

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

Wintersemester 2023/24

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 5. 10. 2023 unter <https://gitlab.cvh-server.de/pgerwinski/hp/tree/2023ws/20231005/>).

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

5. Oktober 2023

Hardwarenahe Programmierung

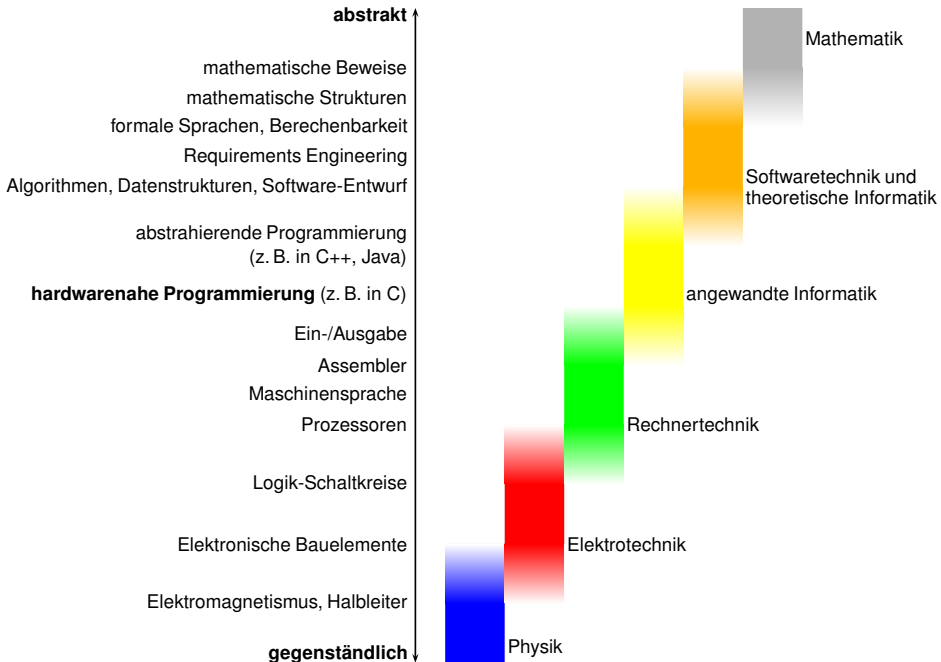
Prof. Dr. rer. nat. Peter Gerwinski

rerum naturalium = der natürlichen Dinge (lat.)

5. Oktober 2023

Vorab: Online-Werkzeuge

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



Hardwarenahe Programmierung

Man kann Computer vollständig beherrschen.

Hardwarenahe Programmierung

Man kann Computer vollständig beherrschen.

Programmierung in C

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

Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen

Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen

Hardware und/oder Betriebssystem



Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

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

Hardware und/oder Betriebssystem



Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

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

„High-Level-Assembler“

Hardware und/oder Betriebssystem



Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

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

„High-Level-Assembler“

- kein „Fallschirm“

Hardware und/oder Betriebssystem



Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

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

„High-Level-Assembler“

Hardware und/oder Betriebssystem

- kein „Fallschirm“
- kompakte Schreibweise

Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

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

„High-Level-Assembler“

Hardware und/oder Betriebssystem

- kein „Fallschirm“
- kompakte Schreibweise

Unix-Hintergrund

Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

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



„High-Level-Assembler“

Hardware und/oder Betriebssystem

- kein „Fallschirm“
- kompakte Schreibweise

Unix-Hintergrund

- Baukastenprinzip

Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

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



„High-Level-Assembler“

Hardware und/oder Betriebssystem

- kein „Fallschirm“
- kompakte Schreibweise

Unix-Hintergrund

- Baukastenprinzip
- konsequente Regeln

Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

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



„High-Level-Assembler“

Hardware und/oder Betriebssystem

- kein „Fallschirm“
- kompakte Schreibweise

Unix-Hintergrund

- Baukastenprinzip
- konsequente Regeln
- kein „Fallschirm“

Zu dieser Lehrveranstaltung



- **Lehrmaterialien:**

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

- **Klausur:**

Zeit: 120 Minuten

Zulässige Hilfsmittel:

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

- **Übungen**

sind mit der Vorlesung und dem Praktikum integriert.

- **Das Praktikum**

findet jede Woche statt.

Diese Woche: vorbereitende Maßnahmen,
Kennenlernen der verwendeten Werkzeuge.

Im Laufe des Semesters: **4 Praktikumsversuche**

Hardwarenahe Programmierung

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

1 Einführung

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

2 Einführung in C

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

3 Bibliotheken

...

2 Einführung in C

2.1 Hello, world!

Text ausgeben

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

```
int main (void)
```

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

2 Einführung in C

2.1 Hello, world!

Text ausgeben

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

```
int main (void)
```

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

printf = „print formatted“

2 Einführung in C

2.1 Hello, world!

Text ausgeben

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

```
int main (void)
```

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

printf = „print formatted“

\n: Zeilenschaltung



2.2 Programme compilieren und ausführen

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

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

```
Hello, world!
```

```
$
```


2.2 Programme compilieren und ausführen

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

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

```
Hello, world!
```

```
$
```

`-o hello-1`

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

2.2 Programme compilieren und ausführen

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

Hier: Kommandozeilen-Interface (CLI)

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

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

- Das können Sie bereits.

2.2 Programme compilieren und ausführen

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

GNU Compiler Collection (GCC) für verschiedene Plattformen:

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

2.2 Programme compilieren und ausführen

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


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

```
Hello, world!
```

```
$
```

2.2 Programme compilieren und ausführen

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



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

2.3 Elementare Aus- und Eingabe

Wert ausgeben

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

```
int main (void)
```

```
{
```

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

```
    printf (42);
```

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

```
    return 0;
```

```
}
```

2.3 Elementare Aus- und Eingabe

Wert ausgeben

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

```
int main (void)
```

```
{
```

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

```
    printf (42);
```

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

```
    return 0;
```

```
}
```

→ Absturz

2.3 Elementare Aus- und Eingabe

Wert ausgeben

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

```
int main (void)
```

```
{
```

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

```
    return 0;
```

```
}
```

Formatspezifikation „d“: „dezimal“



2.3 Elementare Aus- und Eingabe

Wert ausgeben

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

```
int main (void)
```

```
{
```

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

```
    return 0;
```

```
}
```

Formatspezifikation „d“: „dezimal“

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

2.3 Elementare Aus- und Eingabe

Wert einlesen

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

```
int main (void)
```

```
{
```

```
    double a;
```

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

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

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

```
    return 0;
```

```
}
```

Formatspezifikation „lf“:
„long floating-point“

Das „&“ nicht vergessen!

2.4 Elementares Rechnen

Wert an Variable zuweisen

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

```
int main (void)
```

```
{
```

```
    int a;
```

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

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

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

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

```
    return 0;
```

```
}
```

2.4 Elementares Rechnen

Variable bei Deklaration initialisieren

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

Achtung: Initialisierung \neq Zuweisung

Die beiden Gleichheitszeichen haben
subtil unterschiedliche Bedeutungen!

2.5 Verzweigungen

if-Verzweigung

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

2.5 Verzweigungen

if-Verzweigung

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

Wahrheitswerte in C: numerisch

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

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

2.6 Schleifen

while-Schleife

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

2.6 Schleifen

while-Schleife

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

for-Schleife

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


2.6 Schleifen

while-Schleife

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

for-Schleife

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

do-while-Schleife

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

```
int i;
```

```
i = 0;
```

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

```
{
```

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

```
    i++;
```

```
}
```

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

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

```
i = 0;
```

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

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

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

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

```
int i;
```

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

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

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

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

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

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

```
int i;
```

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

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

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

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

2.7 Strukturierte Programmierung

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

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

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

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

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

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

2.7 Strukturierte Programmierung

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

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

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

gut

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

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

nur, wenn
Sie wissen,
was Sie tun

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

Hardwarenahe Programmierung

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

1 Einführung

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

2 Einführung in C

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

...

3 Bibliotheken

...

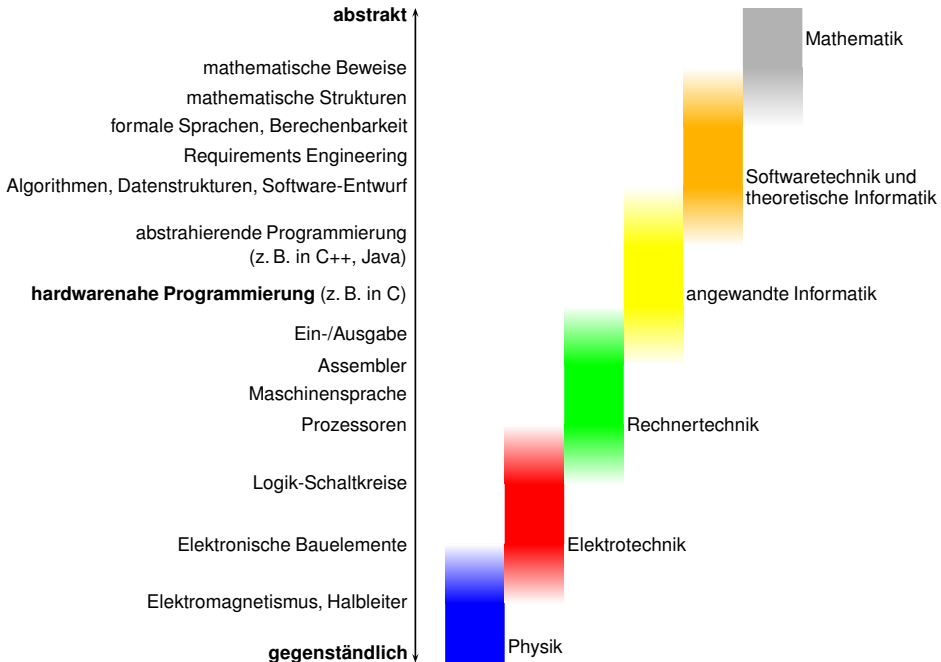
Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

12. Oktober 2023

Vorab: Online-Werkzeuge

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



Hardwarenahe Programmierung

Man kann Computer vollständig beherrschen.

Programmierung in C

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

Hardwarenahe Programmierung

Was ist C?

Etabliertes Profi-Werkzeug

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



„High-Level-Assembler“

Hardware und/oder Betriebssystem

- kein „Fallschirm“
- kompakte Schreibweise

Unix-Hintergrund

- Baukastenprinzip
- konsequente Regeln
- kein „Fallschirm“

Zu dieser Lehrveranstaltung



- **Lehrmaterialien:**

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

- **Klausur:**

Zeit: 120 Minuten

Zulässige Hilfsmittel:

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

- **Übungen**

sind mit der Vorlesung und dem Praktikum integriert.

- **Das Praktikum**

findet jede Woche statt.

Diese Woche: vorbereitende Maßnahmen,
Kennenlernen der verwendeten Werkzeuge.

Im Laufe des Semesters: **4 Praktikumsversuche**

Hardwarenahe Programmierung

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

1 Einführung

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

2 Einführung in C

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

3 Bibliotheken

...

2 Einführung in C

2.1 Hello, world!

Text ausgeben

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

```
int main (void)
```

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

printf = „print formatted“

\n: Zeilenschaltung



2.2 Programme compilieren und ausführen

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


2.2 Programme compilieren und ausführen

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

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

```
Hello, world!
```

```
$
```

`-o hello-1`

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

2.2 Programme compilieren und ausführen

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

Hier: Kommandozeilen-Interface (CLI)

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

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

- Das können Sie bereits.

2.2 Programme compilieren und ausführen

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

GNU Compiler Collection (GCC) für verschiedene Plattformen:

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

2.2 Programme compilieren und ausführen

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


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

```
Hello, world!
```

```
$
```

2.2 Programme compilieren und ausführen

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



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

2.3 Elementare Aus- und Eingabe

Wert ausgeben

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

```
int main (void)
```

```
{
```

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

```
    printf (42);
```

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

```
    return 0;
```

```
}
```

→ Absturz

2.3 Elementare Aus- und Eingabe

Wert ausgeben

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

```
int main (void)
```

```
{
```

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

```
    return 0;
```

```
}
```

Formatspezifikation „d“: „dezimal“

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

2.3 Elementare Aus- und Eingabe

Wert einlesen

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

```
int main (void)
```

```
{
```

```
    double a;
```

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

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

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

```
    return 0;
```

```
}
```

Formatspezifikation „lf“:
„long floating-point“

Das „&“ nicht vergessen!

2.4 Elementares Rechnen

Wert an Variable zuweisen

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

```
int main (void)
```

```
{
```

```
    int a;
```

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

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

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

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

```
    return 0;
```

```
}
```

2.4 Elementares Rechnen

Variable bei Deklaration initialisieren

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

Achtung: Initialisierung \neq Zuweisung

Die beiden Gleichheitszeichen haben
subtil unterschiedliche Bedeutungen!

2.5 Verzweigungen

if-Verzweigung

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

Wahrheitswerte in C: numerisch

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

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

2.6 Schleifen

while-Schleife

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

for-Schleife

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

do-while-Schleife

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

Hardwarenahe Programmierung

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

1 Einführung

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

2 Einführung in C

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

...

3 Bibliotheken

...

```
int i;
```

```
i = 0;
```

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

```
{
```

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

```
    i++;
```

```
}
```

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

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

```
i = 0;
```

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

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

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

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

```
int i;
```

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

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

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

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

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

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

```
int i;
```

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

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

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

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


2.7 Strukturierte Programmierung

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

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

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

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

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

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

2.7 Strukturierte Programmierung

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

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

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

gut

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

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

nur, wenn
Sie wissen,
was Sie tun

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

2.8 Seiteneffekte

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

```
int main (void)
```

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

2.8 Seiteneffekte

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

```
int main (void)
```

```
{
```

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

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

```
    return 0;
```

```
}
```

2.8 Seiteneffekte

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

```
int main (void)
```

```
{
```

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

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

```
    return 0;
```

```
}
```

Ausdruck als Anweisung: Wert wird ignoriert

2.8 Seiteneffekte

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

```
int main (void)
```

```
{
```

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

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

```
    return 0;
```

```
}
```

2.8 Seiteneffekte

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

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

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

2.8 Seiteneffekte

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

```
int main (void)
```

```
{
```

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

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

```
    return 0;
```

```
}
```

- `printf()` ist eine Funktion.

2.8 Seiteneffekte

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

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

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

2.8 Seiteneffekte

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

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

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

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

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

Binäre Operatoren:

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

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

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

Binäre Operatoren:

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

rot = mit Seiteneffekt

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

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

```
int i;
```

```
i = 0;
```

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

```
{
```

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

```
    i++;
```

```
}
```

Binäre Operatoren:

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

rot = mit Seiteneffekt

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

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

Binäre Operatoren:

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

```
int i;
```

```
i = 0;
```

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

```
{
```

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

```
    i++;
```

```
}
```

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

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

rot = mit Seiteneffekt

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

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

Binäre Operatoren:

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

rot = mit Seiteneffekt

```
int i;
```

```
i = 0;
```

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

```
{
```

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

```
    i++;
```

```
}
```

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

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

```
i = 0;
```

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

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

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

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

Binäre Operatoren:

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

rot = mit Seiteneffekt

```
int i;
```

```
i = 0;
```

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

```
{
```

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

```
    i++;
```

```
}
```

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

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

```
i = 0;
```

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

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

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


2.9 Funktionen

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

```
int answer (void)
```

```
{  
    return 42;  
}
```

```
void foo (void)
```

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

```
int main (void)
```

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

2.9 Funktionen

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

```
int answer (void)
```

```
{  
    return 42;  
}
```

```
void foo (void)
```

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

```
int main (void)
```

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

- Funktionsdeklaration:
Typ Name (Parameterliste)
{
 Anweisungen
}

2.9 Funktionen

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

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

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

- Funktionsdeklaration:
Typ Name (Parameterliste)
{
 Anweisungen
}

2.9 Funktionen

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

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

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

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

2.9 Funktionen

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

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

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

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

2.9 Funktionen

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

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

```
void foo (void)
```

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

```
int main (void)
```

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

2.10 Zeiger

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

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

```
{
```

```
    *a = 42;
```

```
}
```

```
int main (void)
```

```
{
```

```
    int answer;
```

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

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

```
    return 0;
```

```
}
```

2.10 Zeiger

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

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

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

- `*a` ist eine `int`.

```
int main (void)
```

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


2.10 Zeiger

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

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

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

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

2.10 Zeiger

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

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

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

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

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

2.10 Zeiger

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

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

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

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

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

- unärer Operator `&`: Adresse

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{
```

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

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

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

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

```
    return 0;
```

```
}
```

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

int main (**void**)

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

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

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

int main (**void**)

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

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

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

int main (**void**)

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

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



2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

int main (**void**)

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

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



2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

int main (**void**)

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



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

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

int main (**void**)

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



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

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

int main (**void**)

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

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

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

int main (**void**)

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

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

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

int main (**void**)

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

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

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

int main (**void**)

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

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

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

#include <stdio.h>

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

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

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

Hardwarenahe Programmierung

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

1 Einführung

2 Einführung in C

2.1 Hello, world!

2.2 Programme compilieren und ausführen

2.3 Elementare Aus- und Eingabe

2.4 Elementares Rechnen

2.5 Verzweigungen

2.6 Schleifen

2.7 Strukturierte Programmierung

2.8 Seiteneffekte

2.9 Funktionen

2.10 Zeiger

2.11 Arrays und Strings

2.12 Strukturen

...

3 Bibliotheken

...

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

19. Oktober 2023

Hardwarenahe Programmierung

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

1 Einführung

2 Einführung in C

2.1 Hello, world!

2.2 Programme compilieren und ausführen

2.3 Elementare Aus- und Eingabe

2.4 Elementares Rechnen

2.5 Verzweigungen

2.6 Schleifen

2.7 Strukturierte Programmierung

2.8 Seiteneffekte

2.9 Funktionen

2.10 Zeiger

2.11 Arrays und Strings

2.12 Strukturen

...

3 Bibliotheken

...

2.8 Seiteneffekte

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

```
int main (void)
```

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

2.8 Seiteneffekte

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

```
int main (void)
```

```
{
```

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

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

```
    return 0;
```

```
}
```

2.8 Seiteneffekte

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

```
int main (void)
```

```
{
```

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

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

```
    return 0;
```

```
}
```

Ausdruck als Anweisung: Wert wird ignoriert

2.8 Seiteneffekte

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

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

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


2.8 Seiteneffekte

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

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

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

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

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

Binäre Operatoren:

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

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

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

Binäre Operatoren:

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

rot = mit Seiteneffekt

2.8 Seiteneffekte bei Operatoren

Unäre Operatoren:

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

Binäre Operatoren:

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

rot = mit Seiteneffekt

```
int i;
```

```
i = 0;
```

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

```
{
```

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

```
    i++;
```

```
}
```

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

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

```
i = 0;
```

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

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

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

2.9 Funktionen

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

```
int answer (void)
```

```
{  
    return 42;  
}
```

```
void foo (void)
```

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

```
int main (void)
```

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

- Funktionsdeklaration:
Typ Name (Parameterliste)
{
 Anweisungen
}

2.9 Funktionen

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

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

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

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

2.9 Funktionen

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

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

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

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

2.9 Funktionen

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

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

```
void foo (void)
```

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

```
int main (void)
```

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


2.10 Zeiger

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

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

- `*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

Hardwarenahe Programmierung

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

1 Einführung

2 Einführung in C

...

2.5 Verzweigungen

2.6 Schleifen

2.7 Strukturierte Programmierung

2.8 Seiteneffekte

2.9 Funktionen

2.10 Zeiger

2.11 Arrays und Strings

2.12 Strukturen

2.13 Dateien und Fehlerbehandlung

2.14 Parameter des Hauptprogramms

2.15 String-Operationen

3 Bibliotheken

...

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

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

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

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

```
    return 0;
```

```
}
```

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

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

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

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

```
    return 0;
```

```
}
```

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

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

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

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

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

```
    return 0;
```

```
}
```

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

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

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

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

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

```
    return 0;
```

```
}
```

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

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

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

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

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

```
    return 0;
```

```
}
```

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

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

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

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

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

```
    return 0;
```

```
}
```

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

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

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

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

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

```
    return 0;
```

```
}
```

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

2.11 Arrays und Strings und Zeichen

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

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

- Welchen Zahlenwert hat '*' im Zeichensatz?

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

(normalerweise: ASCII)

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

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

...

- Groß- in Kleinbuchstaben umwandeln

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


2.12 Strukturen

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

```
typedef struct
```

```
{
```

```
    char day, month;
```

```
    int year;
```

```
}
```

```
date;
```

```
int main (void)
```

```
{
```

```
    date today = { 17, 10, 2022 };
```

```
    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 = 17;
```

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

```
    (*d).year = 2022;
```

```
}
```

```
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 = 17;  
    d->month = 10;  
    d->year = 2022;  
}
```

foo->bar ist Abkürzung für (*foo).bar

```
int main (void)
```

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

2.12 Strukturen

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

```
typedef struct
```

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

```
void set_date (date *d)  
{  
    d->day = 17;  
    d->month = 10;  
    d->year = 2022;  
}
```

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

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

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

2.13 Dateien und Fehlerbehandlung

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

```
int main (void)
```

```
{
```

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

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

```
    fclose (f);
```

```
    return 0;
```

```
}
```

2.13 Dateien und Fehlerbehandlung

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

```
int main (void)
```

```
{
```

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

```
    if (f)
```

```
    {
```

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

```
        fclose (f);
```

```
    }
```

```
    return 0;
```

```
}
```

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

2.13 Dateien und Fehlerbehandlung

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

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

```
int main (void)
```

```
{
```

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

```
    if (f)
```

```
    {
```

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

```
        fclose (f);
```

```
    }
```

```
    else
```

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

```
    return 0;
```

```
}
```

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

2.13 Dateien und Fehlerbehandlung

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

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

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

2.13 Dateien und Fehlerbehandlung

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

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

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

```
int main (void)
```

```
{
```

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

```
    if (!f)
```

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

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

```
    fclose (f);
```

```
    return 0;
```

```
}
```

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

2.13 Dateien und Fehlerbehandlung

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

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

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

```
int main (void)
```

```
{
```

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

```
    if (!f)
```

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

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

```
    fclose (f);
```

```
    return 0;
```

```
}
```

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

2.14 Parameter des Hauptprogramms

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

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

2.14 Parameter des Hauptprogramms

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

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

2.15 String-Operationen

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

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

```
int main (void)
```

```
{
```

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

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

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

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

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

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

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

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

```
    return 0;
```

```
}
```

2.15 String-Operationen

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

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

```
int main (void)
```

```
{
```

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

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

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

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

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

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

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

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

```
    return 0;
```

```
}
```

2.15 String-Operationen

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

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

```
int main (void)
```

```
{
```

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

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

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

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

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

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

```
    return 0;
```

```
}
```

2.15 String-Operationen

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

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

```
int main (void)
```

```
{
```

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

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

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

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

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

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

```
    return 0;
```

```
}
```


2 Einführung in C

Sprachelemente weitgehend komplett

Es fehlen:

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

—> werden eingeführt, wenn wir sie brauchen

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

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

—> Literatur

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

- Praxiserfahrung

—> Übung und Praktikum: nur Einstieg

—> selbständig arbeiten

Hardwarenahe Programmierung

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

1 Einführung

2 Einführung in C

...

2.5 Verzweigungen

2.6 Schleifen

2.7 Strukturierte Programmierung

2.8 Seiteneffekte

2.9 Funktionen

2.10 Zeiger

2.11 Arrays und Strings

2.12 Strukturen

2.13 Dateien und Fehlerbehandlung

2.14 Parameter des Hauptprogramms

2.15 String-Operationen

3 Bibliotheken

...