

# Angewandte Informatik

Prof. Dr. rer. nat. Peter Gerwinski

22. Oktober 2015



# Angewandte Informatik

## 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

## 3 Bibliotheken

## 4 Algorithmen

## 5 Hardwarenahe Programmierung

...



## 2 Einführung in C

### 2.6 Schleifen

„Schleife“ bedeutet nicht nur  
„Und das machst Du jetzt  $n$  Male.“,  
sondern auch:  
„Hier kommst Du nicht raus, solange . . .“



## 2 Einführung in C

### 2.7 Seiteneffekte

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

```
int main (void)
```

```
{
```

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

```
    "\n";
```

```
    return 0;
```

```
}
```

← Ausdruck als Anweisung: Wert wird ignoriert



## 2 Einführung in C

### 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 Einführung in C

### 2.7 Seiteneffekte bei Operatoren

Unäre Operatoren:

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

Binäre Operatoren:

- Rechnen: `+ — * / %`
- Vergleich: `== != < > <= >=`
- Zuweisung: `= += -= *= /= %=`

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 Einführung in C

## 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;
}
```

 Funktion sofort beenden, Wert zurückgeben

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



# 2 Einführung in C

## 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 Einführung in C

### 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 Einführung in C

### 2.11 Arrays und Strings

Ein Array enthält mehrere Variablen gleichen Typs.  
Ein Zeiger zeigt auf eine Variable *und deren Nachbarn*.

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

```
int main (void)
```

```
{
```

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

```
    int i;
```

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

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

```
    return 0;
```

```
}
```



## 2 Einführung in C

### 2.11 Arrays und Strings

Ein Array enthält mehrere Variablen gleichen Typs.  
Ein Zeiger zeigt auf eine Variable *und deren Nachbarn*.

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

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



## 2 Einführung in C

### 2.11 Arrays und Strings

Ein Array enthält mehrere Variablen gleichen Typs.  
Ein Zeiger zeigt auf eine Variable *und deren Nachbarn*.

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

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int i, *p = prime;  
    for (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 Einführung in C

### 2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

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

```
    int i = 0;
```

```
    while (hello_world[i] != 0)
```

```
        printf ("%d", hello_world[i++]);
```

```
    return 0;
```

```
}
```



## 2 Einführung in C

### 2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

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

```
    int i = 0;
```

```
    while (hello_world[i])
```

```
        printf ("%d", hello_world[i++]);
```

```
    return 0;
```

```
}
```



## 2 Einführung in C

### 2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

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

```
    char *p = hello_world;
```

```
    while (*p)
```

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

```
    return 0;
```

```
}
```



## 2 Einführung in C

### 2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

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

```
    char *p = hello_world;
```

```
    while (*p)
```

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

```
    return 0;
```

```
}
```



## 2 Einführung in C

### 2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

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

```
    char *p = hello_world;
```

```
    while (*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.
- Die Formatspezifikation entscheidet über die Ausgabe:
  - %d**    dezimal
  - %x**    hexadezimal
  - %c**    Zeichen
  - %s**    String



## 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;  
}
```



## 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

→ Praktikum: nur Einstieg

→ selbständig arbeiten



# 3 Bibliotheken

## 3.1 Der Präprozessor

**#include**



## 3 Bibliotheken

### 3.1 Der Präprozessor

**#include**: Text einbinden



# 3 Bibliotheken

## 3.1 Der Präprozessor

**#include**: Text einbinden

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



# 3 Bibliotheken

## 3.1 Der Präprozessor

**#include**: Text einbinden

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



# 3 Bibliotheken

## 3.1 Der Präprozessor

**#include**: Text einbinden

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

**#define** 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



# 3 Bibliotheken

## 3.2 Bibliotheken einbinden

Inhalt der Header-Datei: externe Deklarationen

```
extern int answer (void);
```



# 3 Bibliotheken

## 3.2 Bibliotheken einbinden

Inhalt der Header-Datei: externe Deklarationen

```
extern int answer (void);
```

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



# 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



# 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



# 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
- vorcompilierte Bibliothek: `-lfoo`



# 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
- vorcompilierte Bibliothek: `-lfoo`  
= Datei `libfoo.a` in Standard-Verzeichnis



### 3.3 Bibliothek verwenden (Beispiel: OpenGL)

- Include-Dateien:

```
#include <GL/gl.h>
```

```
#include <GL/glu.h>
```

```
#include <GL/glut.h>
```

- Compiler-Aufruf:

```
gcc -Wall -O cube.c -lGL -lGLU -lglut -o cube
```



### 3.3 Bibliothek verwenden (Beispiel: OpenGL)

Selbst geschriebene Funktion übergeben: *Callback*

```
void draw (void)
```

```
{
```

```
    ...
```

```
}
```

```
...
```

```
glutDisplayFunc (draw);
```

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



### 3.3 Bibliothek verwenden (Beispiel: OpenGL)

Selbst geschriebene Funktion übergeben: *Callback*

```
void key_handler (unsigned char key, int x, int y)
```

```
{
```

```
    ...
```

```
}
```

```
...
```

```
glutKeyboardFunc (key_handler);
```

→ OpenGL ruft immer dann, wenn eine Taste gedrückt wurde, die Funktion `key_handler` auf.



### 3.3 Bibliothek verwenden (Beispiel: OpenGL)

Selbst geschriebene Funktion übergeben: *Callback*

```
void key_handler (unsigned char key, int x, int y)
```

```
{
```

```
...
```

```
}
```

```
...
```

gedrückte Taste

Mausposition

```
glutKeyboardFunc (key_handler);
```

→ OpenGL ruft immer dann, wenn eine Taste gedrückt wurde, die Funktion `key_handler` 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  
gcc -c answer.c -o answer.o
```

```
philosophy.o: philosophy.c answer.h  
gcc -c philosophy.c -o philosophy.o
```

- Makros



## 3.4 Projekt organisieren: make

- Regeln
- Makros

```
PHILOSOPHY_SOURCES = philosophy.c answer.h  
PHILOSOPHY_OBJECTS = philosophy.o answer.o  
ANSWER_SOURCES = answer.c
```

```
philosophy: $(PHILOSOPHY_OBJECTS)  
    gcc $(PHILOSOPHY_OBJECTS) -o philosophy
```

```
answer.o: $(ANSWER_SOURCES)  
    gcc -c answer.c -o answer.o
```

```
philosophy.o: $(PHILOSOPHY_SOURCES)  
    gcc -c philosophy.c -o philosophy.o
```

```
clean:  
    rm -f $(PHILOSOPHY_OBJECTS) philosophy
```



## 3.4 Projekt organisieren: make

- Regeln
- Makros

→ 3 Sprachen: C, Präprozessor, make