ist **char** **ch** ein Großbuchstabe?

```
if (ch >= 'A' && ch <= 'Z')
```

...

- Groß- in Kleinbuchstaben umwandeln

```
ch += 'a' - 'A';
```


2.12 Strukturen

```
#include <stdio.h>
```

```
typedef struct
```

```
{
```

```
    char day, month;
```

```
    int year;
```

```
}
```

```
date;
```

```
int main (void)
```

```
{
```

```
    date today = { 7, 11, 2024 };
```

```
    printf ("%d.%d.%d\n", today.day, today.month, today.year);
```

```
    return 0;
```

```
}
```

2.12 Strukturen

```
#include <stdio.h>
```

```
typedef struct
```

```
{
```

```
    char day, month;
```

```
    int year;
```

```
}
```

```
date;
```

```
void set_date (date *d)
```

```
{
```

```
    (*d).day = 7;
```

```
    (*d).month = 11;
```

```
    (*d).year = 2024;
```

```
}
```

```
int main (void)
```

```
{
```

```
    date today;
```

```
    set_date (&today);
```

```
    printf ("%d.%d.%d\n", today.day,  
            today.month, today.year);
```

```
    return 0;
```

```
}
```

2.12 Strukturen

```
#include <stdio.h>
```

```
typedef struct
```

```
{
```

```
    char day, month;
```

```
    int year;
```

```
}
```

```
date;
```

```
void set_date (date *d)
```

```
{
```

```
    d->day = 7;
```

```
    d->month = 11;
```

```
    d->year = 2024;
```

```
}
```

foo->bar ist Abkürzung für (*foo).bar

```
int main (void)
```

```
{
```

```
    date today;
```

```
    set_date (&today);
```

```
    printf ("%d.%d.%d\n", today.day,  
            today.month, today.year);
```

```
    return 0;
```

```
}
```

2.12 Strukturen

```
#include <stdio.h>
```

```
typedef struct
```

```
{  
    char day, month;  
    int year;  
}  
date;
```

```
void set_date (date *d)
```

```
{  
    d->day = 7;  
    d->month = 11;  
    d->year = 2024;  
}
```

`foo->bar` ist Abkürzung für `(*foo).bar`

Eine Funktion, die mit einem **struct** arbeitet, kann man eine *Methode* des **struct** nennen.

```
int main (void)
```

```
{  
    date today;  
    set_date (&today);  
    printf ("%d.%d.%d\n", today.day,  
            today.month, today.year);  
    return 0;  
}
```

2.13 Dateien und Fehlerbehandlung

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    FILE *f = fopen ("fhello.txt", "w");
```

```
    fprintf (f, "Hello, world!\n");
```

```
    fclose (f);
```

```
    return 0;
```

```
}
```

2.13 Dateien und Fehlerbehandlung

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    FILE *f = fopen ("fhello.txt", "w");
```

```
    if (f)
```

```
    {
```

```
        fprintf (f, "Hello,_world!\n");
```

```
        fclose (f);
```

```
    }
```

```
    return 0;
```

```
}
```

- Wenn die Datei nicht geöffnet werden kann, gibt `fopen()` den Wert `NULL` zurück.

2.13 Dateien und Fehlerbehandlung

```
#include <stdio.h>
```

```
#include <errno.h>
```

```
int main (void)
```

```
{
```

```
    FILE *f = fopen ("fhello.txt", "w");
```

```
    if (f)
```

```
    {
```

```
        fprintf (f, "Hello,_world!\n");
```

```
        fclose (f);
```

```
    }
```

```
    else
```

```
        fprintf (stderr, "error_#%d\n", errno);
```

```
    return 0;
```

```
}
```

- Wenn die Datei nicht geöffnet werden kann, gibt `fopen()` den Wert `NULL` zurück.
- Die globale Variable `int errno` enthält dann die Nummer des Fehlers. Benötigt: `#include <errno.h>`

2.13 Dateien und Fehlerbehandlung

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
```

```
int main (void)
{
    FILE *f = fopen ("fhello.txt", "w");
    if (f)
    {
        fprintf (f, "Hello, _world!\n");
        fclose (f);
    }
    else
    {
        char *msg = strerror (errno);
        fprintf (stderr, "%s\n", msg);
    }
    return 0;
}
```

- Wenn die Datei nicht geöffnet werden kann, gibt `fopen()` den Wert `NULL` zurück.
- Die globale Variable `int errno` enthält dann die Nummer des Fehlers. Benötigt: `#include <errno.h>`
- Die Funktion `strerror()` wandelt `errno` in einen Fehlermeldungstext um. Benötigt: `#include <string.h>`

2.13 Dateien und Fehlerbehandlung

```
#include <stdio.h>
```

```
#include <errno.h>
```

```
#include <error.h>
```

```
int main (void)
```

```
{
```

```
    FILE *f = fopen ("fhello.txt", "w");
```

```
    if (!f)
```

```
        error (errno, errno, "cannot_open_file");
```

```
    fprintf (f, "Hello,_world!\n");
```

```
    fclose (f);
```

```
    return 0;
```

```
}
```

- Wenn die Datei nicht geöffnet werden kann, gibt `fopen()` den Wert `NULL` zurück.
- Die globale Variable `int errno` enthält dann die Nummer des Fehlers. Benötigt: `#include <errno.h>`
- Die Funktion `strerror()` wandelt `errno` in einen Fehlermeldungstext um. Benötigt: `#include <string.h>`
- Die Funktion `error()` gibt eine Fehlermeldung aus und beendet das Programm. Benötigt: `#include <error.h>`

2.13 Dateien und Fehlerbehandlung

```
#include <stdio.h>
```

```
#include <errno.h>
```

```
#include <error.h>
```

```
int main (void)
```

```
{
```

```
    FILE *f = fopen ("fhello.txt", "w");
```

```
    if (!f)
```

```
        error (errno, errno, "cannot_open_file");
```

```
    fprintf (f, "Hello,_world!\n");
```

```
    fclose (f);
```

```
    return 0;
```

```
}
```

- Wenn die Datei nicht geöffnet werden kann, gibt `fopen()` den Wert `NULL` zurück.
- Die globale Variable `int errno` enthält dann die Nummer des Fehlers. Benötigt: `#include <errno.h>`
- Die Funktion `strerror()` wandelt `errno` in einen Fehlermeldungstext um. Benötigt: `#include <string.h>`
- Die Funktion `error()` gibt eine Fehlermeldung aus und beendet das Programm. Benötigt: `#include <error.h>`
- **Niemals Fehler einfach ignorieren!**

2.14 Parameter des Hauptprogramms

```
#include <stdio.h>
```

```
int main (int argc, char **argv)
{
    printf ("argc=%d\n", argc);
    for (int i = 0; i < argc; i++)
        printf ("argv[%d]=\n%s\n", i, argv[i]);
    return 0;
}
```

2.14 Parameter des Hauptprogramms

```
#include <stdio.h>
```

```
int main (int argc, char **argv)
{
    printf ("argc=%d\n", argc);
    for (int i = 0; *argv; i++, argv++)
        printf ("argv[%d]=\n", i, *argv);
    return 0;
}
```

2.15 String-Operationen

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello,_world!\n";
```

```
    printf ("%s\n", hello);
```

```
    printf ("%zd\n", strlen (hello));
```

```
    printf ("%s\n", hello + 7);
```

```
    printf ("%zd\n", strlen (hello + 7));
```

```
    hello[5] = 0;
```

```
    printf ("%s\n", hello);
```

```
    printf ("%zd\n", strlen (hello));
```

```
    return 0;
```

```
}
```

2.15 String-Operationen

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main (void)
```

```
{
```

```
    char *anton = "Anton";
```

```
    char *zacharias = "Zacharias";
```

```
    printf ("%d\n", strcmp (anton, zacharias));
```

```
    printf ("%d\n", strcmp (zacharias, anton));
```

```
    printf ("%d\n", strcmp (anton, anton));
```

```
    char buffer[100] = "Huber_";
```

```
    strcat (buffer, anton);
```

```
    printf ("%s\n", buffer);
```

```
    return 0;
```

```
}
```

2.15 String-Operationen

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main (void)
```

```
{
```

```
    char buffer[100] = "";
```

```
    sprintf (buffer, "Die_Antwort_lautet:%d", 42);
```

```
    printf ("%s\n", buffer);
```

```
    char *answer = strstr (buffer, "Antwort");
```

```
    printf ("%s\n", answer);
```

```
    printf ("found_at:%zd\n", answer - buffer);
```

```
    return 0;
```

```
}
```

2.15 String-Operationen

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main (void)
```

```
{
```

```
    char buffer[100] = "";
```

```
    snprintf (buffer, 100, "Die_Antwort_lautet:_%d", 42);
```

```
    printf ("%s\n", buffer);
```

```
    char *answer = strstr (buffer, "Antwort");
```

```
    printf ("%s\n", answer);
```

```
    printf ("found_at:_%zd\n", answer - buffer);
```

```
    return 0;
```

```
}
```


2 Einführung in C

Sprachelemente weitgehend komplett

Es fehlen:

- Ergänzungen (z. B. ternärer Operator, **union**, **unsigned**, **volatile**)
- Bibliotheksfunktionen (z. B. **malloc()**)

—> werden eingeführt, wenn wir sie brauchen

- Konzepte (z. B. rekursive Datenstrukturen, Klassen selbst bauen)

—> werden eingeführt, wenn wir sie brauchen, oder:

—> Literatur

(z. B. Wikibooks: C-Programmierung,
Dokumentation zu Compiler und Bibliotheken)

- Praxiserfahrung

—> Übung und Praktikum: nur Einstieg

—> selbständig arbeiten

3 Bibliotheken

3.1 Der Präprozessor

#include: Text einbinden

3 Bibliotheken

3.1 Der Präprozessor

#include: Text einbinden

- **#include** <stdio.h>: Standard-Verzeichnisse – Standard-Header

3 Bibliotheken

3.1 Der Präprozessor

#include: Text einbinden

- **#include <stdio.h>**: Standard-Verzeichnisse – Standard-Header
- **#include "answer.h"**: auch aktuelles Verzeichnis – eigene Header

3 Bibliotheken

3.1 Der Präprozessor

#include: Text einbinden

- **#include <stdio.h>**: Standard-Verzeichnisse – Standard-Header
- **#include "answer.h"**: auch aktuelles Verzeichnis – eigene Header

#define SIX 6: Text ersetzen lassen – Konstante definieren

3 Bibliotheken

3.1 Der Präprozessor

#include: Text einbinden

- **#include <stdio.h>**: Standard-Verzeichnisse – Standard-Header
- **#include "answer.h"**: auch aktuelles Verzeichnis – eigene Header

#define SIX 6: Text ersetzen lassen – Konstante definieren

- Kein Semikolon!

3 Bibliotheken

3.1 Der Präprozessor

#include: Text einbinden

- **#include <stdio.h>**: Standard-Verzeichnisse – Standard-Header
- **#include "answer.h"**: auch aktuelles Verzeichnis – eigene Header

#define SIX 6: Text ersetzen lassen – Konstante definieren

- Kein Semikolon!
- Berechnungen in Klammern setzen:
#define SIX (1 + 5)

3 Bibliotheken

3.1 Der Präprozessor

#include: Text einbinden

- **#include <stdio.h>**: Standard-Verzeichnisse – Standard-Header
- **#include "answer.h"**: auch aktuelles Verzeichnis – eigene Header

#define SIX 6: Text ersetzen lassen – Konstante definieren

- Kein Semikolon!
- Berechnungen in Klammern setzen:
#define SIX (1 + 5)
- Konvention: Großbuchstaben

3 Bibliotheken

3.2 Bibliotheken einbinden

Inhalt der Header-Datei: externe Deklarationen

extern int answer (**void**);

extern int printf (__const **char** *__restrict __format, ...);

Funktion wird „anderswo“ definiert

- separater C-Quelltext: mit an `gcc` übergeben
- Zusammenfügen zu ausführbarem Programm durch den *Linker*
- vorcompilierte Bibliothek: `-lfoo`
= Datei `libfoo.a` in Standard-Verzeichnis

3.3 Bibliothek verwenden (Beispiel: GTK)

- **#include** <gtk/gtk.h>

3.3 Bibliothek verwenden (Beispiel: GTK)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß:

3.3 Bibliothek verwenden (Beispiel: GTK)

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

- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß:

```
$ pkg-config --cflags --libs gtk4
-I/usr/include/gtk-4.0 -I/usr/include/pango-1.0 -I/usr/
include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/i
nclude -I/usr/include/harfbuzz -I/usr/include/freetype2
-I/usr/include/libpng16 -I/usr/include/libmount -I/usr/
include/blkid -I/usr/include/fribidi -I/usr/include/cai
ro -I/usr/include/pixman-1 -I/usr/include/gdk-pixbuf-2.
0 -I/usr/include/x86_64-linux-gnu -I/usr/include/graphene-1.0 -I/usr/lib/x86_64-linux-gnu/graphene-1.0/include
-mfpmath=sse -msse -msse2 -pthread -lgtk-4 -lpangocairo
-1.0 -lpango-1.0 -lharfbuzz -lgdk_pixbuf-2.0 -lcairo-go
bject -lcairo -lgraphene-1.0 -lgio-2.0 -lgobject-2.0 -l
glib-2.0
```

3.3 Bibliothek verwenden (Beispiel: GTK)

- **#include** <gtk/gtk.h>
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

→ Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c -I/usr/include/gtk-4.0 -I/usr/include/pango-1.0 -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -I/usr/include/harfbuzz -I/usr/include/freetype2 -I/usr/include/libpng16 -I/usr/include/libmount -I/usr/include/blkid -I/usr/include/fribidi -I/usr/include/cairo -I/usr/include/pixman-1 -I/usr/include/gdk-pixbuf-2.0 -I/usr/include/x86_64-linux-gnu -I/usr/include/graphene-1.0 -I/usr/lib/x86_64-linux-gnu/graphene-1.0/include -mfpmath=sse -msse -msse2 -pthread -lgtk-4 -lpangocairo-1.0 -lpango-1.0 -lharfbuzz -lgdk_pixbuf-2.0 -lcairo-gobject -lcairo -lgraphene-1.0 -lgio-2.0 -lgobject-2.0 -l glib-2.0 -o hello-gtk
```

3.3 Bibliothek verwenden (Beispiel: GTK)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

→ Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c $(pkg-config --cflags --libs  
    gtk4) -o hello-gtk
```

Optionen:

u. a. viele Include-Verzeichnisse:

`-I/usr/include/gtk-4.0`

Bibliotheken:

u. a. `-lgtk-4`

3.3 Bibliothek verwenden (Beispiel: GTK)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

—> Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c $(pkg-config --cflags --libs  
    gtk4) -o hello-gtk
```

- Auf manchen Plattformen kommt es auf die Reihenfolge an:

```
$ gcc -Wall -O $(pkg-config --cflags gtk4) \  
    hello-gtk.c $(pkg-config --libs gtk4) \  
    -o hello-gtk
```

(Backslash = „Es geht in der nächsten Zeile weiter.“)

3.4 Callbacks

Selbst geschriebene Funktion übergeben: *Callback*

```
static void hello (GtkWidget *this, gpointer user_data)
{
    char *world = user_data;
    printf ("Hello, %s!\n", world);
}
```

...

```
g_signal_connect (button, "clicked", G_CALLBACK (hello), "world");
```

—> GTK ruft immer dann, wenn der Button betätigt wurde,
die Funktion `hello` auf.

3.4 Callbacks

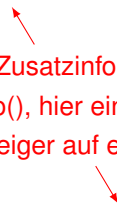
Selbst geschriebene Funktion übergeben: *Callback*

```
static void hello (GtkWidget *this, gpointer user_data)
{
    char *world = user_data;
    printf ("Hello, %s!\n", world);
}
```

...

```
g_signal_connect (button, "clicked", G_CALLBACK (hello), "world");
```

optionale Zusatzinformationen
für hello(), hier ein String
oft ein Zeiger auf ein struct



→ GTK ruft immer dann, wenn der Button betätigt wurde,
die Funktion `hello` auf.

3.4 Callbacks

Selbst geschriebene Funktion übergeben: *Callback*

```
static void draw (GtkDrawingArea *drawing_area, cairo_t *c,  
                  int width, int height, gpointer user_data)
```

```
{  
    /* Zeichenbefehle */
```

```
    ...
```

```
}
```

```
...
```

```
gtk_drawing_area_set_draw_func (GTK_DRAWING_AREA (drawing_area),  
                                draw, NULL, NULL);
```

→ GTK ruft immer dann, wenn es etwas zu zeichnen gibt,
die Funktion `draw` auf.

3.4 Callbacks

Selbst geschriebene Funktion übergeben: *Callback*


```
static void draw (GtkDrawingArea *drawing_area, cairo_t *c,  
                  int width, int height, gpointer user_data)
```

```
{  
    /* Zeichenbefehle */  
    ...  
}
```

```
...
```

```
gtk_drawing_area_set_draw_func (GTK_DRAWING_AREA (drawing_area),  
                                draw, NULL, NULL);
```

repräsentiert den
Bildschirm, auf den
gezeichnet werden soll



→ GTK ruft immer dann, wenn es etwas zu zeichnen gibt,
die Funktion `draw` auf.

3.5 Projekt organisieren: make

- Regeln
- Makros

3.5 Projekt organisieren: make

- Regeln

```
philosophy: philosophy.o answer.o  
gcc philosophy.o answer.o -o philosophy
```

```
answer.o: answer.c answer.h  
gcc -Wall -O answer.c -c
```

```
philosophy.o: philosophy.c answer.h  
gcc -Wall -O philosophy.c -c
```

- Makros

3.5 Projekt organisieren: make

- Regeln
- Makros

TARGET = philosophy

OBJECTS = philosophy.o answer.o

HEADERS = answer.h

CFLAGS = -Wall -O

\$(TARGET): \$(OBJECTS)

gcc \$(OBJECTS) -o \$(TARGET)

answer.o: answer.c \$(HEADERS)

gcc \$(CFLAGS) answer.c -c

philosophy.o: philosophy.c \$(HEADERS)

gcc \$(CFLAGS) philosophy.c -c

clean:

rm -f \$(OBJECTS) \$(TARGET)

3.5 Projekt organisieren: make

- explizite und implizite Regeln

TARGET = philosophy

OBJECTS = philosophy.o answer.o

HEADERS = answer.h

CFLAGS = -Wall -O

\$(TARGET): \$(OBJECTS)

gcc \$(OBJECTS) -o \$(TARGET)

%.o: %.c \$(HEADERS)

gcc \$(CFLAGS) \$< -c

clean:

rm -f \$(OBJECTS) \$(TARGET)

- Makros

3.5 Projekt organisieren: make

- explizite und implizite Regeln
- Makros

→ 3 Sprachen: C, Präprozessor, make

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

21. November 2024

Hardwarenahe Programmierung

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

1 Einführung

2 Einführung in C

...

2.10 Zeiger

2.11 Arrays und Strings

2.12 Strukturen

2.13 Dateien und Fehlerbehandlung

2.14 Parameter des Hauptprogramms

2.15 String-Operationen

3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliotheken verwenden

3.4 Callbacks

3.5 Projekt organisieren: make

4 Hardwarenahe Programmierung

...

2.15 String-Operationen

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello,_world!\n";
```

```
    printf ("%s\n", hello);
```

```
    printf ("%zd\n", strlen (hello));
```

```
    printf ("%s\n", hello + 7);
```

```
    printf ("%zd\n", strlen (hello + 7));
```

```
    hello[5] = 0;
```

```
    printf ("%s\n", hello);
```

```
    printf ("%zd\n", strlen (hello));
```

```
    return 0;
```

```
}
```

2.15 String-Operationen

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main (void)
```

```
{
```

```
    char *anton = "Anton";
```

```
    char *zacharias = "Zacharias";
```

```
    printf ("%d\n", strcmp (anton, zacharias));
```

```
    printf ("%d\n", strcmp (zacharias, anton));
```

```
    printf ("%d\n", strcmp (anton, anton));
```

```
    char buffer[100] = "Huber_";
```

```
    strcat (buffer, anton);
```

```
    printf ("%s\n", buffer);
```

```
    return 0;
```

```
}
```

2.15 String-Operationen

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main (void)
```

```
{
```

```
    char buffer[100] = "";
```

```
    sprintf (buffer, "Die_Antwort_lautet:%d", 42);
```

```
    printf ("%s\n", buffer);
```

```
    char *answer = strstr (buffer, "Antwort");
```

```
    printf ("%s\n", answer);
```

```
    printf ("found_at:%zd\n", answer - buffer);
```

```
    return 0;
```

```
}
```

2.15 String-Operationen

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main (void)
```

```
{
```

```
    char buffer[100] = "";
```

```
    snprintf (buffer, 100, "Die_Antwort_lautet:_%d", 42);
```

```
    printf ("%s\n", buffer);
```

```
    char *answer = strstr (buffer, "Antwort");
```

```
    printf ("%s\n", answer);
```

```
    printf ("found_at:_%zd\n", answer - buffer);
```

```
    return 0;
```

```
}
```

2 Einführung in C

Sprachelemente weitgehend komplett

Es fehlen:

- Ergänzungen (z. B. ternärer Operator, **union**, **unsigned**, **volatile**)
- Bibliotheksfunktionen (z. B. **malloc()**)

—> werden eingeführt, wenn wir sie brauchen

- Konzepte (z. B. rekursive Datenstrukturen, Klassen selbst bauen)

—> werden eingeführt, wenn wir sie brauchen, oder:

—> Literatur

(z. B. Wikibooks: C-Programmierung,
Dokumentation zu Compiler und Bibliotheken)

- Praxiserfahrung

—> Übung und Praktikum: nur Einstieg

—> selbständig arbeiten

3 Bibliotheken

3.1 Der Präprozessor

#include: Text einbinden

3 Bibliotheken

3.1 Der Präprozessor

#include: Text einbinden

- **#include** <stdio.h>: Standard-Verzeichnisse – Standard-Header

3 Bibliotheken

3.1 Der Präprozessor

#include: Text einbinden

- **#include <stdio.h>**: Standard-Verzeichnisse – Standard-Header
- **#include "answer.h"**: auch aktuelles Verzeichnis – eigene Header

3 Bibliotheken

3.1 Der Präprozessor

#include: Text einbinden

- **#include <stdio.h>**: Standard-Verzeichnisse – Standard-Header
- **#include "answer.h"**: auch aktuelles Verzeichnis – eigene Header

#define SIX 6: Text ersetzen lassen – Konstante definieren

3 Bibliotheken

3.1 Der Präprozessor

#include: Text einbinden

- **#include <stdio.h>**: Standard-Verzeichnisse – Standard-Header
- **#include "answer.h"**: auch aktuelles Verzeichnis – eigene Header

#define SIX 6: Text ersetzen lassen – Konstante definieren

- Kein Semikolon!

3 Bibliotheken

3.1 Der Präprozessor

#include: Text einbinden

- **#include** <stdio.h>: Standard-Verzeichnisse – Standard-Header
- **#include** "answer.h": auch aktuelles Verzeichnis – eigene Header

#define SIX 6: Text ersetzen lassen – Konstante definieren

- Kein Semikolon!
- Berechnungen in Klammern setzen:
#define SIX (1 + 5)

3 Bibliotheken

3.1 Der Präprozessor

#include: Text einbinden

- **#include** <stdio.h>: Standard-Verzeichnisse – Standard-Header
- **#include** "answer.h": auch aktuelles Verzeichnis – eigene Header

#define SIX 6: Text ersetzen lassen – Konstante definieren

- Kein Semikolon!
- Berechnungen in Klammern setzen:
#define SIX (1 + 5)
- Konvention: Großbuchstaben

3 Bibliotheken

3.2 Bibliotheken einbinden

Inhalt der Header-Datei: externe Deklarationen

extern int answer (**void**);

extern int printf (__const **char** *__restrict __format, ...);

Funktion wird „anderswo“ definiert

- separater C-Quelltext: mit an `gcc` übergeben
- Zusammenfügen zu ausführbarem Programm durch den *Linker*
- vorcompilierte Bibliothek: `-lfoo`
= Datei `libfoo.a` in Standard-Verzeichnis

3.3 Bibliothek verwenden (Beispiel: GTK)

- **#include** <gtk/gtk.h>

3.3 Bibliothek verwenden (Beispiel: GTK)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß:

3.3 Bibliothek verwenden (Beispiel: GTK)

- **#include** <gtk/gtk.h>
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß:

```
$ pkg-config --cflags --libs gtk4
-I/usr/include/gtk-4.0 -I/usr/include/pango-1.0 -I/usr/
include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/i
nclude -I/usr/include/harfbuzz -I/usr/include/freetype2
-I/usr/include/libpng16 -I/usr/include/libmount -I/usr/
include/blkid -I/usr/include/fribidi -I/usr/include/cai
ro -I/usr/include/pixman-1 -I/usr/include/gdk-pixbuf-2.
0 -I/usr/include/x86_64-linux-gnu -I/usr/include/graphene-1.0 -I/usr/lib/x86_64-linux-gnu/graphene-1.0/include
-mfpmath=sse -msse -msse2 -pthread -lgtk-4 -lpangocairo
-1.0 -lpango-1.0 -lharfbuzz -lgdk_pixbuf-2.0 -lcairo-go
bject -lcairo -lgraphene-1.0 -lgio-2.0 -lgobject-2.0 -l
glib-2.0
```

3.3 Bibliothek verwenden (Beispiel: GTK)

- **#include** <gtk/gtk.h>
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

→ Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c -I/usr/include/gtk-4.0 -I/usr/include/pango-1.0 -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -I/usr/include/harfbuzz -I/usr/include/freetype2 -I/usr/include/libpng16 -I/usr/include/libmount -I/usr/include/blkid -I/usr/include/fribidi -I/usr/include/cairo -I/usr/include/pixman-1 -I/usr/include/gdk-pixbuf-2.0 -I/usr/include/x86_64-linux-gnu -I/usr/include/graphene-1.0 -I/usr/lib/x86_64-linux-gnu/graphene-1.0/include -mfpmath=sse -msse -msse2 -pthread -lgtk-4 -lpangocairo-1.0 -lpango-1.0 -lharfbuzz -lgdk_pixbuf-2.0 -lcairo-gobject -lcairo -lgraphene-1.0 -lgio-2.0 -lgobject-2.0 -l glib-2.0 -o hello-gtk
```

3.3 Bibliothek verwenden (Beispiel: GTK)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

→ Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c $(pkg-config --cflags --libs  
    gtk4) -o hello-gtk
```

Optionen:

u. a. viele Include-Verzeichnisse:

`-I/usr/include/gtk-4.0`

Bibliotheken:

u. a. `-lgtk-4`

3.3 Bibliothek verwenden (Beispiel: GTK)

- **#include** <gtk/gtk.h>
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

—> Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c $(pkg-config --cflags --libs  
    gtk4) -o hello-gtk
```

- Auf manchen Plattformen kommt es auf die Reihenfolge an:

```
$ gcc -Wall -O $(pkg-config --cflags gtk4) \  
    hello-gtk.c $(pkg-config --libs gtk4) \  
    -o hello-gtk
```

(Backslash = „Es geht in der nächsten Zeile weiter.“)

3.4 Callbacks

Selbst geschriebene Funktion übergeben: *Callback*

```
static void hello (GtkWidget *this, gpointer user_data)
{
    char *world = user_data;
    printf ("Hello, %s!\n", world);
}
```

...

```
g_signal_connect (button, "clicked", G_CALLBACK (hello), "world");
```

→ GTK ruft immer dann, wenn der Button betätigt wurde,
die Funktion `hello` auf.

3.4 Callbacks

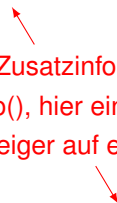
Selbst geschriebene Funktion übergeben: *Callback*

```
static void hello (GtkWidget *this, gpointer user_data)
{
    char *world = user_data;
    printf ("Hello, %s!\n", world);
}
```

...

```
g_signal_connect (button, "clicked", G_CALLBACK (hello), "world");
```

optionale Zusatzinformationen
für hello(), hier ein String
oft ein Zeiger auf ein struct



→ GTK ruft immer dann, wenn der Button betätigt wurde,
die Funktion `hello` auf.

3.4 Callbacks

Selbst geschriebene Funktion übergeben: *Callback*

```
static void draw (GtkDrawingArea *drawing_area, cairo_t *c,  
                  int width, int height, gpointer user_data)
```

```
{  
    /* Zeichenbefehle */
```

```
    ...
```

```
}
```

```
...
```

```
gtk_drawing_area_set_draw_func (GTK_DRAWING_AREA (drawing_area),  
                                draw, NULL, NULL);
```

→ GTK ruft immer dann, wenn es etwas zu zeichnen gibt,
die Funktion `draw` auf.

3.4 Callbacks

Selbst geschriebene Funktion übergeben: *Callback*


```
static void draw (GtkDrawingArea *drawing_area, cairo_t *c,  
                  int width, int height, gpointer user_data)
```

```
{  
    /* Zeichenbefehle */  
    ...  
}
```

```
...
```

```
gtk_drawing_area_set_draw_func (GTK_DRAWING_AREA (drawing_area),  
                                draw, NULL, NULL);
```

repräsentiert den
Bildschirm, auf den
gezeichnet werden soll



→ GTK ruft immer dann, wenn es etwas zu zeichnen gibt,
die Funktion `draw` auf.

3.5 Projekt organisieren: make

- Regeln
- Makros

3.5 Projekt organisieren: make

- Regeln

```
philosophy: philosophy.o answer.o  
gcc philosophy.o answer.o -o philosophy
```

```
answer.o: answer.c answer.h  
gcc -Wall -O answer.c -c
```

```
philosophy.o: philosophy.c answer.h  
gcc -Wall -O philosophy.c -c
```

- Makros

3.5 Projekt organisieren: make

- Regeln
- Makros

TARGET = philosophy

OBJECTS = philosophy.o answer.o

HEADERS = answer.h

CFLAGS = -Wall -O

\$(TARGET): \$(OBJECTS)

gcc \$(OBJECTS) -o \$(TARGET)

answer.o: answer.c \$(HEADERS)

gcc \$(CFLAGS) answer.c -c

philosophy.o: philosophy.c \$(HEADERS)

gcc \$(CFLAGS) philosophy.c -c

clean:

rm -f \$(OBJECTS) \$(TARGET)

3.5 Projekt organisieren: make

- explizite und implizite Regeln

TARGET = philosophy

OBJECTS = philosophy.o answer.o

HEADERS = answer.h

CFLAGS = -Wall -O

\$(TARGET): \$(OBJECTS)

gcc \$(OBJECTS) -o \$(TARGET)

%.o: %.c \$(HEADERS)

gcc \$(CFLAGS) \$< -c

clean:

rm -f \$(OBJECTS) \$(TARGET)

- Makros

3.5 Projekt organisieren: make

- explizite und implizite Regeln
- Makros

→ 3 Sprachen: C, Präprozessor, make

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)

- Computer rechnen im Binärsystem.
- Für viele Anwendungen (z. B. I/O-Ports, Grafik, ...) ist es notwendig, Bits in Zahlen einzeln ansprechen zu können.

4.1.1 Zahlensysteme

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

- Oktal- und Hexadezimalzahlen lassen sich ziffernweise in Binär-Zahlen umrechnen.
- Hexadezimalzahlen sind eine Kurzschreibweise für Binärzahlen, gruppiert zu jeweils 4 Bits.
- Oktalzahlen sind eine Kurzschreibweise für Binärzahlen, gruppiert zu jeweils 3 Bits.
- Trotz Taschenrechner u. ä. lohnt es sich, die o. a. Umrechnungstabelle **auswendig** zu kennen.

4.1.2 Bit-Operationen in C

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

4.1.2 Bit-Operationen in C

C-Datentypen für Bit-Operationen:

#include <stdint.h>

	8 Bit	16 Bit	32 Bit	64 Bit
mit Vorzeichen	int8_t	int16_t	int32_t	int64_t
ohne Vorzeichen	uint8_t	uint16_t	uint32_t	uint64_t

Ausgabe:

#include <stdio.h>

#include <stdint.h>

#include <inttypes.h>

...

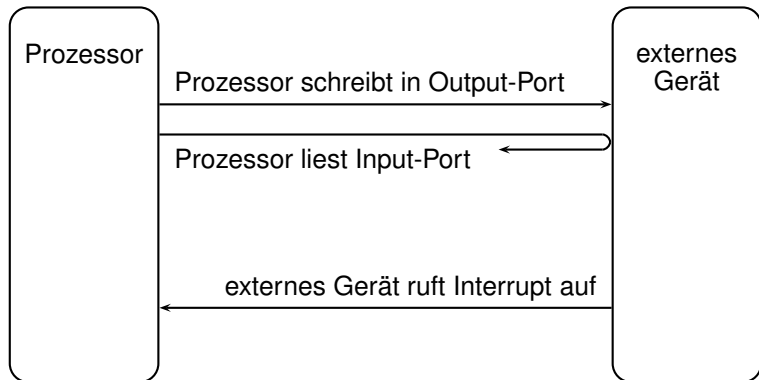
uint64_t x = 42;

printf ("Die_Antwort_lautet:_% " PRIu64 "\n", x);

4.2 I/O-Ports

4.3 Interrupts

Kommunikation mit externen Geräten



4.2 I/O-Ports

In Output-Port schreiben = Aktoren ansteuern

Beispiel: LED

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

```
...
```

```
DDRC = 0x70;    binär: 0111 0000
```

```
PORTC = 0x40;   binär: 0100 0000
```

Herstellerspezifisch!

DDR = Data Direction Register

Bit = 1 für Output-Port

Bit = 0 für Input-Port

Details: siehe Datenblatt und Schaltplan

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 */
```

Herstellerspezifisch!

DDR = Data Direction Register

Bit = 1 für Output-Port

Bit = 0 für Input-Port

Details: siehe Datenblatt und Schaltplan

Praktikumsaufgabe: Druckknopfampel

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

28. November 2024

Hardwarenahe Programmierung

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

1 Einführung

2 Einführung in C

3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliotheken verwenden

3.4 Callbacks

3.5 Projekt organisieren: make

4 Hardwarenahe Programmierung

4.1 Bit-Operationen

4.2 I/O-Ports

4.3 Interrupts

4.4 volatile-Variable

...

5 Algorithmen

6 Objektorientierte Programmierung

7 Datenstrukturen

3 Bibliotheken

3.1 Der Präprozessor

#include: Text einbinden

- **#include** <stdio.h>: Standard-Verzeichnisse – Standard-Header
- **#include** "answer.h": auch aktuelles Verzeichnis – eigene Header

#define SIX 6: Text ersetzen lassen – Konstante definieren

- Kein Semikolon!
- Berechnungen in Klammern setzen:
#define SIX (1 + 5)
- Konvention: Großbuchstaben

3 Bibliotheken

3.2 Bibliotheken einbinden

Inhalt der Header-Datei: externe Deklarationen

extern int answer (**void**);

extern int printf (__const **char** *__restrict __format, ...);

Funktion wird „anderswo“ definiert

- separater C-Quelltext: mit an `gcc` übergeben
- Zusammenfügen zu ausführbarem Programm durch den *Linker*
- vorcompilierte Bibliothek: `-lfoo`
= Datei `libfoo.a` in Standard-Verzeichnis

3.3 Bibliothek verwenden (Beispiel: GTK)

- **#include** <gtk/gtk.h>

3.3 Bibliothek verwenden (Beispiel: GTK)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß:

3.3 Bibliothek verwenden (Beispiel: GTK)

- **#include** <gtk/gtk.h>

- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß:

```
$ pkg-config --cflags --libs gtk4
-I/usr/include/gtk-4.0 -I/usr/include/pango-1.0 -I/usr/
include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/i
nclude -I/usr/include/harfbuzz -I/usr/include/freetype2
-I/usr/include/libpng16 -I/usr/include/libmount -I/usr/
include/blkid -I/usr/include/fribidi -I/usr/include/cai
ro -I/usr/include/pixman-1 -I/usr/include/gdk-pixbuf-2.
0 -I/usr/include/x86_64-linux-gnu -I/usr/include/graphene-1.0 -I/usr/lib/x86_64-linux-gnu/graphene-1.0/include
-mfpmath=sse -msse -msse2 -pthread -lgtk-4 -lpangocairo
-1.0 -lpango-1.0 -lharfbuzz -lgdk_pixbuf-2.0 -lcairo-go
bject -lcairo -lgraphene-1.0 -lgio-2.0 -lgobject-2.0 -l
glib-2.0
```

3.3 Bibliothek verwenden (Beispiel: GTK)

- **#include** <gtk/gtk.h>
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

→ Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c -I/usr/include/gtk-4.0 -I/usr/include/pango-1.0 -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -I/usr/include/harfbuzz -I/usr/include/freetype2 -I/usr/include/libpng16 -I/usr/include/libmount -I/usr/include/blkid -I/usr/include/fribidi -I/usr/include/cairo -I/usr/include/pixman-1 -I/usr/include/gdk-pixbuf-2.0 -I/usr/include/x86_64-linux-gnu -I/usr/include/graphene-1.0 -I/usr/lib/x86_64-linux-gnu/graphene-1.0/include -mfpmath=sse -msse -msse2 -pthread -lgtk-4 -lpangocairo -l.0 -lpango-1.0 -lharfbuzz -lgdk_pixbuf-2.0 -lcairo-gobject -lcairo -lgraphene-1.0 -lgio-2.0 -lgobject-2.0 -l glib-2.0 -o hello-gtk
```

3.3 Bibliothek verwenden (Beispiel: GTK)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

→ Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c $(pkg-config --cflags --libs  
    gtk4) -o hello-gtk
```

Optionen:

u. a. viele Include-Verzeichnisse:

`-I/usr/include/gtk-4.0`

Bibliotheken:

u. a. `-lgtk-4`

3.3 Bibliothek verwenden (Beispiel: GTK)

- **#include** <gtk/gtk.h>
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

—> Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c $(pkg-config --cflags --libs  
    gtk4) -o hello-gtk
```

- Auf manchen Plattformen kommt es auf die Reihenfolge an:

```
$ gcc -Wall -O $(pkg-config --cflags gtk4) \  
    hello-gtk.c $(pkg-config --libs gtk4) \  
    -o hello-gtk
```

(Backslash = „Es geht in der nächsten Zeile weiter.“)

3.4 Callbacks

Selbst geschriebene Funktion übergeben: *Callback*

```
static void hello (GtkWidget *this, gpointer user_data)
{
    char *world = user_data;
    printf ("Hello, %s!\n", world);
}
```

...

```
g_signal_connect (button, "clicked", G_CALLBACK (hello), "world");
```

→ GTK ruft immer dann, wenn der Button betätigt wurde,
die Funktion `hello` auf.

3.4 Callbacks

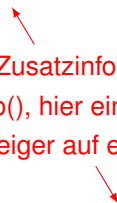
Selbst geschriebene Funktion übergeben: *Callback*

```
static void hello (GtkWidget *this, gpointer user_data)
{
    char *world = user_data;
    printf ("Hello, %s!\n", world);
}
```

...

```
g_signal_connect (button, "clicked", G_CALLBACK (hello), "world");
```

optionale Zusatzinformationen
für hello(), hier ein String
oft ein Zeiger auf ein struct



→ GTK ruft immer dann, wenn der Button betätigt wurde,
die Funktion `hello` auf.

3.4 Callbacks

Selbst geschriebene Funktion übergeben: *Callback*

```
static void draw (GtkDrawingArea *drawing_area, cairo_t *c,  
                  int width, int height, gpointer user_data)
```

```
{  
    /* Zeichenbefehle */
```

```
    ...
```

```
}
```

```
...
```

```
gtk_drawing_area_set_draw_func (GTK_DRAWING_AREA (drawing_area),  
                                draw, NULL, NULL);
```

→ GTK ruft immer dann, wenn es etwas zu zeichnen gibt,
die Funktion `draw` auf.

3.4 Callbacks

Selbst geschriebene Funktion übergeben: *Callback*


```
static void draw (GtkDrawingArea *drawing_area, cairo_t *c,  
                  int width, int height, gpointer user_data)
```

```
{  
    /* Zeichenbefehle */  
    ...  
}
```

```
...
```

```
gtk_drawing_area_set_draw_func (GTK_DRAWING_AREA (drawing_area),  
                                draw, NULL, NULL);
```

repräsentiert den
Bildschirm, auf den
gezeichnet werden soll



→ GTK ruft immer dann, wenn es etwas zu zeichnen gibt,
die Funktion `draw` auf.

3.5 Projekt organisieren: make

- Regeln
- Makros

3.5 Projekt organisieren: make

- Regeln

```
philosophy: philosophy.o answer.o  
gcc philosophy.o answer.o -o philosophy
```

```
answer.o: answer.c answer.h  
gcc -Wall -O answer.c -c
```

```
philosophy.o: philosophy.c answer.h  
gcc -Wall -O philosophy.c -c
```

- Makros

3.5 Projekt organisieren: make

- Regeln
- Makros

TARGET = philosophy

OBJECTS = philosophy.o answer.o

HEADERS = answer.h

CFLAGS = -Wall -O

\$(TARGET): \$(OBJECTS)

gcc \$(OBJECTS) -o \$(TARGET)

answer.o: answer.c \$(HEADERS)

gcc \$(CFLAGS) answer.c -c

philosophy.o: philosophy.c \$(HEADERS)

gcc \$(CFLAGS) philosophy.c -c

clean:

rm -f \$(OBJECTS) \$(TARGET)

3.5 Projekt organisieren: make

- explizite und implizite Regeln

TARGET = philosophy

OBJECTS = philosophy.o answer.o

HEADERS = answer.h

CFLAGS = -Wall -O

\$(TARGET): \$(OBJECTS)

gcc \$(OBJECTS) -o \$(TARGET)

%.o: %.c \$(HEADERS)

gcc \$(CFLAGS) \$< -c

clean:

rm -f \$(OBJECTS) \$(TARGET)

- Makros

3.5 Projekt organisieren: make

- explizite und implizite Regeln
- Makros

→ 3 Sprachen: C, Präprozessor, make

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)

- Computer rechnen im Binärsystem.
- Für viele Anwendungen (z. B. I/O-Ports, Grafik, ...) ist es notwendig, Bits in Zahlen einzeln ansprechen zu können.

4.1.1 Zahlensysteme

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

- Oktal- und Hexadezimalzahlen lassen sich ziffernweise in Binär-Zahlen umrechnen.
- Hexadezimalzahlen sind eine Kurzschreibweise für Binärzahlen, gruppiert zu jeweils 4 Bits.
- Oktalzahlen sind eine Kurzschreibweise für Binärzahlen, gruppiert zu jeweils 3 Bits.
- Trotz Taschenrechner u. ä. lohnt es sich, die o. a. Umrechnungstabelle **auswendig** zu kennen.

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))`

4.1.2 Bit-Operationen in C

C-Datentypen für Bit-Operationen:

#include <stdint.h>

	8 Bit	16 Bit	32 Bit	64 Bit
mit Vorzeichen	int8_t	int16_t	int32_t	int64_t
ohne Vorzeichen	uint8_t	uint16_t	uint32_t	uint64_t

Ausgabe:

#include <stdio.h>

#include <stdint.h>

#include <inttypes.h>

...

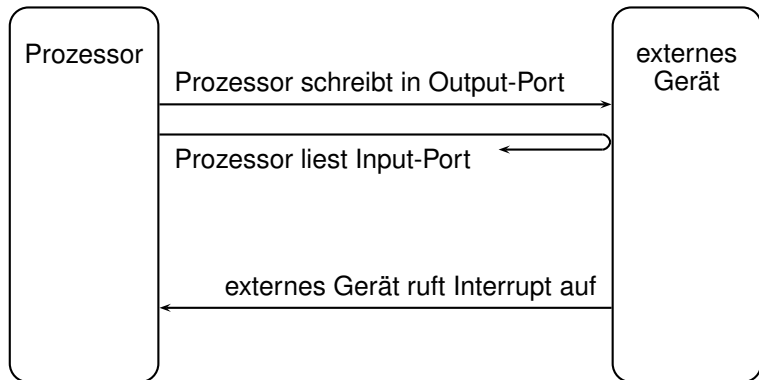
uint64_t x = 42;

printf ("Die_Antwort_lautet:_%" PRIu64 "\n", x);

4.2 I/O-Ports

4.3 Interrupts

Kommunikation mit externen Geräten



4.2 I/O-Ports

In Output-Port schreiben = Aktoren ansteuern

Beispiel: LED

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

```
...
```

```
DDRC = 0x70;    binär: 0111 0000
```

```
PORTC = 0x40;   binär: 0100 0000
```

Herstellerspezifisch!

DDR = Data Direction Register

Bit = 1 für Output-Port

Bit = 0 für Input-Port

Details: siehe Datenblatt und Schaltplan

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 */
```

Herstellerspezifisch!

DDR = Data Direction Register

Bit = 1 für Output-Port

Bit = 0 für Input-Port

Details: siehe Datenblatt und Schaltplan

Praktikumsaufgabe: Druckknopfampel

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

5. Dezember 2024

Hardwarenahe Programmierung

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

1 Einführung

2 Einführung in C

3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliotheken verwenden

3.4 Callbacks

3.5 Projekt organisieren: make

4 Hardwarenahe Programmierung

4.1 Bit-Operationen

4.2 I/O-Ports

4.3 Interrupts

4.4 volatile-Variable

...

5 Algorithmen

6 Objektorientierte Programmierung

7 Datenstrukturen

3.3 Bibliothek verwenden (Beispiel: GTK)

- **#include** <gtk/gtk.h>

3.3 Bibliothek verwenden (Beispiel: GTK)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß:

3.3 Bibliothek verwenden (Beispiel: GTK)

- **#include** <gtk/gtk.h>

- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß:

```
$ pkg-config --cflags --libs gtk4
-I/usr/include/gtk-4.0 -I/usr/include/pango-1.0 -I/usr/
include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/i
nclude -I/usr/include/harfbuzz -I/usr/include/freetype2
-I/usr/include/libpng16 -I/usr/include/libmount -I/usr/
include/blkid -I/usr/include/fribidi -I/usr/include/cai
ro -I/usr/include/pixman-1 -I/usr/include/gdk-pixbuf-2.
0 -I/usr/include/x86_64-linux-gnu -I/usr/include/graphene-1.0 -I/usr/lib/x86_64-linux-gnu/graphene-1.0/include
-mfpmath=sse -msse -msse2 -pthread -lgtk-4 -lpangocairo
-1.0 -lpango-1.0 -lharfbuzz -lgdk_pixbuf-2.0 -lcairo-go
bject -lcairo -lgraphene-1.0 -lgio-2.0 -lgobject-2.0 -l
glib-2.0
```

3.3 Bibliothek verwenden (Beispiel: GTK)

- **#include** <gtk/gtk.h>
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

→ Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c -I/usr/include/gtk-4.0 -I/usr/include/pango-1.0 -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -I/usr/include/harfbuzz -I/usr/include/freetype2 -I/usr/include/libpng16 -I/usr/include/libmount -I/usr/include/blkid -I/usr/include/fribidi -I/usr/include/cairo -I/usr/include/pixman-1 -I/usr/include/gdk-pixbuf-2.0 -I/usr/include/x86_64-linux-gnu -I/usr/include/graphene-1.0 -I/usr/lib/x86_64-linux-gnu/graphene-1.0/include -mfpmath=sse -msse -msse2 -pthread -lgtk-4 -lpangocairo -l.0 -lpango-1.0 -lharfbuzz -lgdk_pixbuf-2.0 -lcairo-gobject -lcairo -lgraphene-1.0 -lgio-2.0 -lgobject-2.0 -l glib-2.0 -o hello-gtk
```

3.3 Bibliothek verwenden (Beispiel: GTK)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

→ Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c $(pkg-config --cflags --libs  
    gtk4) -o hello-gtk
```

Optionen:

u. a. viele Include-Verzeichnisse:

`-I/usr/include/gtk-4.0`

Bibliotheken:

u. a. `-lgtk-4`

3.3 Bibliothek verwenden (Beispiel: GTK)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

—> Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c $(pkg-config --cflags --libs  
    gtk4) -o hello-gtk
```

- Auf manchen Plattformen kommt es auf die Reihenfolge an:

```
$ gcc -Wall -O $(pkg-config --cflags gtk4) \  
    hello-gtk.c $(pkg-config --libs gtk4) \  
    -o hello-gtk
```

(Backslash = „Es geht in der nächsten Zeile weiter.“)

3.4 Callbacks

Selbst geschriebene Funktion übergeben: *Callback*

```
static void hello (GtkWidget *this, gpointer user_data)
{
    char *world = user_data;
    printf ("Hello, %s!\n", world);
}
```

...

```
g_signal_connect (button, "clicked", G_CALLBACK (hello), "world");
```

→ GTK ruft immer dann, wenn der Button betätigt wurde,
die Funktion `hello` auf.

3.4 Callbacks

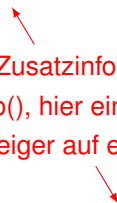
Selbst geschriebene Funktion übergeben: *Callback*

```
static void hello (GtkWidget *this, gpointer user_data)
{
    char *world = user_data;
    printf ("Hello, %s!\n", world);
}
```

...

```
g_signal_connect (button, "clicked", G_CALLBACK (hello), "world");
```

optionale Zusatzinformationen
für hello(), hier ein String
oft ein Zeiger auf ein struct



→ GTK ruft immer dann, wenn der Button betätigt wurde,
die Funktion `hello` auf.

3.4 Callbacks

Selbst geschriebene Funktion übergeben: *Callback*

```
static void draw (GtkDrawingArea *drawing_area, cairo_t *c,  
                  int width, int height, gpointer user_data)
```

```
{  
    /* Zeichenbefehle */
```

```
    ...
```

```
}
```

```
...
```

```
gtk_drawing_area_set_draw_func (GTK_DRAWING_AREA (drawing_area),  
                                draw, NULL, NULL);
```

→ GTK ruft immer dann, wenn es etwas zu zeichnen gibt,
die Funktion `draw` auf.

3.4 Callbacks

Selbst geschriebene Funktion übergeben: *Callback*

```
static void draw (GtkDrawingArea *drawing_area, cairo_t *c,  
                  int width, int height, gpointer user_data)
```

```
{  
    /* Zeichenbefehle */  
    ...  
}
```

```
...
```

```
gtk_drawing_area_set_draw_func (GTK_DRAWING_AREA (drawing_area),  
                                draw, NULL, NULL);
```

repräsentiert den
Bildschirm, auf den
gezeichnet werden soll

→ GTK ruft immer dann, wenn es etwas zu zeichnen gibt,
die Funktion `draw` auf.

3.5 Projekt organisieren: make

- Regeln
- Makros

3.5 Projekt organisieren: make

- Regeln

```
philosophy: philosophy.o answer.o  
gcc philosophy.o answer.o -o philosophy
```

```
answer.o: answer.c answer.h  
gcc -Wall -O answer.c -c
```

```
philosophy.o: philosophy.c answer.h  
gcc -Wall -O philosophy.c -c
```

- Makros

3.5 Projekt organisieren: make

- Regeln
- Makros

TARGET = philosophy

OBJECTS = philosophy.o answer.o

HEADERS = answer.h

CFLAGS = -Wall -O

\$(TARGET): \$(OBJECTS)

gcc \$(OBJECTS) -o \$(TARGET)

answer.o: answer.c \$(HEADERS)

gcc \$(CFLAGS) answer.c -c

philosophy.o: philosophy.c \$(HEADERS)

gcc \$(CFLAGS) philosophy.c -c

clean:

rm -f \$(OBJECTS) \$(TARGET)

3.5 Projekt organisieren: make

- explizite und implizite Regeln

TARGET = philosophy

OBJECTS = philosophy.o answer.o

HEADERS = answer.h

CFLAGS = -Wall -O

\$(TARGET): \$(OBJECTS)

gcc \$(OBJECTS) -o \$(TARGET)

%.o: %.c \$(HEADERS)

gcc \$(CFLAGS) \$< -c

clean:

rm -f \$(OBJECTS) \$(TARGET)

- Makros

3.5 Projekt organisieren: make

- explizite und implizite Regeln
- Makros

→ 3 Sprachen: C, Präprozessor, make

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)

- Computer rechnen im Binärsystem.
- Für viele Anwendungen (z. B. I/O-Ports, Grafik, ...) ist es notwendig, Bits in Zahlen einzeln ansprechen zu können.

4.1.1 Zahlensysteme

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

- Oktal- und Hexadezimalzahlen lassen sich ziffernweise in Binär-Zahlen umrechnen.
- Hexadezimalzahlen sind eine Kurzschreibweise für Binärzahlen, gruppiert zu jeweils 4 Bits.
- Oktalzahlen sind eine Kurzschreibweise für Binärzahlen, gruppiert zu jeweils 3 Bits.
- Trotz Taschenrechner u. ä. lohnt es sich, die o. a. Umrechnungstabelle **auswendig** zu kennen.

4.1.2 Bit-Operationen in C

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

4.1.2 Bit-Operationen in C

C-Datentypen für Bit-Operationen:

#include <stdint.h>

	8 Bit	16 Bit	32 Bit	64 Bit
mit Vorzeichen	int8_t	int16_t	int32_t	int64_t
ohne Vorzeichen	uint8_t	uint16_t	uint32_t	uint64_t

Ausgabe:

#include <stdio.h>

#include <stdint.h>

#include <inttypes.h>

...

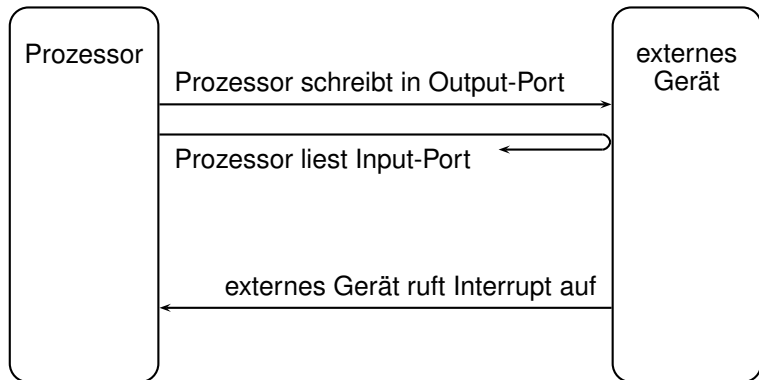
uint64_t x = 42;

printf ("Die_Antwort_lautet:_% " PRIu64 "\n", x);

4.2 I/O-Ports

4.3 Interrupts

Kommunikation mit externen Geräten



4.2 I/O-Ports

In Output-Port schreiben = Aktoren ansteuern

Beispiel: LED

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

```
...
```

```
DDRC = 0x70;    binär: 0111 0000
```

```
PORTC = 0x40;   binär: 0100 0000
```

Herstellerspezifisch!

DDR = Data Direction Register

Bit = 1 für Output-Port

Bit = 0 für Input-Port

Details: siehe Datenblatt und Schaltplan

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 */
```

Herstellerspezifisch!

DDR = Data Direction Register

Bit = 1 für Output-Port

Bit = 0 für Input-Port

Details: siehe Datenblatt und Schaltplan

Praktikumsaufgabe: Druckknopfampel

4.3 Interrupts

Externes Gerät ruft (per Stromsignal) Unterprogramm auf

Zeiger hinterlegen: „Interrupt-Vektor“

Beispiel: eingebaute Uhr

statt Zählschleife (`_delay_ms`):

Hauptprogramm kann

andere Dinge tun

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

... „Dies ist ein Interrupt-Handler.“

Interrupt-Vektor darauf zeigen lassen

```
ISR (TIMER0B_COMP_vect)
```

```
{
```

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

```
}
```

Herstellerspezifisch!

Initialisierung über spezielle Ports: `TCCR0B`, `TIMSK0`

Details: siehe Datenblatt und Schaltplan

4.3 Interrupts

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

Beispiel: Taster

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

...

```
ISR (INT0_vect)
```

```
{  
    PORTD ^= 0x40;  
}
```

statt *Busy Waiting*:
Hauptprogramm kann
andere Dinge tun

Herstellerspezifisch!

Initialisierung über spezielle Ports: **EICRA**, **EIMSK**

Details: siehe Datenblatt und Schaltplan

4.4 volatile-Variable

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

Beispiel: Taster

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

```
...
```

```
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;
```

```
}
```

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?

4.4 volatile-Variable

Was ist eigentlich PORTD?

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

4.4 volatile-Variable

Was ist eigentlich PORTD?

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

```
PORTD = 0x01;
```

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

4.4 volatile-Variable

Was ist eigentlich PORTD?

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

```
PORTD = 0x01;
```

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

Zahl: 0x2B

4.4 volatile-Variable

Was ist eigentlich PORTD?

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

```
PORTD = 0x01;
```

→
$$\underbrace{(*(\text{volatile uint8_t } *) ((0x0B) + 0x20))}_{\substack{\text{Umwandlung in Zeiger} \\ \text{auf } \text{volatile uint8_t}}} = 0x01;$$

Zahl: 0x2B

4.4 volatile-Variable

Was ist eigentlich PORTD?

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

```
PORTD = 0x01;
```

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

↑
Dereferenzierung des Zeigers

Umwandlung in Zeiger
auf **volatile** uint8_t

Zahl: 0x2B

4.4 volatile-Variable

Was ist eigentlich PORTD?

```
avr-gcc -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

4.4 volatile-Variable

Was ist eigentlich PORTD?

```
avr-gcc -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.

Hardwarenahe Programmierung

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

- 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 Binärdarstellung negativer Zahlen
 - 4.7 Binärdarstellung von Gleitkommazahlen
 - 4.8 Speicherausrichtung – Alignment
- 5 Algorithmen**
- 6 Objektorientierte Programmierung**
- 7 Datenstrukturen**

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

12. Dezember 2024

Hardwarenahe Programmierung

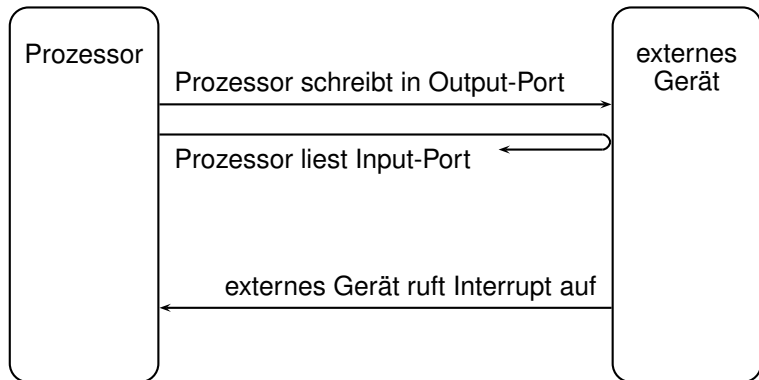
<https://gitlab.cvh-server.de/pgerwinski/hp>

- 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
 - ...
- 5 Algorithmen
 - 5.1 Differentialgleichungen
 - ...
- 6 Objektorientierte Programmierung
- 7 Datenstrukturen

4.2 I/O-Ports

4.3 Interrupts

Kommunikation mit externen Geräten



4.2 I/O-Ports

In Output-Port schreiben = Aktoren ansteuern

Beispiel: LED

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

```
...
```

```
DDRC = 0x70;    binär: 0111 0000
```

```
PORTC = 0x40;    binär: 0100 0000
```

Herstellerspezifisch!

DDR = Data Direction Register

Bit = 1 für Output-Port

Bit = 0 für Input-Port

Details: siehe Datenblatt und Schaltplan

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 */
```

Herstellerspezifisch!

DDR = Data Direction Register

Bit = 1 für Output-Port

Bit = 0 für Input-Port

Details: siehe Datenblatt und Schaltplan

Praktikumsaufgabe: Druckknopfampel

4.3 Interrupts

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

Beispiel: eingebaute Uhr

statt Zählschleife (`_delay_ms`):
Hauptprogramm kann
andere Dinge tun

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

```
...  
ISR (TIMER0B_COMP_vect)  
{  
    PORTD ^= 0x40;  
}
```

„Dies ist ein Interrupt-Handler.“

Interrupt-Vektor darauf zeigen lassen

Herstellerspezifisch!

Initialisierung über spezielle Ports: `TCCR0B`, `TIMSK0`

Details: siehe Datenblatt und Schaltplan

4.3 Interrupts

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

Beispiel: Taster

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

```
...
```

```
ISR (INT0_vect)
```

```
{  
    PORTD ^= 0x40;  
}
```

statt *Busy Waiting*:
Hauptprogramm kann
andere Dinge tun

Herstellerspezifisch!

Initialisierung über spezielle Ports: **EICRA**, **EIMSK**

Details: siehe Datenblatt und Schaltplan

4.4 volatile-Variable

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

Beispiel: Taster

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

```
...
```

```
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;
```

```
}
```

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 -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.

Hardwarenahe Programmierung

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

- 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 Binärdarstellung negativer Zahlen
 - 4.7 Binärdarstellung von Gleitkommazahlen
 - 4.8 Speicherausrichtung – Alignment
- 5 Algorithmen**
- 6 Objektorientierte Programmierung**
- 7 Datenstrukturen**

4.5 Byte-Reihenfolge – Endianness

4.5.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

4.5 Byte-Reihenfolge – Endianness

4.5.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

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

4.5 Byte-Reihenfolge – Endianness

4.5.1 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“

4.5 Byte-Reihenfolge – Endianness

4.5.1 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

4.5 Byte-Reihenfolge – Endianness

4.5.1 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“

4.5 Byte-Reihenfolge – Endianness

4.5.1 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

4.5 Byte-Reihenfolge – Endianness

4.5.1 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

4.5 Byte-Reihenfolge – Endianness

4.5.1 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!**

4.5 Byte-Reihenfolge – Endianness

4.5.1 Konzept

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

Welche Bits liegen wo?

—→ Geschmackssache

... **außer bei Datenaustausch!**

- Dateiformate
- Datenübertragung

4.5 Byte-Reihenfolge – Endianness

4.5.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian

4.5 Byte-Reihenfolge – Endianness

4.5.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian
- ältere AIFF-Dateien (.aiff): Big-Endian
- neuere AIFF-Dateien (.aiff): Little-Endian

4.5 Byte-Reihenfolge – Endianness

4.5.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian
- ältere AIFF-Dateien (.aiff): Big-Endian
- neuere AIFF-Dateien (.aiff): Little-Endian

Grafik-Formate: Reihenfolge der Bits in den Bytes

- PBM-Dateien: Big-Endian
- XBM-Dateien: Little-Endian

4.5 Byte-Reihenfolge – Endianness

4.5.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian
- ältere AIFF-Dateien (.aiff): Big-Endian
- neuere AIFF-Dateien (.aiff): Little-Endian

Grafik-Formate: Reihenfolge der Bits in den Bytes

- PBM-Dateien: Big-Endian, MSB first
- XBM-Dateien: Little-Endian, LSB first

MSB/LSB = most/least significant bit

4.5 Byte-Reihenfolge – Endianness

4.5.3 Datenübertragung

- RS-232 (serielle Schnittstelle): LSB first
- I²C: MSB first
- USB: beides

4.5 Byte-Reihenfolge – Endianness

4.5.3 Datenübertragung

- RS-232 (serielle Schnittstelle): LSB first
- I²C: MSB first
- USB: beides
- Ethernet: LSB first
- TCP/IP (Internet): Big-Endian

4.6 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

4.6 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

→ $255 + 1 = 0$

4.6 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 .

4.6 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

4.6 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

→ $127 + 1 = -128$

4.6 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/>

4.6 Binärdarstellung negativer Zahlen

Frage: Für welche Zahl steht der Speicherinhalt

a3	90
----	----

 (hexadezimal)?

4.6 Binärdarstellung negativer Zahlen

Frage: Für welche Zahl steht der Speicherinhalt

a3	90
----	----

 (hexadezimal)?

Antwort: Das kommt darauf an. ;—)

4.6 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)

4.6 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	

4.7 Speicherausrichtung – Alignment

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

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

4.7 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

4.7 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

4.7 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

4.7 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;
```

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

4.7 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 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t)$$

$$y'(t) = v_y(t)$$

$$v'_x(t) = 0$$

$$v'_y(t) = -g$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \qquad x(t) = \int v_x(t) dt$$

$$y'(t) = v_y(t) \qquad y(t) = \int v_y(t) dt$$

\Rightarrow

$$v'_x(t) = 0 \qquad v_x(t) = \int 0 dt$$

$$v'_y(t) = -g \qquad v_y(t) = \int -g dt$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \qquad x(t) = \int v_x(t) dt$$

$$y'(t) = v_y(t) \qquad y(t) = \int v_y(t) dt$$

\Rightarrow

$$v'_x(t) = 0 \qquad v_x(t) = \int 0 dt = v_{0x}$$

$$v'_y(t) = -g \qquad v_y(t) = \int -g dt$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \qquad x(t) = \int v_x(t) dt = \int v_{0x} dt$$

$$y'(t) = v_y(t) \qquad y(t) = \int v_y(t) dt$$

\Rightarrow

$$v'_x(t) = 0 \qquad v_x(t) = \int 0 dt = v_{0x}$$

$$v'_y(t) = -g \qquad v_y(t) = \int -g dt$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \qquad x(t) = \int v_x(t) dt = \int v_{0x} dt = x_0 + v_{0x} \cdot t$$

$$y'(t) = v_y(t) \qquad y(t) = \int v_y(t) dt$$

\Rightarrow

$$v'_x(t) = 0 \qquad v_x(t) = \int 0 dt = v_{0x}$$

$$v'_y(t) = -g \qquad v_y(t) = \int -g dt$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \qquad x(t) = \int v_x(t) dt = \int v_{0x} dt = x_0 + v_{0x} \cdot t$$

$$y'(t) = v_y(t) \qquad y(t) = \int v_y(t) dt$$

\Rightarrow

$$v'_x(t) = 0 \qquad v_x(t) = \int 0 dt = v_{0x}$$

$$v'_y(t) = -g \qquad v_y(t) = \int -g dt = v_{0y} - g \cdot t$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \qquad x(t) = \int v_x(t) dt = \int v_{0x} dt = x_0 + v_{0x} \cdot t$$

$$y'(t) = v_y(t) \qquad y(t) = \int v_y(t) dt = \int v_{0y} - g \cdot t dt$$

\Rightarrow

$$v'_x(t) = 0 \qquad v_x(t) = \int 0 dt = v_{0x}$$

$$v'_y(t) = -g \qquad v_y(t) = \int -g dt = v_{0y} - g \cdot t$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \quad x(t) = \int v_x(t) dt = \int v_{0x} dt = x_0 + v_{0x} \cdot t$$

$$y'(t) = v_y(t) \quad \Rightarrow \quad y(t) = \int v_y(t) dt = \int v_{0y} - g \cdot t dt = y_0 + v_{0y} \cdot t - \frac{1}{2}gt^2$$

$$v'_x(t) = 0 \quad v_x(t) = \int 0 dt = v_{0x}$$

$$v'_y(t) = -g \quad v_y(t) = \int -g dt = v_{0y} - g \cdot t$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \quad x += v_x * dt;$$

$$y'(t) = v_y(t) \quad y += v_y * dt;$$

\Rightarrow

$$v'_x(t) = 0 \quad v_x += 0 * dt;$$

$$v'_y(t) = -g \quad v_y += -g * dt;$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

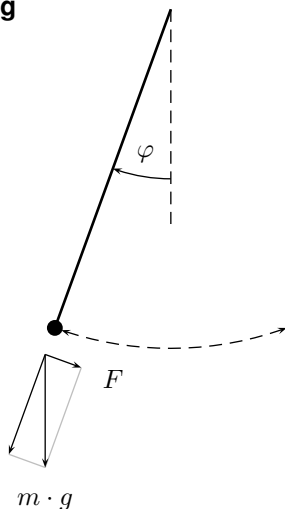
Beispiel 2: Mathematisches Pendel

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

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

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

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



5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

Beispiel 2: Mathematisches Pendel

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

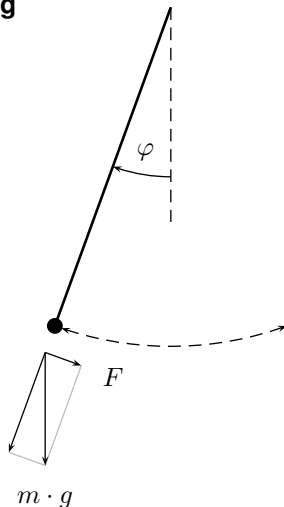
$$\omega'(t) = -\frac{g}{l} \cdot \sin \varphi(t)$$

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

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

Beispiel 3: Weltraum-Simulation

Praktikumsaufgabe



Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

19. Dezember 2024

Hardwarenahe Programmierung

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

1 Einführung

2 Einführung in C

3 Bibliotheken

4 Hardwarenahe Programmierung

...

4.4 volatile-Variable

4.5 Byte-Reihenfolge – Endianness

4.6 Binärdarstellung negativer Zahlen

4.7 Speicherausrichtung – Alignment

4.8 Binärdarstellung von Gleitkommazahlen

5 Algorithmen

5.1 Differentialgleichungen

5.2 Rekursion

5.3 Aufwandsabschätzungen

...

6 Objektorientierte Programmierung

7 Datenstrukturen

4.4 volatile-Variable

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

Beispiel: Taster

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

```
...
```

```
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;
```

```
}
```

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 -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.

4.5 Byte-Reihenfolge – Endianness

4.5.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

4.5 Byte-Reihenfolge – Endianness

4.5.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

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

4.5 Byte-Reihenfolge – Endianness

4.5.1 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“

4.5 Byte-Reihenfolge – Endianness

4.5.1 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

4.5 Byte-Reihenfolge – Endianness

4.5.1 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“

4.5 Byte-Reihenfolge – Endianness

4.5.1 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

4.5 Byte-Reihenfolge – Endianness

4.5.1 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

4.5 Byte-Reihenfolge – Endianness

4.5.1 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!**

4.5 Byte-Reihenfolge – Endianness

4.5.1 Konzept

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

Welche Bits liegen wo?

—→ Geschmackssache

... **außer bei Datenaustausch!**

- Dateiformate
- Datenübertragung

4.5 Byte-Reihenfolge – Endianness

4.5.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian

4.5 Byte-Reihenfolge – Endianness

4.5.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian
- ältere AIFF-Dateien (.aiff): Big-Endian
- neuere AIFF-Dateien (.aiff): Little-Endian

4.5 Byte-Reihenfolge – Endianness

4.5.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian
- ältere AIFF-Dateien (.aiff): Big-Endian
- neuere AIFF-Dateien (.aiff): Little-Endian

Grafik-Formate: Reihenfolge der Bits in den Bytes

- PBM-Dateien: Big-Endian
- XBM-Dateien: Little-Endian

4.5 Byte-Reihenfolge – Endianness

4.5.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian
- ältere AIFF-Dateien (.aiff): Big-Endian
- neuere AIFF-Dateien (.aiff): Little-Endian

Grafik-Formate: Reihenfolge der Bits in den Bytes

- PBM-Dateien: Big-Endian, MSB first
- XBM-Dateien: Little-Endian, LSB first

MSB/LSB = most/least significant bit

4.5 Byte-Reihenfolge – Endianness

4.5.3 Datenübertragung

- RS-232 (serielle Schnittstelle): LSB first
- I²C: MSB first
- USB: beides

4.5 Byte-Reihenfolge – Endianness

4.5.3 Datenübertragung

- RS-232 (serielle Schnittstelle): LSB first
- I²C: MSB first
- USB: beides
- Ethernet: LSB first
- TCP/IP (Internet): Big-Endian

4.6 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

4.6 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

→ $255 + 1 = 0$

4.6 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 .

4.6 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

4.6 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

→ $127 + 1 = -128$

4.6 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/>

4.6 Binärdarstellung negativer Zahlen

Frage: Für welche Zahl steht der Speicherinhalt

a3	90
----	----

 (hexadezimal)?

4.6 Binärdarstellung negativer Zahlen

Frage: Für welche Zahl steht der Speicherinhalt

a3	90
----	----

 (hexadezimal)?

Antwort: Das kommt darauf an. ;—)

4.6 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)

4.6 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	

4.7 Speicherausrichtung – Alignment

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

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

4.7 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

4.7 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

4.7 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

4.7 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;
```

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

4.7 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

4.8 Binärdarstellung von Gleitkommazahlen

Beispiel für Gleitkommazahl: $2,351 \cdot 10^5$ (oder: 2.351×10^5)

Bezeichnungen: Mantisse $\cdot 10^{\text{Exponent}}$

C-Schreibweise: **2.351e5** (oder: **2.351E5**)

4.8 Binärdarstellung von Gleitkommazahlen

Beispiel für Gleitkommazahl: $2,351 \cdot 10^5$ (oder: 2.351×10^5)

Bezeichnungen: Mantisse $\cdot 10^{\text{Exponent}}$

C-Schreibweise: **2.351e5** (oder: **2.351E5**)

Wie speichert man Gleitkommazahlen?

m -Bit-Zahl, davon

- e Bits für den Exponenten (einschließlich Vorzeichen) zur Basis 2 (statt 10),
- 1 Bit für das Vorzeichen der Mantisse,
- $m - e - 1$ Bits für die Mantisse.

4.8 Binärdarstellung von Gleitkommazahlen

Beispiel für Gleitkommazahl: $2,351 \cdot 10^5$ (oder: 2.351×10^5)

Bezeichnungen: Mantisse $\cdot 10^{\text{Exponent}}$

C-Schreibweise: **2.351e5** (oder: **2.351E5**)

Wie speichert man Gleitkommazahlen?

m -Bit-Zahl, davon

- e Bits für den Exponenten (einschließlich Vorzeichen) zur Basis 2 (statt 10),
- 1 Bit für das Vorzeichen der Mantisse,
- $m - e - 1$ Bits für die Mantisse.

Trick: Mantisse als *normalisierte Zahl* abspeichern

4.8 Binärdarstellung von Gleitkommazahlen

Beispiel für Gleitkommazahl: $2,351 \cdot 10^5$ (oder: 2.351×10^5)

Bezeichnungen: Mantisse $\cdot 10^{\text{Exponent}}$

C-Schreibweise: **2.351e5** (oder: **2.351E5**)

Wie speichert man Gleitkommazahlen?

m -Bit-Zahl, davon

- e Bits für den Exponenten (einschließlich Vorzeichen) zur Basis 2 (statt 10),
- 1 Bit für das Vorzeichen der Mantisse,
- $m - e - 1$ Bits für die Mantisse.

Trick: Mantisse als *normalisierte Zahl* abspeichern

Vorteil gegenüber ganzen Zahlen:

größerer Wertebereich bei vergleichbarem Speicherplatzbedarf

Nachteil gegenüber ganzen Zahlen: Rundungsfehler

→ **ungeeignet** für Anwendungen, bei denen es auf jedes Bit ankommt
(z. B. Verschlüsselung)

4.8 Binärdarstellung von Gleitkommazahlen

Problem beim Arbeiten mit Gleitkommazahlen: Auslöschung von Ziffern

- Zahlen aufsummieren:
vorher sortieren, mit der kleinsten Zahl beginnen

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t)$$

$$y'(t) = v_y(t)$$

$$v'_x(t) = 0$$

$$v'_y(t) = -g$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \qquad x(t) = \int v_x(t) dt$$

$$y'(t) = v_y(t) \qquad y(t) = \int v_y(t) dt$$

\Rightarrow

$$v'_x(t) = 0 \qquad v_x(t) = \int 0 dt$$

$$v'_y(t) = -g \qquad v_y(t) = \int -g dt$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \qquad x(t) = \int v_x(t) dt$$

$$y'(t) = v_y(t) \qquad y(t) = \int v_y(t) dt$$

\Rightarrow

$$v'_x(t) = 0 \qquad v_x(t) = \int 0 dt = v_{0x}$$

$$v'_y(t) = -g \qquad v_y(t) = \int -g dt$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \qquad x(t) = \int v_x(t) dt = \int v_{0x} dt$$

$$y'(t) = v_y(t) \qquad y(t) = \int v_y(t) dt$$

\Rightarrow

$$v'_x(t) = 0 \qquad v_x(t) = \int 0 dt = v_{0x}$$

$$v'_y(t) = -g \qquad v_y(t) = \int -g dt$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \qquad x(t) = \int v_x(t) dt = \int v_{0x} dt = x_0 + v_{0x} \cdot t$$

$$y'(t) = v_y(t) \qquad y(t) = \int v_y(t) dt$$

\Rightarrow

$$v'_x(t) = 0 \qquad v_x(t) = \int 0 dt = v_{0x}$$

$$v'_y(t) = -g \qquad v_y(t) = \int -g dt$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \qquad x(t) = \int v_x(t) dt = \int v_{0x} dt = x_0 + v_{0x} \cdot t$$

$$y'(t) = v_y(t) \qquad y(t) = \int v_y(t) dt$$

\Rightarrow

$$v'_x(t) = 0 \qquad v_x(t) = \int 0 dt = v_{0x}$$

$$v'_y(t) = -g \qquad v_y(t) = \int -g dt = v_{0y} - g \cdot t$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \qquad x(t) = \int v_x(t) dt = \int v_{0x} dt = x_0 + v_{0x} \cdot t$$

$$y'(t) = v_y(t) \qquad y(t) = \int v_y(t) dt = \int v_{0y} - g \cdot t dt$$

\Rightarrow

$$v'_x(t) = 0 \qquad v_x(t) = \int 0 dt = v_{0x}$$

$$v'_y(t) = -g \qquad v_y(t) = \int -g dt = v_{0y} - g \cdot t$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \qquad x(t) = \int v_x(t) dt = \int v_{0x} dt = x_0 + v_{0x} \cdot t$$

$$y'(t) = v_y(t) \qquad \Rightarrow \qquad y(t) = \int v_y(t) dt = \int v_{0y} - g \cdot t dt = y_0 + v_{0y} \cdot t - \frac{1}{2}gt^2$$

$$v'_x(t) = 0 \qquad v_x(t) = \int 0 dt = v_{0x}$$

$$v'_y(t) = -g \qquad v_y(t) = \int -g dt = v_{0y} - g \cdot t$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \quad x += v_x * dt;$$

$$y'(t) = v_y(t) \quad y += v_y * dt;$$

\Rightarrow

$$v'_x(t) = 0 \quad v_x += 0 * dt;$$

$$v'_y(t) = -g \quad v_y += -g * dt;$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

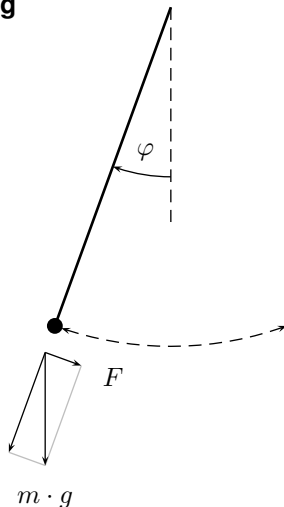
Beispiel 2: Mathematisches Pendel

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

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

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

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



5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

Beispiel 2: Mathematisches Pendel

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

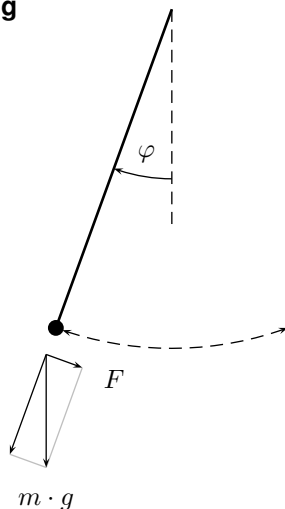
$$\omega'(t) = -\frac{g}{l} \cdot \sin \varphi(t)$$

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

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

Beispiel 3: Weltraum-Simulation

Praktikumsaufgabe



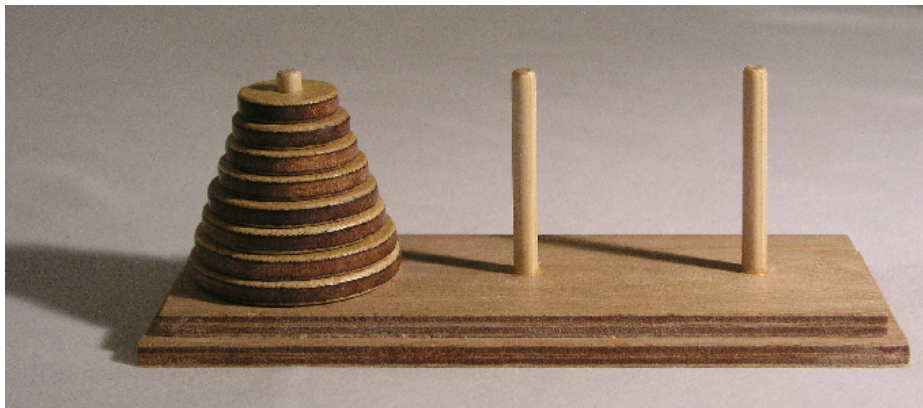
5.2 Rekursion

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

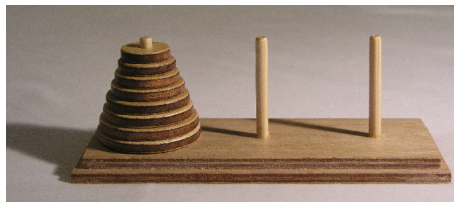


5.2 Rekursion

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.

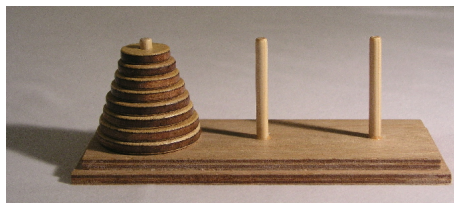


5.2 Rekursion

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

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

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

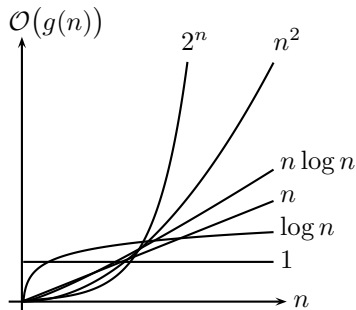
Wann ist ein Programm „schnell“?

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



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Türme von Hanoi: $\mathcal{O}(2^n)$

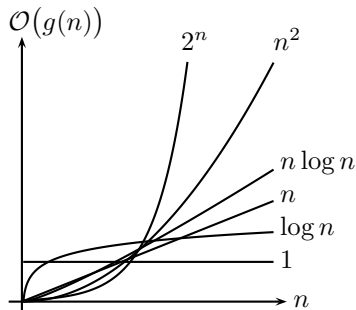
Für jede zusätzliche Scheibe
verdoppelt sich die Rechenzeit!

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

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“?

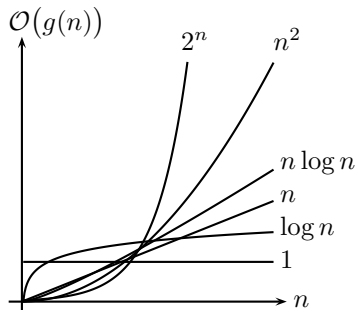
Faustregel:

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

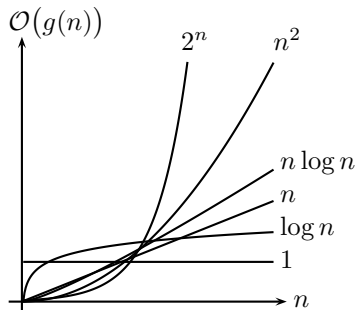
Faustregel:

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen mit Schummeln



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

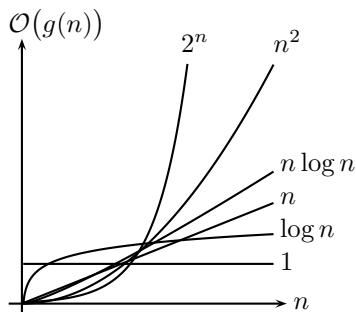
Faustregel:

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen mit Schummeln: $\mathcal{O}(1)$



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

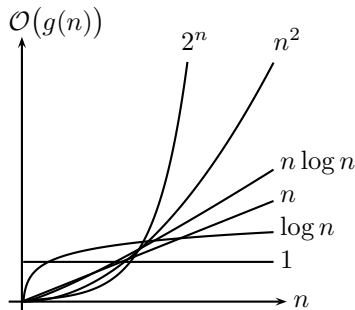
Faustregel:

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



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

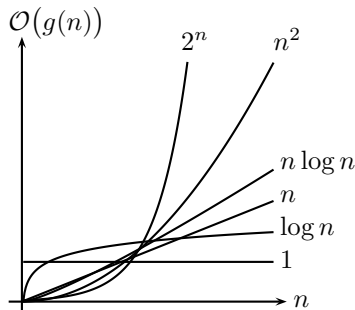
Faustregel:

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: $\mathcal{O}(n)$



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

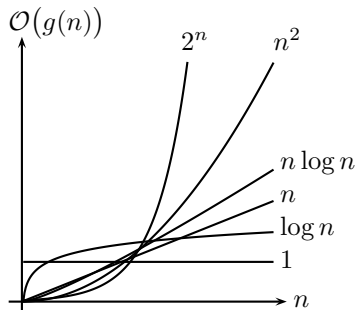
Faustregel:

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



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

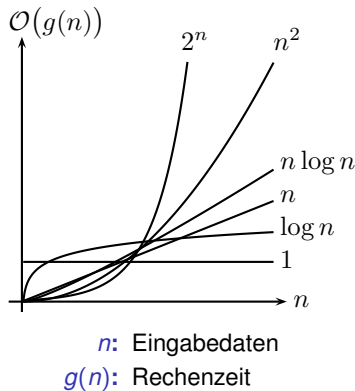
Faustregel:

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



5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

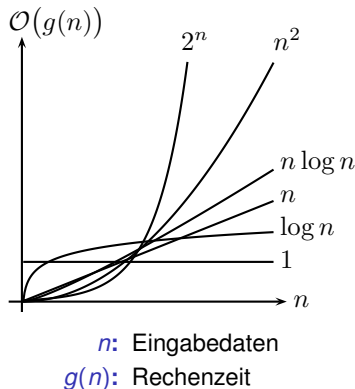
Faustregel:

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



5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

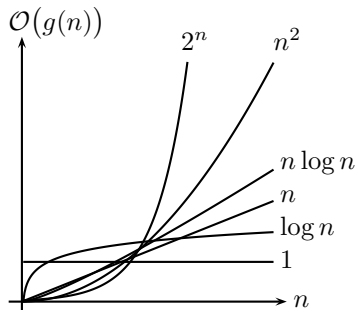
Faustregel:

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



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

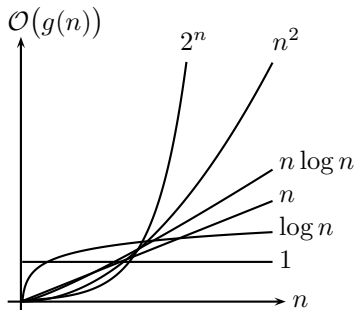
Faustregel:

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



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

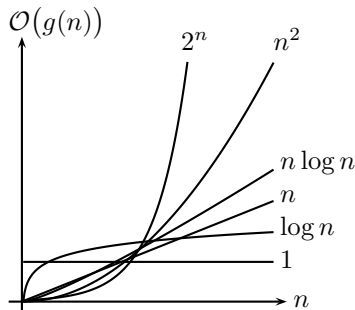
Schachtelung der Schleifen zählen

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“)

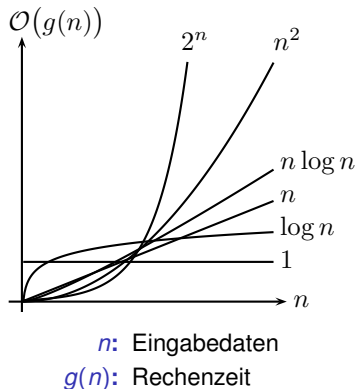
```
int c = 1;
for (int i = 0; i < e; i++)
    c = (c * m) % N;
```

- $\mathcal{O}(e)$ 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:

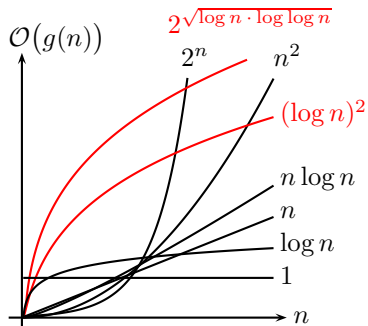
Schachtelung der Schleifen zählen

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

Wie schnell ist RSA?

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

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



n : Eingabedaten

$g(n)$: Rechenzeit

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

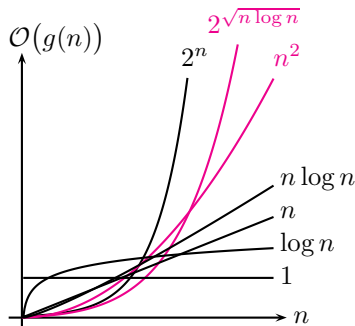
$$O((\log n)^2)$$

$$O(n^2)$$

- 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).

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

9. Januar 2025

Hardwarenahe Programmierung

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

1 Einführung

2 Einführung in C

3 Bibliotheken

4 Hardwarenahe Programmierung

...

4.8 Binärdarstellung von Gleitkommazahlen

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

...

7 Datenstrukturen

4.8 Binärdarstellung von Gleitkommazahlen

Beispiel für Gleitkommazahl: $2,351 \cdot 10^5$ (oder: 2.351×10^5)

Bezeichnungen: Mantisse $\cdot 10^{\text{Exponent}}$

C-Schreibweise: **2.351e5** (oder: **2.351E5**)

4.8 Binärdarstellung von Gleitkommazahlen

Beispiel für Gleitkommazahl: $2,351 \cdot 10^5$ (oder: 2.351×10^5)

Bezeichnungen: Mantisse $\cdot 10^{\text{Exponent}}$

C-Schreibweise: **2.351e5** (oder: **2.351E5**)

Wie speichert man Gleitkommazahlen?

m -Bit-Zahl, davon

- e Bits für den Exponenten (einschließlich Vorzeichen) zur Basis 2 (statt 10),
- 1 Bit für das Vorzeichen der Mantisse,
- $m - e - 1$ Bits für die Mantisse.

4.8 Binärdarstellung von Gleitkommazahlen

Beispiel für Gleitkommazahl: $2,351 \cdot 10^5$ (oder: 2.351×10^5)

Bezeichnungen: Mantisse $\cdot 10^{\text{Exponent}}$

C-Schreibweise: **2.351e5** (oder: **2.351E5**)

Wie speichert man Gleitkommazahlen?

m -Bit-Zahl, davon

- e Bits für den Exponenten (einschließlich Vorzeichen) zur Basis 2 (statt 10),
- 1 Bit für das Vorzeichen der Mantisse,
- $m - e - 1$ Bits für die Mantisse.

Trick: Mantisse als *normalisierte Zahl* abspeichern

4.8 Binärdarstellung von Gleitkommazahlen

Beispiel für Gleitkommazahl: $2,351 \cdot 10^5$ (oder: 2.351×10^5)

Bezeichnungen: Mantisse $\cdot 10^{\text{Exponent}}$

C-Schreibweise: **2.351e5** (oder: **2.351E5**)

Wie speichert man Gleitkommazahlen?

m -Bit-Zahl, davon

- e Bits für den Exponenten (einschließlich Vorzeichen) zur Basis 2 (statt 10),
- 1 Bit für das Vorzeichen der Mantisse,
- $m - e - 1$ Bits für die Mantisse.

Trick: Mantisse als *normalisierte Zahl* abspeichern

Vorteil gegenüber ganzen Zahlen:

größerer Wertebereich bei vergleichbarem Speicherplatzbedarf

Nachteil gegenüber ganzen Zahlen: Rundungsfehler

→ **ungeeignet** für Anwendungen, bei denen es auf jedes Bit ankommt
(z. B. Verschlüsselung)

4.8 Binärdarstellung von Gleitkommazahlen

Problem beim Arbeiten mit Gleitkommazahlen: Auslöschung von Ziffern

- Zahlen aufsummieren:
vorher sortieren, mit der kleinsten Zahl beginnen

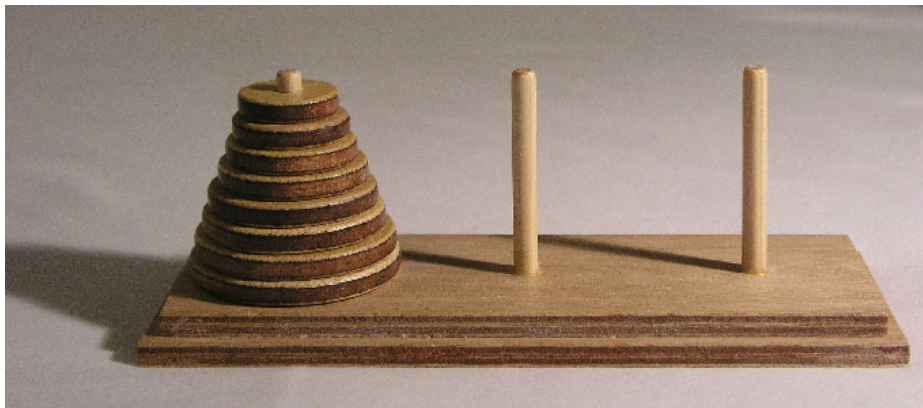
5.2 Rekursion

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

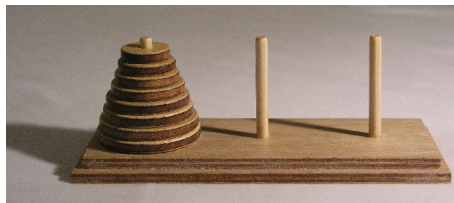


5.2 Rekursion

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.

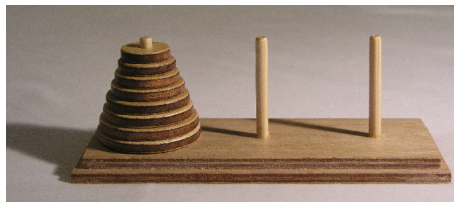


5.2 Rekursion

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

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

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

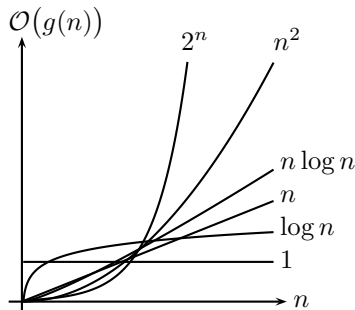
Wann ist ein Programm „schnell“?

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



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Türme von Hanoi: $\mathcal{O}(2^n)$

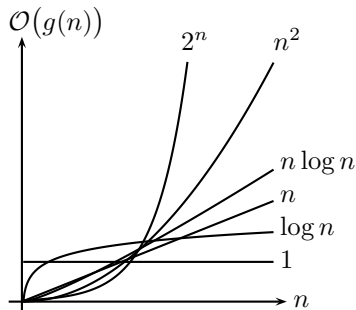
Für jede zusätzliche Scheibe
verdoppelt sich die Rechenzeit!

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

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“?

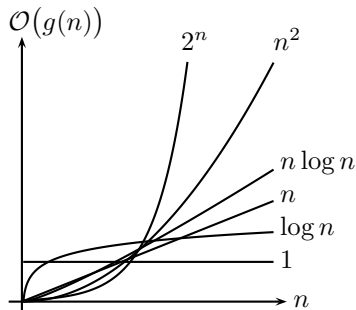
Faustregel:

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

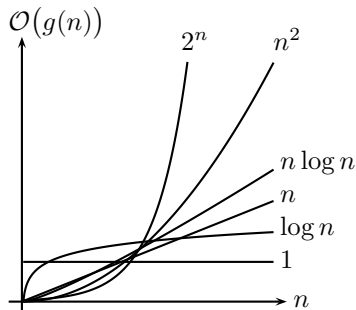
Faustregel:

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen mit Schummeln



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

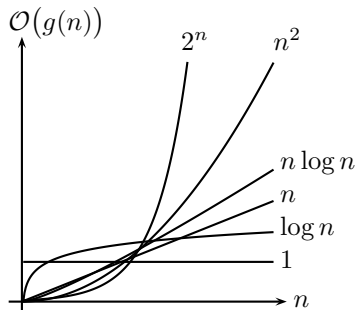
Faustregel:

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen mit Schummeln: $\mathcal{O}(1)$



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

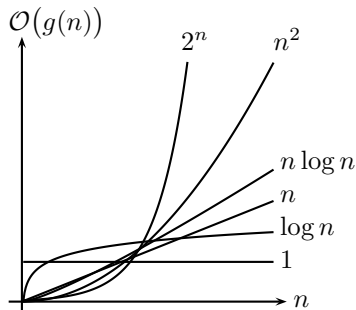
Faustregel:

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



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

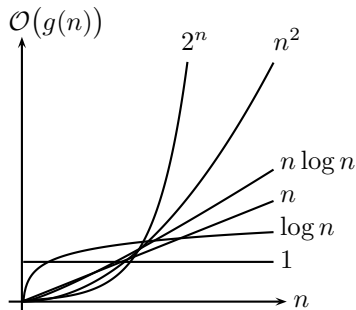
Faustregel:

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: $\mathcal{O}(n)$



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

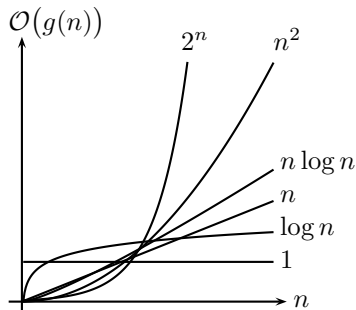
Faustregel:

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen mit Schummeln: $O(1)$
- Maximum suchen: $O(n)$
- Selection-Sort



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

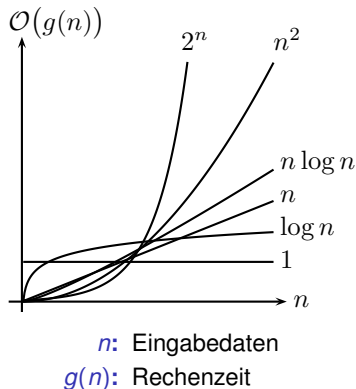
Faustregel:

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



5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

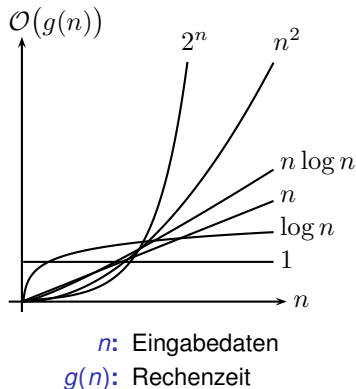
Faustregel:

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



5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

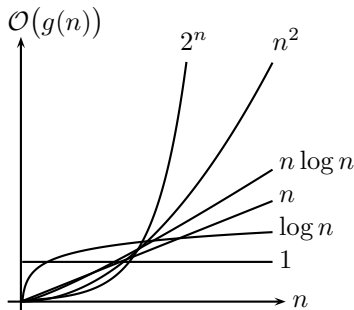
Faustregel:

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



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

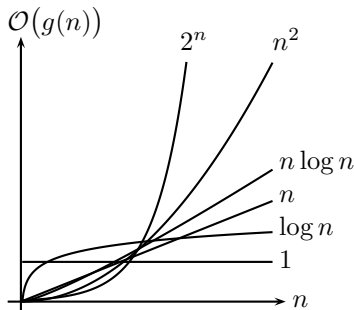
Faustregel:

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



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

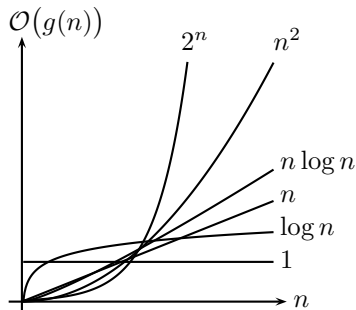
Schachtelung der Schleifen zählen

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“)

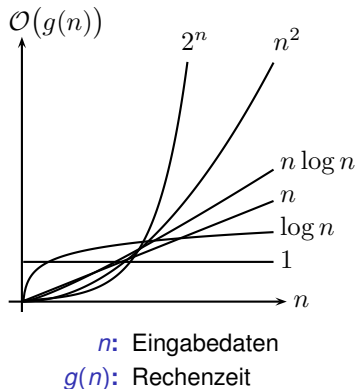
```
int c = 1;
for (int i = 0; i < e; i++)
    c = (c * m) % N;
```

- $\mathcal{O}(e)$ 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:

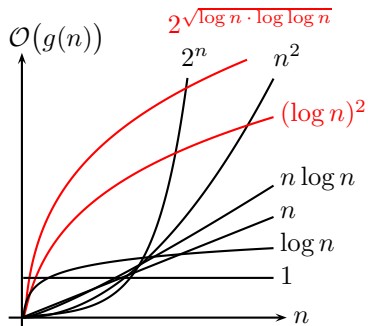
Schachtelung der Schleifen zählen

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

Wie schnell ist RSA?

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

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



n : Eingabedaten

$g(n)$: Rechenzeit

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

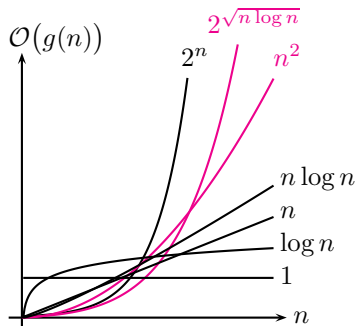
$$O((\log n)^2)$$

$$O(n^2)$$

- 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)

→ nächste Woche

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);
```


Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

16. Januar 2025

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

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

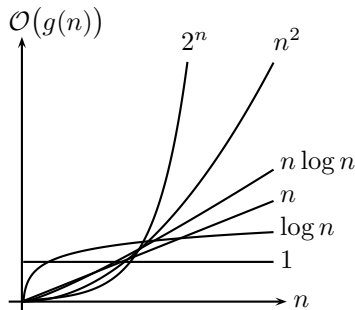
Wann ist ein Programm „schnell“?

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



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Türme von Hanoi: $\mathcal{O}(2^n)$

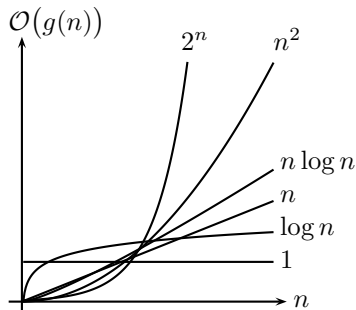
Für jede zusätzliche Scheibe
verdoppelt sich die Rechenzeit!

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

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“?

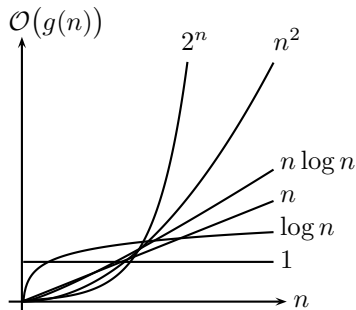
Faustregel:

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

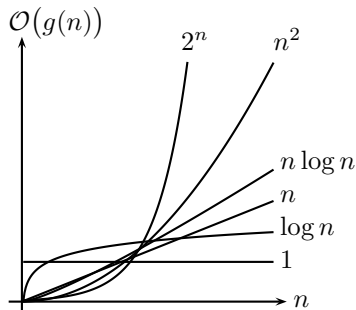
Faustregel:

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen mit Schummeln



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

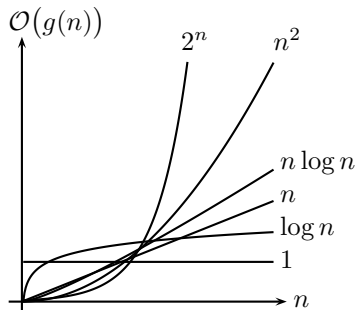
Faustregel:

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen mit Schummeln: $\mathcal{O}(1)$



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

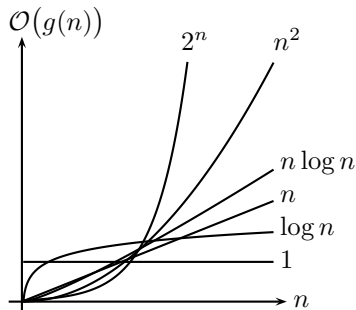
Faustregel:

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



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

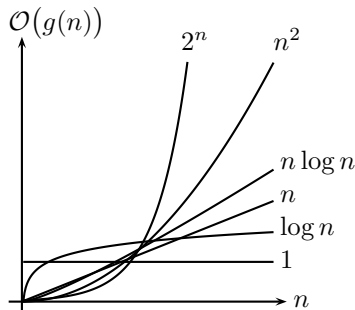
Faustregel:

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: $\mathcal{O}(n)$



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

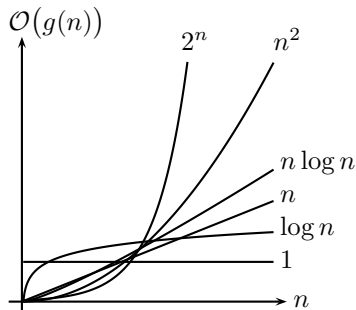
Faustregel:

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



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

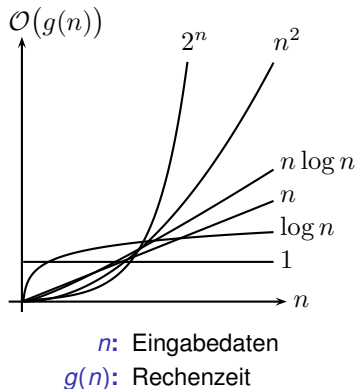
Faustregel:

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



5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

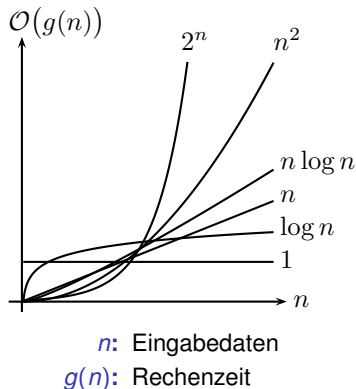
Faustregel:

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



5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

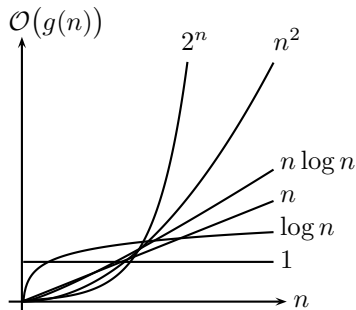
Faustregel:

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



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

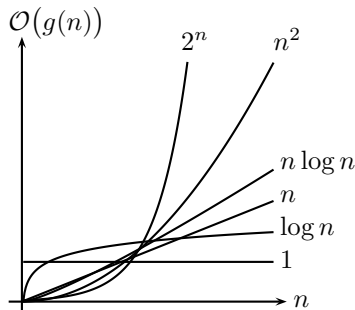
Faustregel:

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



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

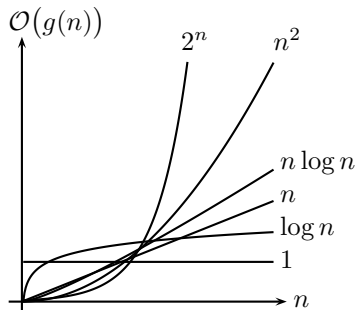
Faustregel:

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“)

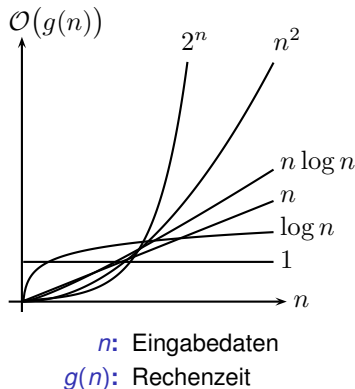
```
int c = 1;
for (int i = 0; i < e; i++)
    c = (c * m) % N;
```

- $\mathcal{O}(e)$ 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:

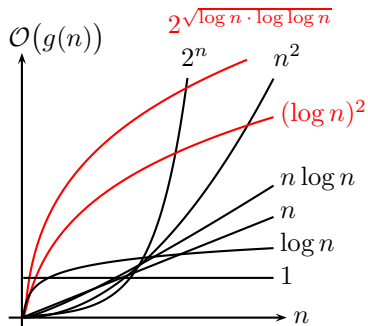
Schachtelung der Schleifen zählen

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):

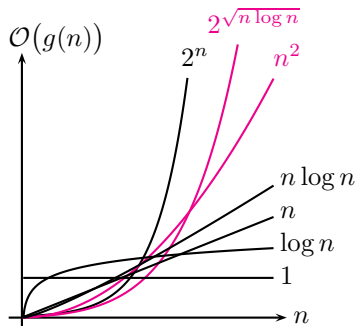
$$O((\log n)^2)$$

$$O(n^2)$$

- 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

6.0 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

```
void print_integer (t_object *this)
```

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



Zeiger auf Funktionen

```
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;
```

6.5 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**

6.6 Ausblick: C++

```
typedef struct
{
    void (* print) (union t_object *this);
} t_base;
```

```
typedef struct
{
    void (* print) (...);
    int content;
} t_integer;
```

```
typedef struct
{
    void (* print) (union t_object *this);
    char *content;
} t_string;
```

6.6 Ausblick: C++

```
struct TBase  
{  
    virtual void print (void);  
};
```

```
struct TInteger: public TBase  
{  
    virtual void print (void);  
    int content;  
};
```

```
struct TString: public TBase  
{  
    virtual void print (void);  
    char *content;  
};
```

7 Datenstrukturen

7.1 Stack und FIFO

Im letzten Praktikumsversuch:

- Array nur zum Teil benutzt
- Variable speichert genutzte Länge
- Elemente hinten anfügen oder entfernen

→ Stack

- hinten anfügen/entfernen: $\mathcal{O}(1)$
- vorne oder in der Mitte einfügen/entfernen: $\mathcal{O}(n)$

Auch möglich:

- Array nur zum Teil benutzt
- 2 Variable speichern genutzte Länge (ringförmig)
- Elemente hinten anfügen oder vorne entfernen

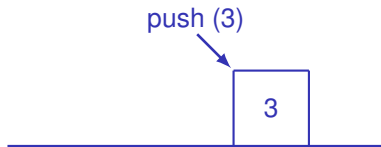
→ FIFO

- vorne oder hinten anfügen oder entfernen: $\mathcal{O}(1)$
- in der Mitte einfügen/entfernen: $\mathcal{O}(n)$

7 Datenstrukturen

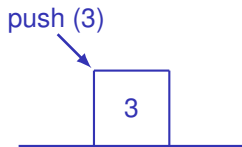
7.1 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

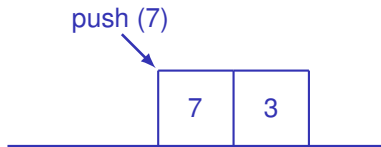


LIFO = Stack = Stapel

7 Datenstrukturen

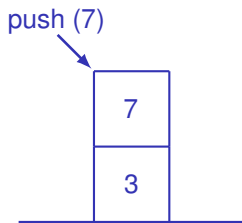
7.1 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



LIFO = Stack = Stapel

7 Datenstrukturen

7.1 Stack und FIFO

„First In – First Out“

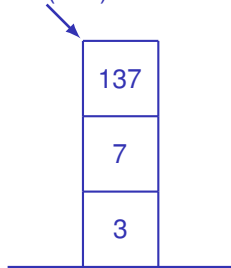
push (137)



FIFO = Queue = Reihe

„Last In – First Out“

push (137)



LIFO = Stack = Stapel

7 Datenstrukturen

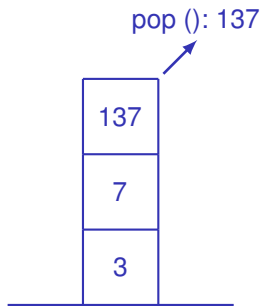
7.1 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

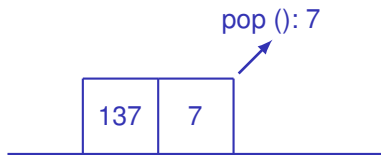


LIFO = Stack = Stapel

7 Datenstrukturen

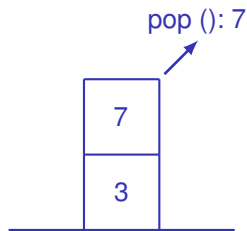
7.1 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

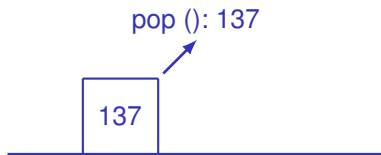


LIFO = Stack = Stapel

7 Datenstrukturen

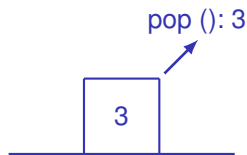
7.1 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

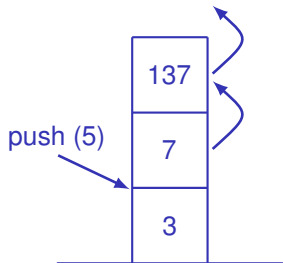


LIFO = Stack = Stapel

7 Datenstrukturen

7.1 Stack und FIFO

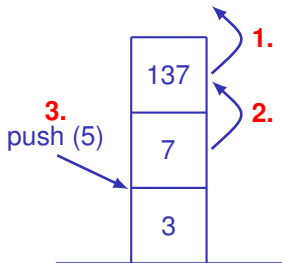
Array (Stack, FIFO):
in der Mitte einfügen



7 Datenstrukturen

7.1 Stack und FIFO

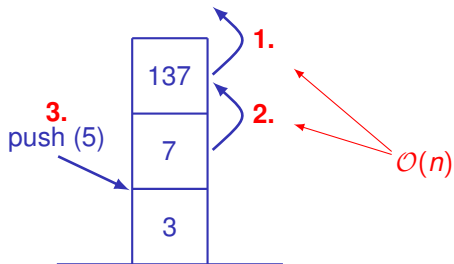
Array (Stack, FIFO):
in der Mitte einfügen



7 Datenstrukturen

7.1 Stack und FIFO

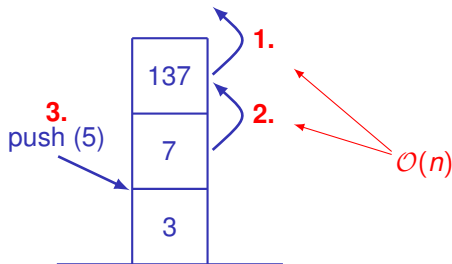
Array (Stack, FIFO):
in der Mitte einfügen



7 Datenstrukturen

7.1 Stack und FIFO

Array (Stack, FIFO):
in der Mitte einfügen

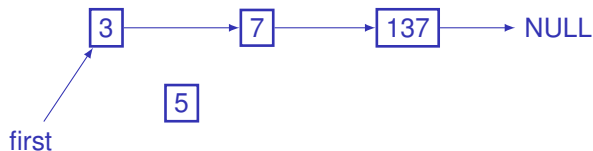


In Array (Stack, FIFO) ...

- einfügen: $O(n)$
- suchen: $O(n)$
- geschickt suchen: $O(\log n)$
- beim Einfügen sortieren:
 ~~$O(n \log n)$~~ $O(n^2)$

7 Datenstrukturen

7.2 Verkettete Listen

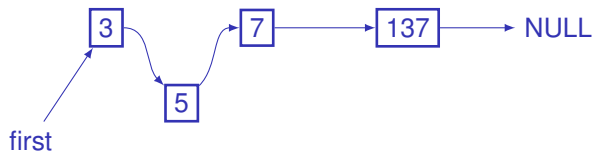


- Jeder Datensatz enthält einen Zeiger auf das nächste Element.
- Beim letzten Element zeigt der Zeiger auf **NULL**.
- Eine Variable zeigt auf das erste Element.
- Wenn die Liste leer ist, zeigt die Variable auf **NULL**.

→ (einfach) **verkettete Liste**

7 Datenstrukturen

7.2 Verkettete Listen



- Jeder Datensatz enthält einen Zeiger auf das nächste Element.
- Beim letzten Element zeigt der Zeiger auf **NULL**.
- Eine Variable zeigt auf das erste Element.
- Wenn die Liste leer ist, zeigt die Variable auf **NULL**.

→ (einfach) **verkettete Liste**

7 Datenstrukturen

7.2 Verkettete Listen

In Array (Stack, FIFO) ...

- in der Mitte einfügen: $\mathcal{O}(n)$
- wahlfreier Zugriff: $\mathcal{O}(1)$
- suchen: $\mathcal{O}(n)$
- geschickt suchen: $\mathcal{O}(\log n)$
- beim Einfügen sortieren:
 ~~$\mathcal{O}(n \log n)$~~ $\mathcal{O}(n^2)$

In (einfach) verkettete/r Liste ...

- in der Mitte einfügen: $\mathcal{O}(1)$
- wahlfreier Zugriff: $\mathcal{O}(n)$
- suchen: $\mathcal{O}(n)$
- ~~geschickt~~ suchen: $\mathcal{O}(n)$
- beim Einfügen sortieren:
 ~~$\mathcal{O}(n \log n)$~~ $\mathcal{O}(n^2)$

7 Datenstrukturen

7.2 Verkettete Listen

In Array (Stack, FIFO) ...

- in der Mitte einfügen: $\mathcal{O}(n)$
- wahlfreier Zugriff: $\mathcal{O}(1)$
- suchen: $\mathcal{O}(n)$
- geschickt suchen: $\mathcal{O}(\log n)$
- beim Einfügen sortieren:
 ~~$\mathcal{O}(n \log n)$~~ $\mathcal{O}(n^2)$

In (einfach) verkettete/r Liste ...

- in der Mitte einfügen: $\mathcal{O}(1)$
- wahlfreier Zugriff: $\mathcal{O}(n)$
- suchen: $\mathcal{O}(n)$
- ~~geschickt~~ suchen: $\mathcal{O}(n)$
- beim Einfügen sortieren:
 ~~$\mathcal{O}(n \log n)$~~ $\mathcal{O}(n^2)$

In (ausbalancierten) Bäumen ...

- in der Mitte einfügen: $\mathcal{O}(\log n)$
- wahlfreier Zugriff: $\mathcal{O}(\log n)$
- suchen: $\mathcal{O}(\log n)$
- beim Einfügen sortieren:
 $\mathcal{O}(n \log n)$

7 Datenstrukturen

7.3 Bäume

```
typedef struct node
```

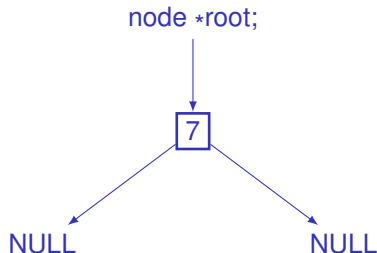
```
{  
    int content;  
    struct node *left, *right;  
} node;
```

```
node *root;
```

7 Datenstrukturen

7.3 Bäume

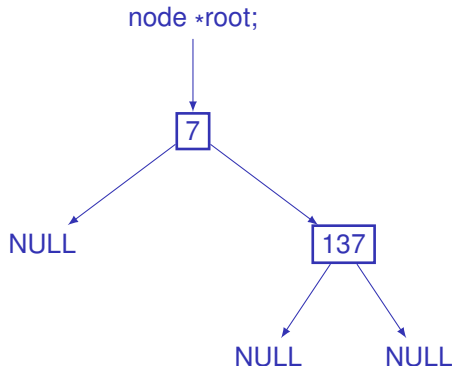
```
typedef struct node
{
    int content;
    struct node *left, *right;
} node;
```



7 Datenstrukturen

7.3 Bäume

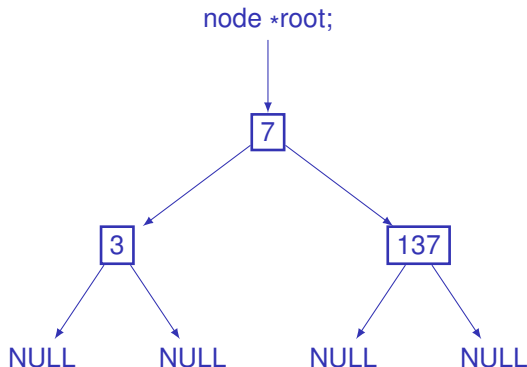
```
typedef struct node
{
    int content;
    struct node *left, *right;
} node;
```



7 Datenstrukturen

7.3 Bäume

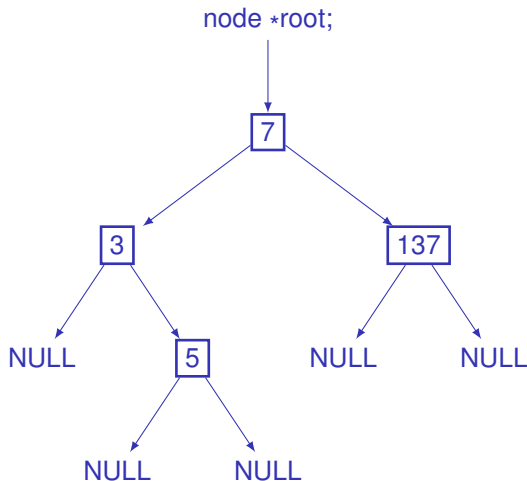
```
typedef struct node
{
    int content;
    struct node *left, *right;
} node;
```



7 Datenstrukturen

7.3 Bäume

```
typedef struct node
{
    int content;
    struct node *left, *right;
} node;
```



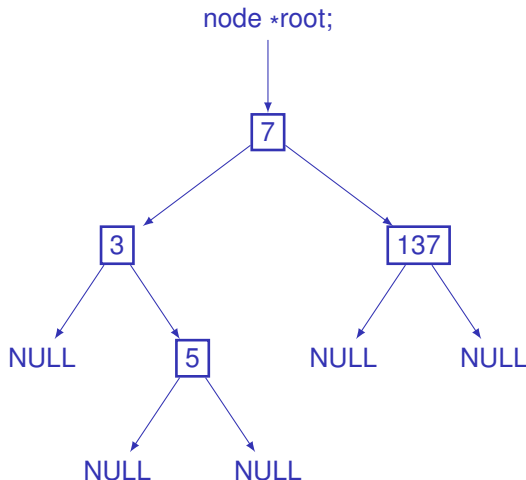
7 Datenstrukturen

7.3 Bäume

```
typedef struct node
```

```
{  
    int content;  
    struct node *left, *right;  
} node;
```

- Einfügen: rekursiv, $\mathcal{O}(\log n)$
- Suchen: rekursiv, $\mathcal{O}(\log n)$
- beim Einfügen sortieren:
rekursiv, $\mathcal{O}(n \log n)$



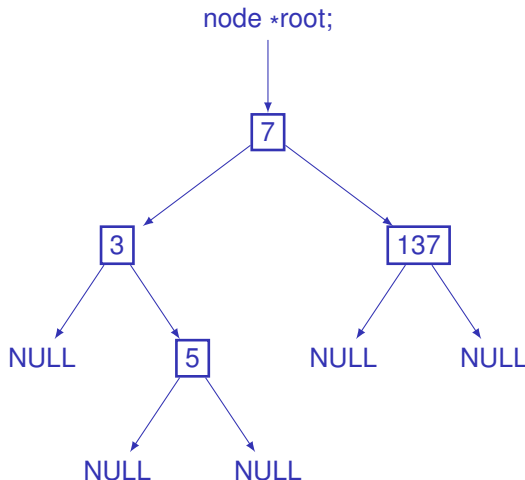
7 Datenstrukturen

7.3 Bäume

```
typedef struct node
```

```
{  
    int content;  
    struct node *left, *right;  
} node;
```

- Einfügen: rekursiv, $\mathcal{O}(\log n)$
- Suchen: rekursiv, $\mathcal{O}(\log n)$
- beim Einfügen sortieren:
rekursiv, $\mathcal{O}(n \log n)$
- **Worst Case: $\mathcal{O}(n^2)$**
vorher bereits sortiert
→ balancierte Bäume
Anwendung: Datenbanken



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

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
 - 7.1 Stack und FIFO
 - 7.2 Verkettete Listen
 - 7.3 Bäume

***Vielen Dank für Ihre Aufmerksamkeit
und viel Erfolg bei den Prüfungen!***