

Angewandte Informatik

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

17. Oktober 2016

Hardwarenahe Programmierung

1 Einführung

1.1 Was ist angewandte Informatik / hardwarenahe Programmierung?

1.2 Programmierung in C

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

...

3 Bibliotheken

...

Einführung

Programmierung in 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“

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 -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:_%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 ausgeben

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

```
int main (void)
```

```
{
```

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

```
    return 0;
```

```
}
```

printf() erwartet immer einen
String als ersten Parameter.

Weitere Parameter (z. B. Zahlen):
immer mit **Formatspezifikationen**

Formatspezifikation „d“: „dezimal“

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

2.3 Elementare Aus- und Eingabe

Wert ausgeben

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

```
int main (void)
```

```
{
```

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

```
    return 0;
```

```
}
```

wird nicht ausgegeben

printf() erwartet immer einen
String als ersten Parameter.

Weitere Parameter (z. B. Zahlen):
immer mit **Formatspezifikationen**

2.3 Elementare Aus- und Eingabe

Wert ausgeben

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

```
int main (void)
```

```
{
```

```
    printf (42, "_lautet_die_Antwort.\n");
```

```
    return 0;
```

```
}
```

kein String, daher **Absturz**



printf() erwartet immer einen
String als ersten Parameter.



Weitere Parameter (z. B. Zahlen):
immer mit **Formatspezifikationen**

2.3 Elementare Aus- und Eingabe

Wert einlesen

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“



**Die Formatspezifikation besagt,
was scanf() akzeptieren soll.**

2.3 Elementare Aus- und Eingabe

Wert einlesen

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

```
int main (void)
```

```
{
```

```
    double a;
```

```
    scanf ("Bitte_eine_Zahl_eingeben:_%lf", &a);
```

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

```
    return 0;
```

```
}
```

Der Benutzer muß die Zahl als
„Bitte eine Zahl eingeben: 3.14“ eingeben,
damit scanf() sie akzeptiert.

**Die Formatspezifikation besagt,
was scanf() akzeptieren soll.**

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;
```

```
}
```

2.5 Verzweigungen

if-Verzweigung

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

Wahrheitswerte in C: numerisch

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

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

2.5 Verzweigungen

if-Verzweigung

```
if (b != 0)
    printf ("%d\n", a / b);
else
    printf ("Division_durch_0?!?\n");
```

Wahrheitswerte in C: numerisch

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

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

2.5 Verzweigungen

if-Verzweigung

```
if (b != 0)
    printf ("%d\n", a / b);
else
    printf ("Division_durch_0?!?\n");
```

```
if (b == 0)
    printf ("Division_durch_0?!?\n");
else
{
    if (b == 1)
        printf ("Division_durch_1_ist_langweilig.\n");
    else
        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);
```


2.5 Verzweigungen

if-Verzweigung

```
if (b != 0)
    printf ("%d\n", a / b);
else
    printf ("Division_durch_0?!?\n");
```

```
if (b == 0)
    printf ("Division_durch_0?!?\n");
else
    if (b == 1)
        printf ("Division_durch_1_ist_langweilig.\n");
    else
        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);
```

2.5 Verzweigungen

if-Verzweigung

```
if (b != 0)
    printf ("%d\n", a / b);
else
    printf ("Division_durch_0?!?\n");
```

```
if (b == 0)
    printf ("Division_durch_0?!?\n");
else if (b == 1)
    printf ("Division_durch_1_ist_langweilig.\n");
else
    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);
```

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;  
}
```

```
$ 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.
- „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++));
```


Hardwarenahe Programmierung

1 Einführung

1.1 Was ist angewandte Informatik / hardwarenahe Programmierung?

1.2 Programmierung in C

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

...

Hardwarenahe Programmierung

1 Einführung

1.1 Was ist angewandte Informatik / hardwarenahe Programmierung?

1.2 Programmierung in C

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

...

2.9 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)`

Hardwarenahe Programmierung

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

...