

# Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

Wintersemester 2018/19

## 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 8.10.2018 unter <https://gitlab.cvh-server.de/pgerwinski/hp/tree/master/20181008/>).

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

8. Oktober 2018

# Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

*rerum naturalium* = der natürlichen Dinge (lat.)

8. Oktober 2018



# Hardwarenahe Programmierung

Man kann Computer vollständig beherrschen.

## Rechnertechnik

Man kann vollständig verstehen, wie Computer funktionieren.

# Hardwarenahe Programmierung

Man kann Computer vollständig beherrschen.

## Programmierung in C

# Hardwarenahe Programmierung

Man kann Computer vollständig beherrschen.

## Programmierung in C

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen



# Hardwarenahe Programmierung

Man kann Computer vollständig beherrschen.

## Programmierung in C

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen

Hardware und/oder Betriebssystem

# Hardwarenahe Programmierung

Man kann Computer vollständig beherrschen.

## Programmierung in C

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen



Hardware und/oder Betriebssystem

- Programmierkenntnisse werden nicht vorausgesetzt,  
aber schnelles Tempo

# Hardwarenahe Programmierung

Man kann Computer vollständig beherrschen.

## Programmierung in C

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen



Hardware und/oder Betriebssystem

- Programmierkenntnisse werden nicht vorausgesetzt, aber schnelles Tempo
- Hardware direkt ansprechen und effizient einsetzen

# Hardwarenahe Programmierung

Man kann Computer vollständig beherrschen.

## Programmierung in C und C++

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen



Hardware und/oder Betriebssystem

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

# 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
- leistungsfähig, aber gefährlich

# Hardwarenahe Programmierung

## Was ist C?

Etabliertes Profi-Werkzeug

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

„High-Level-Assembler“

# Hardwarenahe Programmierung

## Was ist C?

Etabliertes Profi-Werkzeug

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

„High-Level-Assembler“

- kein „Fallschirm“



# Hardwarenahe Programmierung

## Was ist C?

Etabliertes Profi-Werkzeug

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

„High-Level-Assembler“

- kein „Fallschirm“
- kompakte Schreibweise

# Hardwarenahe Programmierung

## Was ist C?

Etabliertes Profi-Werkzeug

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

„High-Level-Assembler“

- 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“

- 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“

- 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“

- kein „Fallschirm“
- kompakte Schreibweise

### Unix-Hintergrund

- Baukastenprinzip
- konsequente Regeln
- kein „Fallschirm“

# Zu dieser Lehrveranstaltung



- **Lehrmaterialien:**

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

- **Klausur:**

Zeit: 150 Minuten

Zulässige Hilfsmittel:

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

- **Übungen**

finden bereits diese Woche statt.

- **Praktikumstermine:**

- Versuch 1: 10. 10., 17. 10. und 24. 10. 2018
- Versuch 2 bis 4: Termine werden noch bekanntgegeben.

# Hardwarenahe Programmierung

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

## 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;  
}
```



## 2.2 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.c`) muß sich im aktuellen Verzeichnis befinden.
- aktuelles Verzeichnis ausgeben: `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](#)
- 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 Online-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*),  
≠ 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;  
printf ("%d\n", i++);  
goto loop;  
endloop:
```

(siehe z. B.:  
<http://xkcd.com/292/>)

```
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.git>

## 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

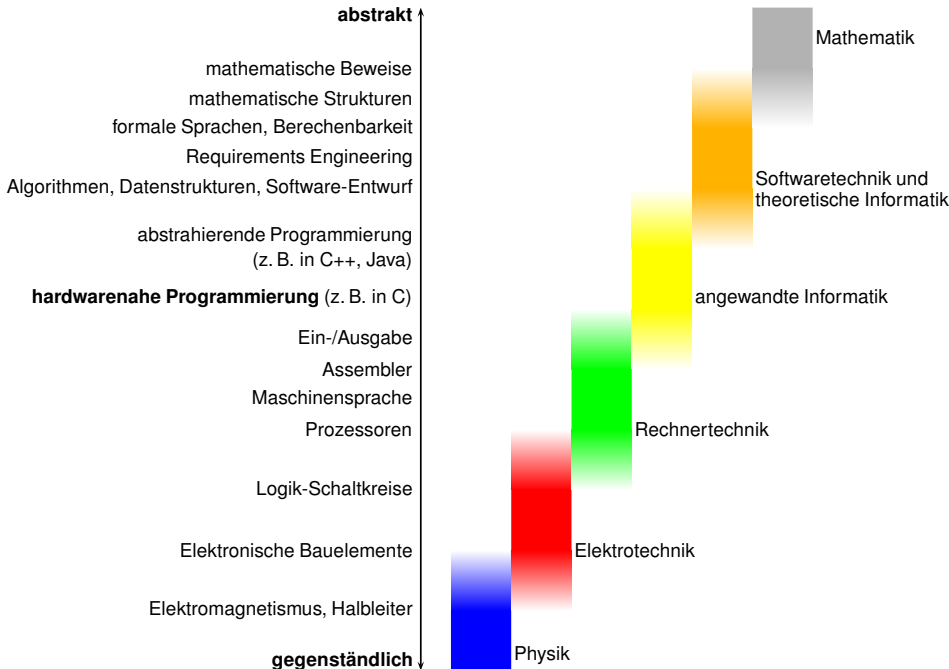
...



# Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

15. Oktober 2018



# Hardwarenahe Programmierung

Man kann Computer vollständig beherrschen.

## Rechnertechnik

Man kann vollständig verstehen, wie Computer funktionieren.

# Hardwarenahe Programmierung

Man kann Computer vollständig beherrschen.

## Programmierung in C und C++

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen



Hardware und/oder Betriebssystem

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

# Hardwarenahe Programmierung

## Was ist C?

### Etabliertes Profi-Werkzeug

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

### „High-Level-Assembler“

- kein „Fallschirm“
- kompakte Schreibweise

### Unix-Hintergrund

- Baukastenprinzip
- konsequente Regeln
- kein „Fallschirm“

# Zu dieser Lehrveranstaltung



- **Lehrmaterialien:**

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

- **Klausur:**

Zeit: 150 Minuten

Zulässige Hilfsmittel:

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

- **Übungen**

finden bereits diese Woche statt.

- **Praktikumstermine:**

- Versuch 1: 10. 10., 17. 10. und 24. 10. 2018
- Versuch 2 bis 4: Termine werden noch bekanntgegeben.

# Hardwarenahe Programmierung

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

## 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

...

## 3 Bibliotheken

...

## 2 Einführung in C

### 2.1 Hello, world!

Text ausgeben

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

```
int main (void)
```

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



## 2.2 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.c`) muß sich im aktuellen Verzeichnis befinden.
- aktuelles Verzeichnis ausgeben: `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 `Win-builds`
- außerdem: Texteditor  
`vi(m)`, `nano`, `Emacs`, `Notepad++`, ...  
(Microsoft Notepad ist *nicht* geeignet!)

## 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 Online-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.git>

## 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 Seiteneffekte
- 2.8 Strukturierte Programmierung
- 2.9 Funktionen
- 2.10 Zeiger

...

## 3 Bibliotheken

...

## 2.7 Seiteneffekte

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

```
int main (void)  
{  
    printf ("%d\n", 42);  
    "\n";  
    return 0;  
}
```

## 2.7 Seiteneffekte

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

```
int main (void)
```

```
{
```

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

```
    "\n";
```

← Ausdruck als Anweisung: Wert wird ignoriert

```
    return 0;
```

```
}
```

## 2.7 Seiteneffekte

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

```
int main (void)
```

```
{
```

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

```
    "\n";
```

```
    return 0;
```

```
}
```

← Ausdruck als Anweisung: Wert wird ignoriert

## 2.7 Seiteneffekte

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

```
int main (void)
```

```
{  
    int a = printf ("%d\n", 42);  
    printf ("%d\n", a);  
    return 0;  
}
```



## 2.7 Seiteneffekte

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

```
int main (void)
{
    int a = printf ("%d\n", 42);
    printf ("%d\n", a);
    return 0;
}
```

```
$ gcc -Wall -O side-effects-1.c -o side-effects-1
$ ./side-effects-1
42
3
$
```

## 2.7 Seiteneffekte

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

```
int main (void)
```

```
{
```

```
    int a = printf ("%d\n", 42);
```

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

```
    return 0;
```

```
}
```

- `printf()` ist eine Funktion.

## 2.7 Seiteneffekte

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

```
int main (void)
```

```
{  
    int a = printf ("%d\n", 42);  
    printf ("%d\n", a);  
    return 0;  
}
```

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

## 2.7 Seiteneffekte

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

```
int main (void)
```

```
{  
    int a = printf ("%d\n", 42);  
    printf ("%d\n", a);  
    return 0;  
}
```

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

## 2.7 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: `,`

## 2.7 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: `,`

rot = mit Seiteneffekt

## 2.7 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: `,`

rot = mit Seiteneffekt

## 2.7 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: `,`

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);
```



## 2.7 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: `,`

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++);
```

## 2.7 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: `,`

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++));
```

```
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++));
```

## 2.8 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++));
```

## 2.8 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;  
printf ("%d\n", i++);  
goto loop;  
endloop:
```

(siehe z. B.:  
<http://xkcd.com/292/>)

```
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++));
```

## 2.9 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;  
}
```

# Hardwarenahe Programmierung

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

## 1 Einführung

## 2 Einführung in C

...

2.5 Verzweigungen

2.6 Schleifen

2.7 Seiteneffekte

2.8 Strukturierte Programmierung

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

22. Oktober 2018

# Hardwarenahe Programmierung

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

## 1 Einführung

## 2 Einführung in C

...

2.5 Verzweigungen

2.6 Schleifen

2.7 Seiteneffekte

2.8 Strukturierte Programmierung

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

...



## 2.7 Seiteneffekte

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

```
int main (void)
```

```
{
```

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

```
    "\n";
```

← Ausdruck als Anweisung: Wert wird ignoriert

```
    return 0;
```

```
}
```

## 2.7 Seiteneffekte

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

```
int main (void)
```

```
{
```

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

```
    "\n";
```

```
    return 0;
```

```
}
```

← Ausdruck als Anweisung: Wert wird ignoriert

## 2.7 Seiteneffekte

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

```
int main (void)
```

```
{  
    int a = printf ("%d\n", 42);  
    printf ("%d\n", a);  
    return 0;  
}
```

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

## 2.7 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: `,`

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++));
```

## 2.8 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;  
printf ("%d\n", i++);  
goto loop;  
endloop:
```

(siehe z. B.:  
<http://xkcd.com/292/>)

```
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 Strukturierte Programmierung

Beispiel:

Gesucht ist eine Zahl  $a$  mit der Eigenschaft  $a \cdot 1117 \bmod 65535 = 137$   
sowie ihr kleinster Primfaktor  $t$ .

(Aufgaben dieses Typs treten im Kontext von Verschlüsselung auf.)

```
for (int a = 0; a < 1000000; a++)
{
    if ((a * 1117) % 65535 == 137)
    {
        for (int t = 2; t < 1000000; t++)
        {
            if (a % t == 0)
            {
                printf("a=_%d\nt=_%d\n", a, t);
                a = 2000000;
                break;
            }
            else
                continue;
        }
    }
}
```

(schlecht)

```
int N = 65535;
int p = 1117;
int q = 137;
int a = 0;
while ((a * p) % N != q)
    a++;
int t = 2;
while (a % t != 0)
    t++;
printf("a=_%d\nt=_%d\n", a, t);
```

(deutlich besser)

## 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  
}
- Das Hauptprogramm ist eine gewöhnliche Funktion mit dem Namen `main()` und dem Rückgabewert `int`:  
0: Erfolg  
≠ 0: Fehler

## 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;  
}
```

# Hardwarenahe Programmierung

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

## 1 Einführung

## 2 Einführung in C

...

2.5 Verzweigungen

2.6 Schleifen

2.7 Seiteneffekte

2.8 Strukturierte Programmierung

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;
```

```
}
```

```
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

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

```
int main (void)
{
    char hello[] = "Hello, _world!\n";
    printf ("%s", hello);
    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   **%s** String

## 2.11 Arrays und Strings

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

```
int main (void)
{
    char *hello = "Hello, _world!\n";
    printf ("%s", hello);
    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:

<b>%d</b>	dezimal	<b>%c</b>	Zeichen
<b>%x</b>	hexadezimal	<b>%s</b>	String

## 2.11 Arrays und Strings

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

```
int main (void)
{
    char *hello = "Hello, _world!\n";
    while (*hello)
        printf ("%c", *hello++);
    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:

<b>%d</b> dezimal	<b>%c</b> Zeichen
<b>%x</b> hexadezimal	<b>%s</b> String



## 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 = { 30, 10, 2014 };
```

```
    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 = 30;
```

```
    (*d).month = 10;
```

```
    (*d).year = 2014;
```

```
}
```

```
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 = 30;  
    d->month = 10;  
    d->year = 2014;  
}
```

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;  
}
```

# Hardwarenahe Programmierung

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

## 1 Einführung

## 2 Einführung in C

...

2.5 Verzweigungen

2.6 Schleifen

2.7 Seiteneffekte

2.8 Strukturierte Programmierung

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

29. Oktober 2018

# Hardwarenahe Programmierung

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

## 1 Einführung

## 2 Einführung in C

...

2.7 Seiteneffekte

2.8 Strukturierte Programmierung

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 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;  
}
```

Ende-Markierung



- `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;  
}
```

Ende-Markierung



**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;
```

```
}
```

- 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

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

```
int main (void)
{
    char hello[] = "Hello, _world!\n";
    printf ("%s", hello);
    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:

<b>%d</b>	dezimal	<b>%c</b>	Zeichen
<b>%x</b>	hexadezimal	<b>%s</b>	String

## 2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

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

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

```
    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:

<b>%d</b>	dezimal	<b>%c</b>	Zeichen
<b>%x</b>	hexadezimal	<b>%s</b>	String

## 2.11 Arrays und Zeiger und Strings

h = 

H	e	l	l	o	!	\n	Null
---	---	---	---	---	---	----	------

H	e	l	l	o	!	\n	Null
---	---	---	---	---	---	----	------

  
↑  
h = 

--

```
char h[] = "Hello!\n";
```

```
printf ("%s", h);
```

```
h[1] = 'a';
```

```
int i = 0;
```

```
while (h[i])  
    printf ("%c", h[i++]);
```

```
while (*h)  
    printf ("%c", *h++);
```

```
char *h = "Hello!\n";
```

```
printf ("%s", h);
```

```
h[1] = 'a';
```

```
int i = 0;
```

```
while (h[i])  
    printf ("%c", h[i++]);
```

```
while (*h)  
    printf ("%c", *h++);
```

## 2.11 Arrays und Zeiger und Strings

h = 

H	e	l	l	o	!	\n	Null
---	---	---	---	---	---	----	------

h = 

H	e	l	l	o	!	\n	Null
---	---	---	---	---	---	----	------

↑  
ohne Schreibzugriff

```
char h[] = "Hello!\n";
```

```
printf ("%s", h);
```

```
h[1] = 'a';
```

```
int i = 0;
```

```
while (h[i])  
    printf ("%c", h[i++]);
```

```
while (*h)  
    printf ("%c", *h++);
```

```
char *h = "Hello!\n";
```

```
printf ("%s", h);
```

~~h[1] = 'a';~~

```
int i = 0;
```

```
while (h[i])  
    printf ("%c", h[i++]);
```

```
while (*h)  
    printf ("%c", *h++);
```

## 2.11 Arrays und Zeiger und Strings

h = 

H	e	l	l	o	!	\n	Null
---	---	---	---	---	---	----	------

Adresse des Arrays

```
char h[] = "Hello!\n";
```

```
printf ("%s", h);
```

```
h[1] = 'a';
```

```
int i = 0;
```

```
while (h[i])  
    printf ("%c", h[i++]);
```

```
while (*h)  
    printf ("%c", *h++);
```

H	e	l	l	o	!	\n	Null
---	---	---	---	---	---	----	------

h = 

--

ohne Schreibzugriff

```
char *h = "Hello!\n";
```

```
printf ("%s", h);
```

~~h[1] = 'a';~~

```
int i = 0;
```

```
while (h[i])  
    printf ("%c", h[i++]);
```

```
while (*h)  
    printf ("%c", *h++);
```

## 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;
```

- Eigener Datentyp: `date`
- zusammengesetzt aus elementareren Datentypen

```
int main (void)
```

```
{  
    date today = { 30, 10, 2014 };  
    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 = 30;  
    (*d).month = 10;  
    (*d).year = 2014;  
}
```

- Eigener Datentyp: `date`
- zusammengesetzt aus elementarerer Datentypen
- `set_date()`: „Methode“ von `date`
- `d->day` ist Abkürzung für `(*d).day`

```
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 = 30;  
    d->month = 10;  
    d->year = 2014;  
}
```

- Eigener Datentyp: `date`
- zusammengesetzt aus elementarerer Datentypen
- `set_date()`: „Methode“ von `date`
- `d->day` ist Abkürzung für `(*d).day`

```
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 (-1, 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 (-1, 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;
}
```

# Hardwarenahe Programmierung

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

## 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

...

...

# Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

5. November 2018

# Hardwarenahe Programmierung

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

## 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

...

...

## 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 (1, 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]_=%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; i < argc; i++)
        printf ("argv[%d]=\n", i, argv[i]);
    return 0;
}
```

- **argc**: Anzahl der Parameter
- **argv**: Array von Strings  
(= Zeiger auf Zeiger auf **chars**)  
mit den Parametern
- Parameter Nr. 0:  
Name des Programms selbst,  
wie es aufgerufen wurde

## 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;
}
```

- **argc**: Anzahl der Parameter
- **argv**: Array von Strings  
(= Zeiger auf Zeiger auf **chars**)  
mit den Parametern
- Parameter Nr. 0:  
Name des Programms selbst,  
wie es aufgerufen wurde
- letzter Eintrag im Array:  
**NULL**-Zeiger  
(als Ende-Markierung)

## 2.14 Parameter des Hauptprogramms

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

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

- **argc**: Anzahl der Parameter
- **argv**: Array von Strings  
(= Zeiger auf Zeiger auf **chars**)  
mit den Parametern
- Parameter Nr. 0:  
Name des Programms selbst,  
wie es aufgerufen wurde
- letzter Eintrag im Array:  
**NULL**-Zeiger  
(als Ende-Markierung)

## 2.14 Parameter des Hauptprogramms

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

```
int main (int argc, char **argv)
```

```
{
```

```
    printf ("argc=_=%d\n", argc);
```

```
    int i = 0;
```

```
    while (*argv)
```

```
    {
```

```
        printf ("argv[%d]_=%s\n", i, *argv);
```

```
        i++;
```

```
        argv++;
```

```
    }
```

```
    return 0;
```

```
}
```

- **argc**: Anzahl der Parameter
- **argv**: Array von Strings  
(= Zeiger auf Zeiger auf **chars**)  
mit den Parametern
- Parameter Nr. 0:  
Name des Programms selbst,  
wie es aufgerufen wurde
- letzter Eintrag im Array:  
**NULL**-Zeiger  
(als Ende-Markierung)

# Hardwarenahe Programmierung

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

## 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

...

...

## 2.15 String-Operationen

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

```
int main (void)
```

```
{
```

```
    char name[100];
```

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

```
    fgets (name, 100, stdin);
```

```
    printf ("Hallo,_%s!\n", name);
```

```
    return 0;
```

```
}
```

Probleme mit `scanf ("%s", name)`:

- Leerzeichen beendet Eingabe
- keine Prüfung der Puffergröße



## 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 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;
```

```
}
```

- `strlen()` gibt die Länge eines Strings zurück.
- Es enthält eine Schleife.

## 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;
```

```
}
```

- `strlen()` gibt die Länge eines Strings zurück.
- Es enthält eine Schleife.
- Typ des Rückgabewerts: `size_t` = ganze Zahl von der Größe eines Zeigers
- in `printf()`: `%zd` (*size*)

## 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;
```

```
}
```

- `strlen()` gibt die Länge eines Strings zurück.
- Es enthält eine Schleife.
- Typ des Rückgabewerts: `size_t` = ganze Zahl von der Größe eines Zeigers
- in `printf()`: `%zd` (*size*)
- Zeiger erhöhen: String vorne abschneiden

## 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;
```

```
}
```

- `strlen()` gibt die Länge eines Strings zurück.
- Es enthält eine Schleife.
- Typ des Rückgabewerts: `size_t` = ganze Zahl von der Größe eines Zeigers
- in `printf()`: `%zd` (*size*)
- Zeiger erhöhen: String vorne abschneiden
- 0-Symbol (= Ende-Markierung) in den String schreiben: String hinten abschneiden

## 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;
```

```
}
```

- `strlen()` gibt die Länge eines Strings zurück.
- Es enthält eine Schleife.
- Typ des Rückgabewerts: `size_t` = ganze Zahl von der Größe eines Zeigers
- in `printf()`: `%zd` (*size*)
- Zeiger erhöhen: String vorne abschneiden
- 0-Symbol (= Ende-Markierung) in den String schreiben: String hinten abschneiden
- **Der für den String reservierte Speicherplatz bleibt derselbe!**

## 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 *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;
```

```
}
```

- **strcmp()**: Strings vergleichen
- alphabetisch nach ASCII (Groß- < Kleinbuchstaben, ohne Umlaute usw.)
- Rückgabewert:
  - 1, wenn linker String kleiner,
  - + 1, wenn rechter String kleiner,
  - 0, wenn beide Strings gleich



## 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;
```

```
}
```

- **strcmp()**: Strings vergleichen
- alphabetisch nach ASCII  
(Groß- < Kleinbuchstaben,  
ohne Umlaute usw.)
- Rückgabewert:
  - 1, wenn linker String kleiner,
  - + 1, wenn rechter String kleiner,
  - 0, wenn beide Strings gleich
- **strcat()**: String anhängen  
(*concatenate*)

## 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;
```

```
}
```

- **strcmp()**: Strings vergleichen
- alphabetisch nach ASCII  
(Groß- < Kleinbuchstaben,  
ohne Umlaute usw.)
- Rückgabewert:
  - 1, wenn linker String kleiner,
  - + 1, wenn rechter String kleiner,
  - 0, wenn beide Strings gleich
- **strcat()**: String anhängen  
(*concatenate*)
- **Ob der Speicherplatz reicht,  
wird nicht geprüft!**

## 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>

- **sprintf()**: in einen String schreiben

**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] = "";
```

```
    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;
```

```
}
```

- `sprintf()`: in einen String schreiben
- **Ob der Speicherplatz reicht, wird nicht geprüft!**

## 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;
```

```
}
```

- **sprintf()**: in einen String schreiben
- **Ob der Speicherplatz reicht, wird nicht geprüft!**
- **strstr()**: String in String suchen
- Rückgabewert: Zeiger auf den gefundenen String
- Index berechnen:  
Zeiger – Zeiger = Zahl  
von der Größe eines Zeigers

## 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.git>

## 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

...

...



# Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

12. November 2018

# Hardwarenahe Programmierung

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

## 1 Einführung

## 2 Einführung in C

...

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

...

...

## 2.15 String-Operationen

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

```
int main (void)
```

```
{
```

```
    char name[100];
```

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

```
    fgets (name, 100, stdin);
```

```
    printf ("Hallo,_%s!\n", name);
```

```
    return 0;
```

```
}
```

Probleme mit `scanf ("%s", name)`:

- Leerzeichen beendet Eingabe
- keine Prüfung der Puffergröße

## 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;
```

```
}
```

- `strlen()` gibt die Länge eines Strings zurück.
- Es enthält eine Schleife.
- Typ des Rückgabewerts: `size_t` = ganze Zahl von der Größe eines Zeigers
- in `printf()`: `%zd` (*size*)
- Zeiger erhöhen: String vorne abschneiden
- 0-Symbol (= Ende-Markierung) in den String schreiben: String hinten abschneiden
- **Der für den String reservierte Speicherplatz bleibt derselbe!**

## 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;
```

```
}
```

- **strcmp()**: Strings vergleichen
- alphabetisch nach ASCII  
(Groß- < Kleinbuchstaben,  
ohne Umlaute usw.)
- Rückgabewert:
  - 1, wenn linker String kleiner,
  - + 1, wenn rechter String kleiner,
  - 0, wenn beide Strings gleich
- **strcat()**: String anhängen  
(*concatenate*)
- **Ob der Speicherplatz reicht,  
wird nicht geprüft!**

## 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;
```

```
}
```

- **sprintf()**: in einen String schreiben
- **Ob der Speicherplatz reicht, wird nicht geprüft!**
- **strstr()**: String in String suchen
- Rückgabewert: Zeiger auf den gefundenen String
- Index berechnen:  
Zeiger – Zeiger = Zahl  
von der Größe eines Zeigers

## 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

Prof. Dr. rer. nat. Peter Gerwinski

19. November 2018



# Hardwarenahe Programmierung

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

## 1 Einführung

## 2 Einführung in C

...

### 2.15 String-Operationen

## 3 Bibliotheken

### 3.1 Der Präprozessor

### 3.2 Bibliotheken einbinden

### 3.3 Bibliotheken verwenden

### 3.4 Projekt organisieren: make

## 4 Hardwarenahe Programmierung

## 5 Algorithmen

### 5.1 Differentialgleichungen

...

...

## 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** VIER 4: 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 VIER 4**: 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** VIER 4: Text ersetzen lassen – Konstante definieren

- Kein Semikolon!
- Berechnungen in Klammern setzen:  
**#define** VIER (2 + 2)

## 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** VIER 4: Text ersetzen lassen – Konstante definieren

- Kein Semikolon!
- Berechnungen in Klammern setzen:  
**#define** VIER (2 + 2)
- 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` 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` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß:

```
$ pkg-config --cflags --libs gtk+-3.0
-pthread -I/usr/include/gtk-3.0 -I/usr/include/at-spi2-atk/2.0 -I/usr/include/at-spi-2.0 -I/usr/include/dbus-1.0 -I/usr/lib/x86_64-linux-gnu/dbus-1.0/include -I/usr/include/gtk-3.0 -I/usr/include/gio-unix-2.0/ -I/usr/include/cairo -I/usr/include/pango-1.0 -I/usr/include/harfbuzz -I/usr/include/pango-1.0 -I/usr/include/atk-1.0 -I/usr/include/cairo -I/usr/include/pixman-1 -I/usr/include/freetype2 -I/usr/include/libpng16 -I/usr/include/gdk-pixbuf-2.0 -I/usr/include/libpng16 -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -lgtk-3 -lgdk-3 -lpangocairo-1.0 -lpango-1.0 -latk-1.0 -lcairo-gobject -lcairo -lgdk_pixbuf-2.0 -lgio-2.0 -lgobject-2.0 -lglib-2.0
```

## 3.3 Bibliothek verwenden (Beispiel: GTK+)

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

→ Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c -pthread -I/usr/include/gtk-3.0 -I/usr/include/at-spi2-atk/2.0 -I/usr/include/at-spi-2.0 -I/usr/include/dbus-1.0 -I/usr/lib/x86_64-linux-gnu/dbus-1.0/include -I/usr/include/gtk-3.0 -I/usr/include/gio-unix-2.0/ -I/usr/include/cairo -I/usr/include/pango-1.0 -I/usr/include/harfbuzz -I/usr/include/pango-1.0 -I/usr/include/atk-1.0 -I/usr/include/cairo -I/usr/include/pixman-1 -I/usr/include/freetype2 -I/usr/include/libpng16 -I/usr/include/gdk-pixbuf-2.0 -I/usr/include/libpng16 -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -lgtk-3 -lgdk-3 -lpangocairo-1.0 -lpango-1.0 -latk-1.0 -lcairo-gobject -lcairo -lgdk_pixbuf-2.0 -lgio-2.0 -lgobject-2.0 -lglib-2.0 -o hello-gtk
```

## 3.3 Bibliothek verwenden (Beispiel: GTK+)

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

→ Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c $(pkg-config --cflags --libs)
    -o hello-gtk
```

## 3.3 Bibliothek verwenden (Beispiel: GTK+)

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

→ Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c $(pkg-config --cflags --libs)  
-o hello-gtk
```

Optionen:

u. a. viele Include-Verzeichnisse:

`-I/usr/include/gtk-3.0`

Bibliotheken:

u. a. `-lgtk-3 -lcairo`

## 3.3 Bibliothek verwenden (Beispiel: GTK+)

Selbst geschriebene Funktion übergeben: *Callback*

```
gboolean draw (GtkWidget *widget, cairo_t *c, gpointer data)
{
    /* Zeichenbefehle */
    ...

    return FALSE;
}

...
```

```
g_signal_connect (drawing_area, "draw", G_CALLBACK (draw), NULL);
```


→ GTK+ ruft immer dann, wenn es etwas zu zeichnen gibt,  
die Funktion `draw` auf.

### 3.3 Bibliothek verwenden (Beispiel: GTK+)

Selbst geschriebene Funktion übergeben: *Callback*

```
gboolean draw (GtkWidget *widget, cairo_t *c, gpointer data)
{
    /* Zeichenbefehle */
    ...
    return FALSE;
}
...
```

repräsentiert den  
Bildschirm, auf den  
gezeichnet werden soll



```
g_signal_connect (drawing_area, "draw", G_CALLBACK (draw), NULL);
```

→ GTK+ ruft immer dann, wenn es etwas zu zeichnen gibt,  
die Funktion `draw` auf.

### 3.3 Bibliothek verwenden (Beispiel: GTK+)

Selbst geschriebene Funktion übergeben: *Callback*

```
gboolean draw (GtkWidget *widget, cairo_t *c, gpointer data)
{
```

```
    /* Zeichenbefehle */
```

```
    ...
```

```
    return FALSE;
```

```
}
```

```
...
```

```
g_signal_connect (drawing_area, "draw", G_CALLBACK (draw), NULL);
```

repräsentiert den  
Bildschirm, auf den  
gezeichnet werden soll

optionale Zusatzinformationen  
für draw(), typischerweise  
ein Zeiger auf ein struct

→ GTK+ ruft immer dann, wenn es etwas zu zeichnen gibt,  
die Funktion `draw` auf.



## 3.3 Bibliothek verwenden (Beispiel: GTK+)

Selbst geschriebene Funktion übergeben: *Callback*

```
gboolean timer (GtkWidget *widget)
{
    /* Rechenbefehle */
    ...

    gtk_widget_queue_draw_area (widget, 0, 0, WIDTH, HEIGHT);
    g_timeout_add (50, (GSourceFunc) timer, widget);
    return FALSE;
}

...

g_timeout_add (50, (GSourceFunc) timer, drawing_area);
```

→ GTK+ ruft nach 50 Millisekunden die Funktion `timer` auf.

## 3.3 Bibliothek verwenden (Beispiel: GTK+)

Selbst geschriebene Funktion übergeben: *Callback*

```
gboolean timer (GtkWidget *widget)
{
    /* Rechenbefehle */
    ...
```

Dieser Bereich soll  
neu gezeichnet werden.



```
    gtk_widget_queue_draw_area (widget, 0, 0, WIDTH, HEIGHT);
    g_timeout_add (50, (GSourceFunc) timer, widget);
    return FALSE;
}
```

...

```
g_timeout_add (50, (GSourceFunc) timer, drawing_area);
```


→ GTK+ ruft nach 50 Millisekunden die Funktion `timer` auf.

## 3.3 Bibliothek verwenden (Beispiel: GTK+)

Selbst geschriebene Funktion übergeben: *Callback*

```
gboolean timer (GtkWidget *widget)
{
    /* Rechenbefehle */
    ...
```

Dieser Bereich soll  
neu gezeichnet werden.



```
gtk_widget_queue_draw_area (widget, 0, 0, WIDTH, HEIGHT);
g_timeout_add (50, (GSourceFunc) timer, widget);
return FALSE;
}
```

In weiteren 50 Millisekunden soll  
die Funktion erneut aufgerufen werden.

```
g_timeout_add (50, (GSourceFunc) timer, drawing_area);
```

→ GTK+ ruft nach 50 Millisekunden die Funktion *timer* auf.

## 3.3 Bibliothek verwenden (Beispiel: GTK+)

Selbst geschriebene Funktion übergeben: *Callback*

```
gboolean timer (GtkWidget *widget)
{
    /* Rechenbefehle */
    ...
```

Dieser Bereich soll  
neu gezeichnet werden.



```
    gtk_widget_queue_draw_area (widget, 0, 0, WIDTH, HEIGHT);
    g_timeout_add (50, (GSourceFunc) timer, widget);
    return FALSE;
}
```

In weiteren 50 Millisekunden soll  
die Funktion erneut aufgerufen werden.

Explizite Typumwandlung  
eines Zeigers (später)



```
g_timeout_add (50, (GSourceFunc) timer, drawing_area);
```

→ GTK+ ruft nach 50 Millisekunden die Funktion `timer` auf.

# 5 Algorithmen

## 5.1 Differentialgleichungen

### Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$\frac{dx}{dt} = v_x \Rightarrow x(t) = x_0 + v_x t$$

$$\frac{dx}{dt} = v_x \Rightarrow dx = v_x \cdot dt$$
$$x += dx$$

$$\frac{dy}{dt} = v_y \Rightarrow y(t) = y_0 + v_y t$$

---

$$\frac{dy}{dt} = v_y(t) \Rightarrow y(t) = y_0 + \int v_y(t) dt = y_0 + v_{y0} t - \frac{1}{2} g t^2$$

$$\frac{dv_y}{dt} = -g \Rightarrow v_y(t) = v_{y0} - g t$$

$$\frac{dv_y}{dt} = -g \Rightarrow dv_y = -g \cdot dt$$
$$v_y += -g \cdot dt$$

# 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

$$x'(t) = v_x(t) \qquad x(t) = \int v_x(t) dt = \int v_{0x} dt = x_0 + v_{0x} \cdot t$$

$$y'(t) = v_y(t) \qquad y(t) = \int v_y(t) dt = \int v_{0y} - g \cdot t dt = y_0 + v_{0y} \cdot t - \frac{1}{2}gt^2$$

$\Rightarrow$

$$v'_x(t) = 0 \qquad v_x(t) = \int 0 dt = v_{0x}$$

$$v'_y(t) = -g \qquad v_y(t) = \int -g dt = v_{0y} - g \cdot t$$

# 5 Algorithmen

## 5.1 Differentialgleichungen

### Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \quad x += v_x * dt;$$

$$y'(t) = v_y(t) \quad y += v_y * dt;$$

$\Rightarrow$

Siehe: [gtk-13.c](#)

$$v'_x(t) = 0 \quad v_x += 0 * dt;$$

$$v'_y(t) = -g \quad v_y += -g * dt;$$

# 5 Algorithmen

## 5.1 Differentialgleichungen

**Beispiel 1: Gleichmäßig beschleunigte Bewegung**

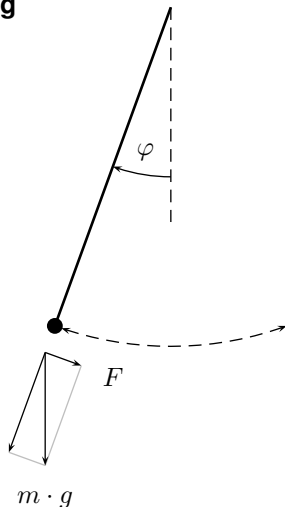
**Beispiel 2: Mathematisches Pendel**

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

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

- 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**

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

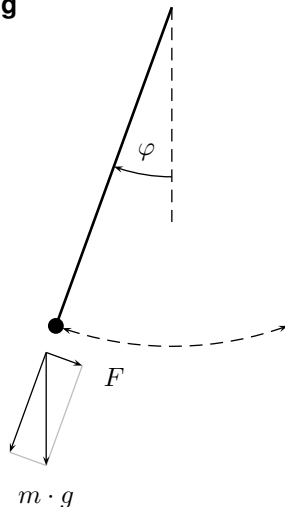
$$\omega'(t) = -\frac{g}{l} \cdot \sin \varphi(t)$$

- 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

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

## 1 Einführung

## 2 Einführung in C

## 3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliotheken verwenden

3.4 Projekt organisieren: make

## 4 Hardwarenahe Programmierung

4.1 Bit-Operationen

4.2 I/O-Ports

4.3 Interrupts

...

## 5 Algorithmen

5.1 Differentialgleichungen

...

...

# Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

26. November 2018

# Hardwarenahe Programmierung

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

## 1 Einführung

## 2 Einführung in C

## 3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliotheken verwenden

3.4 Projekt organisieren: make

## 4 Hardwarenahe Programmierung

4.1 Bit-Operationen

4.2 I/O-Ports

4.3 Interrupts

...

## 5 Algorithmen

5.1 Differentialgleichungen

...

...

## 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** VIER 4: Text ersetzen lassen – Konstante definieren

- Kein Semikolon!
- Berechnungen in Klammern setzen:  
**#define** VIER (2 + 2)
- 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>`
- Mit `pkg-config --cflags --libs gtk+-3.0` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß:

```
$ pkg-config --cflags --libs gtk+-3.0
-pthread -I/usr/include/gtk-3.0 -I/usr/include/at-spi2-atk/2.0 -I/usr/include/at-spi-2.0 -I/usr/include/dbus-1.0 -I/usr/lib/x86_64-linux-gnu/dbus-1.0/include -I/usr/include/gtk-3.0 -I/usr/include/gio-unix-2.0/ -I/usr/include/cairo -I/usr/include/pango-1.0 -I/usr/include/harfbuzz -I/usr/include/pango-1.0 -I/usr/include/atk-1.0 -I/usr/include/cairo -I/usr/include/pixman-1 -I/usr/include/freetype2 -I/usr/include/libpng16 -I/usr/include/gdk-pixbuf-2.0 -I/usr/include/libpng16 -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -lgtk-3 -lgdk-3 -lpangocairo-1.0 -lpango-1.0 -latk-1.0 -lcairo-gobject -lcairo -lgdk_pixbuf-2.0 -lgio-2.0 -lgobject-2.0 -lglib-2.0
```

## 3.3 Bibliothek verwenden (Beispiel: GTK+)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk+-3.0` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

→ Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c -pthread -I/usr/include/gtk-3.0 -I/usr/include/at-spi2-atk/2.0 -I/usr/include/at-spi2.0 -I/usr/include/dbus-1.0 -I/usr/lib/x86_64-linux-gnu/dbus-1.0/include -I/usr/include/gtk-3.0 -I/usr/include/gio-unix-2.0/ -I/usr/include/cairo -I/usr/include/pango-1.0 -I/usr/include/harfbuzz -I/usr/include/pango-1.0 -I/usr/include/atk-1.0 -I/usr/include/cairo -I/usr/include/pixman-1 -I/usr/include/freetype2 -I/usr/include/libpng16 -I/usr/include/gdk-pixbuf-2.0 -I/usr/include/libpng16 -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -lgtk-3 -lgdk-3 -lpangocairo-1.0 -lpango-1.0 -latk-1.0 -lcairo-gobject -lcairo -lgdk_pixbuf-2.0 -lgio-2.0 -lgobject-2.0 -lglib-2.0 -o hello-gtk
```

## 3.3 Bibliothek verwenden (Beispiel: GTK+)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk+-3.0` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

→ Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c $(pkg-config --cflags --libs  
    gtk+-3.0) -o hello-gtk
```

Optionen:

u. a. viele Include-Verzeichnisse:

`-I/usr/include/gtk-3.0`

Bibliotheken:

u. a. `-lgtk-3 -lcairo`



### 3.3 Bibliothek verwenden (Beispiel: GTK+)

Selbst geschriebene Funktion übergeben: *Callback*

```
gboolean draw (GtkWidget *widget, cairo_t *c, gpointer data)
{
    /* Zeichenbefehle */
    ...
    return FALSE;
}
```

repräsentiert den  
Bildschirm, auf den  
gezeichnet werden soll

optionale Zusatzinformationen  
für draw(), typischerweise  
ein Zeiger auf ein struct

```
g_signal_connect (drawing_area, "draw", G_CALLBACK (draw), NULL);
```

→ GTK+ ruft immer dann, wenn es etwas zu zeichnen gibt,  
die Funktion `draw` auf.

## 3.3 Bibliothek verwenden (Beispiel: GTK+)

Selbst geschriebene Funktion übergeben: *Callback*

```
gboolean timer (GtkWidget *widget)
{
    /* Rechenbefehle */
    ...
```

Dieser Bereich soll  
neu gezeichnet werden.



```
    gtk_widget_queue_draw_area (widget, 0, 0, WIDTH, HEIGHT);
    g_timeout_add (50, (GSourceFunc) timer, widget);
    return FALSE;
}
```

In weiteren 50 Millisekunden soll  
die Funktion erneut aufgerufen werden.

Explizite Typumwandlung  
eines Zeigers (später)



```
g_timeout_add (50, (GSourceFunc) timer, drawing_area);
```

→ GTK+ ruft nach 50 Millisekunden die Funktion `timer` auf.

## 3.4 Projekt organisieren: make

- Regeln
- Makros

## 3.4 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

## 3.4 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)

## 3.4 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

## 3.4 Projekt organisieren: make

- explizite und implizite Regeln
- Makros

→ 3 Sprachen: C, Präprozessor, make

# 5 Algorithmen

## 5.1 Differentialgleichungen

### Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \qquad x(t) = \int v_x(t) dt = \int v_{0x} dt = x_0 + v_{0x} \cdot t$$

$$y'(t) = v_y(t) \qquad y(t) = \int v_y(t) dt = \int v_{0y} - g \cdot t dt = y_0 + v_{0y} \cdot t - \frac{1}{2}gt^2$$

$\Rightarrow$

$$v'_x(t) = 0 \qquad v_x(t) = \int 0 dt = v_{0x}$$

$$v'_y(t) = -g \qquad v_y(t) = \int -g dt = v_{0y} - g \cdot t$$



# 5 Algorithmen

## 5.1 Differentialgleichungen

### Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \quad x += v_x * dt;$$

$$y'(t) = v_y(t) \quad y += v_y * dt;$$

$\Rightarrow$

Siehe: [gtk-13.c](#)

$$v'_x(t) = 0 \quad v_x += 0 * dt;$$

$$v'_y(t) = -g \quad v_y += -g * dt;$$

# 5 Algorithmen

## 5.1 Differentialgleichungen

**Beispiel 1: Gleichmäßig beschleunigte Bewegung**

**Beispiel 2: Mathematisches Pendel**

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

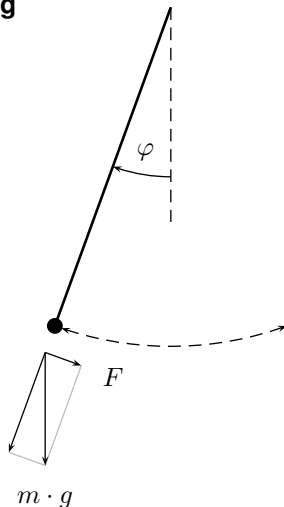
$$\omega'(t) = -\frac{g}{l} \cdot \sin \varphi(t)$$

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

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

**Beispiel 3: Weltraum-Simulation**

Praktikumsaufgabe



# Hardwarenahe Programmierung

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

## 1 Einführung

## 2 Einführung in C

## 3 Bibliotheken

### 3.1 Der Präprozessor

### 3.2 Bibliotheken einbinden

### 3.3 Bibliotheken verwenden

### 3.4 Projekt organisieren: make

## 4 Hardwarenahe Programmierung

### 4.1 Bit-Operationen

### 4.2 I/O-Ports

### 4.3 Interrupts

...

## 5 Algorithmen

### 5.1 Differentialgleichungen

...

...

## 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	<b>0</b>	0000	<b>0</b>	1000	<b>8</b>
001	<b>1</b>	0001	<b>1</b>	1001	<b>9</b>
010	<b>2</b>	0010	<b>2</b>	1010	<b>A</b>
011	<b>3</b>	0011	<b>3</b>	1011	<b>B</b>
100	<b>4</b>	0100	<b>4</b>	1100	<b>C</b>
101	<b>5</b>	0101	<b>5</b>	1101	<b>D</b>
110	<b>6</b>	0110	<b>6</b>	1110	<b>E</b>
111	<b>7</b>	0111	<b>7</b>	1111	<b>F</b>

- 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))`

# Hardwarenahe Programmierung

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

## 1 Einführung

## 2 Einführung in C

## 3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliotheken verwenden

3.4 Projekt organisieren: make

## 4 Hardwarenahe Programmierung

4.1 Bit-Operationen

4.2 I/O-Ports

4.3 Interrupts

...

## 5 Algorithmen

5.1 Differentialgleichungen

...

...

# Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

3. Dezember 2018



# Hardwarenahe Programmierung

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

## 1 Einführung

## 2 Einführung in C

## 3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliotheken verwenden

3.4 Projekt organisieren: make

## 4 Hardwarenahe Programmierung

4.1 Bit-Operationen

4.2 Programmierung von Mikrocontrollern

4.3 I/O-Ports

4.4 Interrupts

4.5 volatile-Variable

...

## 5 Algorithmen

5.1 Differentialgleichungen

...

...

## 3.4 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

## 3.4 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)

## 3.4 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

## 3.4 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	<b>0</b>	0000	<b>0</b>	1000	<b>8</b>
001	<b>1</b>	0001	<b>1</b>	1001	<b>9</b>
010	<b>2</b>	0010	<b>2</b>	1010	<b>A</b>
011	<b>3</b>	0011	<b>3</b>	1011	<b>B</b>
100	<b>4</b>	0100	<b>4</b>	1100	<b>C</b>
101	<b>5</b>	0101	<b>5</b>	1101	<b>D</b>
110	<b>6</b>	0110	<b>6</b>	1110	<b>E</b>
111	<b>7</b>	0111	<b>7</b>	1111	<b>F</b>

- 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))`



# Hardwarenahe Programmierung

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

## 1 Einführung

## 2 Einführung in C

## 3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliotheken verwenden

3.4 Projekt organisieren: make

## 4 Hardwarenahe Programmierung

4.1 Bit-Operationen

4.2 Programmierung von Mikrocontrollern

4.3 I/O-Ports

4.4 Interrupts

4.5 volatile-Variable

...

## 5 Algorithmen

5.1 Differentialgleichungen

...

...

## 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 Programmierung von Mikrocontrollern

### Cross-Entwicklungswerkzeuge


- laufen auf „normalem“ Computer, erzeugen Software für anderen Computer oder Mikrocontroller

- Cross-Compiler, -Assembler und -Linker

```
avr-gcc -Wall -Os -mmcu=atmega328p blink.c -o blink.elf
```

- Werkzeuge zum Herunterladen<sup>1</sup> auf den Mikrocontroller

```
avr-objcopy -O ihex blink.elf blink.hex  
avrdude -P /dev/ttyACM0 -c arduino -p m328p \  
-U flash:w:blink.hex
```



„Es geht in der  
nächsten Zeile  
weiter.“

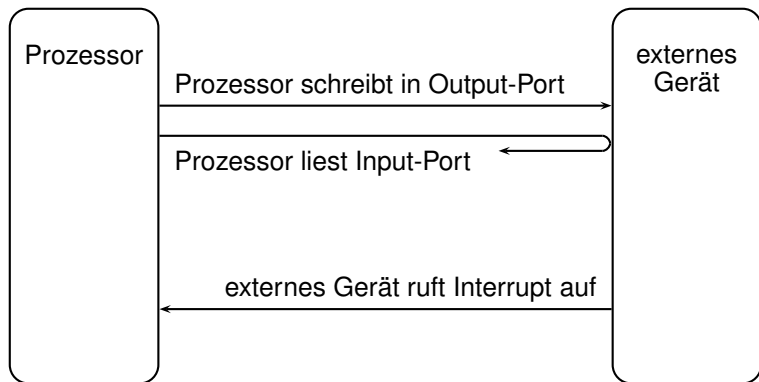
---

<sup>1</sup>So ist die Sprechweise.

## 4.3 I/O-Ports

## 4.4 Interrupts

Kommunikation mit externen Geräten



## 4.3 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*

## 4.3 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

## 4.4 Interrupts

Externes Gerät ruft (per Stromsignal) Unterprogramm auf  
Zeiger hinterlegen: „Interrupt-Vektor“

Beispiel: eingebaute Uhr

```
#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*

## 4.4 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*



## 4.4 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*

## 4.5 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;
```

```
}
```

## 4.5 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



## 4.5 volatile-Variable

Was ist eigentlich PORTD?

## 4.5 volatile-Variable

Was ist eigentlich PORTD?

```
avr-gcc -Wall -Os -mmcu=atmega328p blink-3.c -E
```

## 4.5 volatile-Variable

Was ist eigentlich PORTD?

```
avr-gcc -Wall -Os -mmcu=atmega328p blink-3.c -E
```

```
PORTD = 0x01;
```

```
→ (*(volatile uint8_t *) ((0x0B) + 0x20)) = 0x01;
```

## 4.5 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

## 4.5 volatile-Variable

Was ist eigentlich PORTD?

```
avr-gcc -Wall -Os -mmcu=atmega328p blink-3.c -E
```

```
PORTD = 0x01;
```

→ 
$$\underbrace{*(\text{volatile uint8\_t } *)}_{\substack{\text{Umwandlung in Zeiger} \\ \text{auf } \text{volatile uint8\_t}}} \underbrace{((0x0B) + 0x20)}_{\text{Zahl: } 0x2B} = 0x01;$$



## 4.5 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

## 4.5 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

## 4.5 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  
Dereferenzierung des Zeigers

Zahl: 0x2B

→ **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.git>

## 1 Einführung

## 2 Einführung in C

## 3 Bibliotheken

## 4 Hardwarenahe Programmierung

### 4.1 Bit-Operationen

### 4.2 Programmierung von Mikrocontrollern

### 4.3 I/O-Ports

### 4.4 Interrupts

### 4.5 volatile-Variable

### 4.6 Byte-Reihenfolge – Endianness

### 4.7 Binärdarstellung von Zahlen

### 4.8 Speicherausrichtung – Alignment

## 5 Algorithmen

### 5.1 Differentialgleichungen

...

...