

Angewandte Informatik

Prof. Dr. rer. nat. Peter Gerwinski

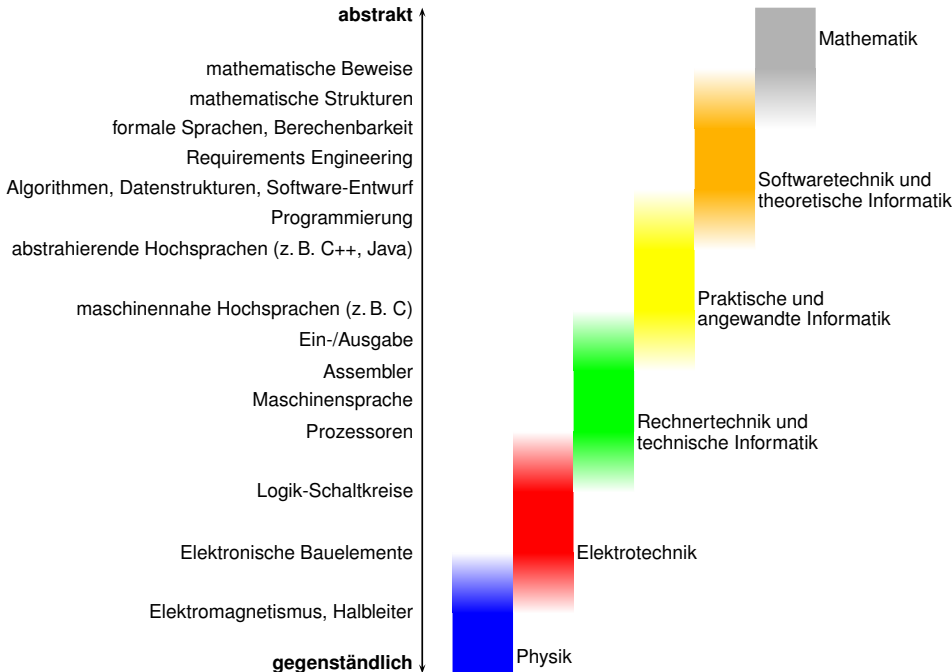
1. Oktober 2015

Angewandte Informatik

Prof. Dr. rer. nat. Peter Gerwinski

rerum naturalium = der natürlichen Dinge (lat.)

1. Oktober 2015



Angewandte Informatik

Man kann Computer vollständig beherrschen.

Rechnertechnik

Man kann vollständig verstehen, wie Computer funktionieren.

Angewandte Informatik

Man kann Computer vollständig beherrschen.

Angewandte Informatik

Programmierung

Man kann Computer vollständig beherrschen.

Angewandte Informatik

Programmierung in C

Angewandte Informatik

Programmierung in C

- kleinster gemeinsamer Nenner für viele Plattformen

Angewandte Informatik

Programmierung in C

- kleinster gemeinsamer Nenner für viele Plattformen

Hardware und/oder Betriebssystem



Angewandte Informatik

Programmierung in C

- kleinster gemeinsamer Nenner für viele Plattformen



Hardware und/oder Betriebssystem

- Programmierkenntnisse werden nicht vorausgesetzt,
aber schnelles Tempo

Angewandte Informatik

Programmierung in C

- kleinster gemeinsamer Nenner für viele Plattformen



Hardware und/oder Betriebssystem

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

Angewandte Informatik

Programmierung in C und C++

- kleinster gemeinsamer Nenner für viele Plattformen



Hardware und/oder Betriebssystem

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

Angewandte Informatik

Was ist C?

- kleinster gemeinsamer Nenner für viele Plattformen

Angewandte Informatik

Was ist C?

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

Angewandte Informatik

Was ist C?

Etabliertes Profi-Werkzeug

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

Angewandte Informatik

Was ist C?

Etabliertes Profi-Werkzeug

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

„High-Level-Assembler“

Angewandte Informatik

Was ist C?

Etabliertes Profi-Werkzeug

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

„High-Level-Assembler“

- kein „Fallschirm“

Angewandte Informatik

Was ist C?

Etabliertes Profi-Werkzeug

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

„High-Level-Assembler“

- kein „Fallschirm“
- kompakte Schreibweise

Angewandte Informatik

Was ist C?

Etabliertes Profi-Werkzeug

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

„High-Level-Assembler“

- kein „Fallschirm“
- kompakte Schreibweise

Unix-Hintergrund

Angewandte Informatik

Was ist C?

Etabliertes Profi-Werkzeug

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

„High-Level-Assembler“

- kein „Fallschirm“
- kompakte Schreibweise

Unix-Hintergrund

- Baukastenprinzip

Angewandte Informatik

Was ist C?

Etabliertes Profi-Werkzeug

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

„High-Level-Assembler“

- kein „Fallschirm“
- kompakte Schreibweise

Unix-Hintergrund

- Baukastenprinzip
- konsequente Regeln

Angewandte Informatik

Was ist C?

Etabliertes Profi-Werkzeug

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

„High-Level-Assembler“

- kein „Fallschirm“
- kompakte Schreibweise

Unix-Hintergrund

- Baukastenprinzip
- konsequente Regeln
- kein „Fallschirm“

Angewandte Informatik

1 Einführung

1.1 Was ist angewandte Informatik?

1.2 Programmierung in C

2 Einführung in C

2.1 Hello, world!

2.2 Programme compilieren und ausführen

2.3 Zahlenwerte ausgeben

2.4 Elementares Rechnen

2.5 Verzweigungen

2.6 Schleifen

2.7 Seiteneffekte

...

3 Bibliotheken

...

2 Einführung in C

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

2.2 Programme compilieren und ausführen

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

2 Einführung in C

2.2 Programme compilieren und ausführen

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

2 Einführung in C

2.3 Zahlenwerte ausgeben

Wert ausgeben

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

```
int main (void)
```

```
{
```

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

```
    return 0;
```

```
}
```

2 Einführung in C

2.4 Elementares Rechnen

Wert einlesen

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

```
int main (void)
```

```
{
```

```
    double a;
```

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

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

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

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

```
    return 0;
```

```
}
```

2 Einführung in C

2.4 Elementares Rechnen

Wert an Variable zuweisen

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

```
int main (void)
```

```
{
```

```
    int a;
```

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

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

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

```
    return 0;
```

```
}
```

2 Einführung in C

2.5 Verzweigungen

if-Verzweigung

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

2 Einführung in C

2.6 Schleifen

while-Schleife

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


2 Einführung in C

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 Einführung in C

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 Einführung in C

2.7 Seiteneffekte

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

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

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)
```

```
{
```

```
    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;  
}
```

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;  
}
```

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

```
$ ./side-effects-1
```

```
42
```

```
3
```

```
$
```

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.

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)

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: `=` `+=` `-=` `*=` `/=` `%=`

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

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`

```
int i;
```

```
i = 0;
```

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

Binäre Operatoren:

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

rot = mit Seiteneffekt

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 += 1;
```

```
}
```

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

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

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 += 1;
```

```
}
```

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

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

```
i = 0;
```

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

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

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 += 1;
```

```
}
```

```
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

```
int i;
```

```
i = 0;
```

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

```
{
```

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

```
    i += 1;
```

```
}
```

```
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

2.8 Strukturierte Programmierung

```
int i;
```

```
i = 0;
```

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

```
{
```

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

```
    i += 1;
```

```
}
```

```
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

2.8 Strukturierte Programmierung

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

```
int i;

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

```
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

2.8 Strukturierte Programmierung

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

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

```
int i;

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

```
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

2.8 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:
```

```
int i;  
  
i = 0;      gut  
while (i < 10)  
{  
    printf ("%d\n", i);  
    i += 1;  
}
```

```
for (i = 0; i < 10; i++)  
    printf ("%d\n", i);      nur, wenn  
                             Sie wissen,  
                             was Sie tun  
  
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>
```

```
int answer (void)
```

```
{
```

```
    return 42;
```

```
}
```

```
void foo (void)
```

```
{
```

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

```
}
```

```
int main (void)
```

```
{
```

```
    foo ();
```

```
    return 0;
```

```
}
```

2 Einführung in C

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 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;
}
```

- Funktionsdeklaration:
Typ Name (Parameterliste)
{
 Anweisungen
}

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;
}
```

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

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;
}
```

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

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;
}
```

- 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;
}
```

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

2 Einführung in C

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 Einführung in C

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 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;
}
```


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 Zeiger zeigt auf eine Variable.

2 Einführung in C

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

2 Einführung in C

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, i = 0;  
    while (i < 5)  
        printf ("%d\n", *(p + i++));  
    return 0;  
}
```

2 Einführung in C

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, i = 0;
```

```
    while (i < 5)
```

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

```
    return 0;
```

```
}
```

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

2 Einführung in C

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, i = 0;
```

```
    while (i < 5)
```

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

```
    return 0;
```

```
}
```

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

2 Einführung in C

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, i = 0;  
    while (i < 5)  
        printf ("%d\n", *(p + i++));  
    return 0;  
}
```

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



2 Einführung in C

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, i = 0;
```

```
    while (i < 5)
```

```
        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 Einführung in C

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, i = 0;  
    while (i < 5)  
        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 Einführung in C


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, i = 0;  
    while (i < 5)  
        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

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 = 0;  
    while (i < 5)  
        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 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**.

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**.

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**.

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.

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

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

Aufgaben

Aufgabe 1: Multiplikationstabelle

Geben Sie mit Hilfe einer Schleife ein „Einmaleins“ aus:

1 * 7 = 7

2 * 7 = 14

...

10 * 7 = 70

Hinweis: Verwenden Sie Formatspezifikationen wie z. B. `%3d`
(siehe die Dokumentation zu `printf()`)

Aufgabe 2: Fibonacci-Zahlen berechnen

1. Zahl: 0

2. Zahl: 1

nächste Zahl = Summe der beiden vorherigen

Aufgabe 3: Schaltjahr ermitteln

Jahreszahl erfragen

Wenn die Zahl durch 4 teilbar ist, ist es ein Schaltjahr.

Wenn die Zahl durch 100 teilbar ist, ist es kein Schaltjahr.

Wenn die Zahl durch 400 teilbar ist,
ist es doch wieder ein Schaltjahr.

Angewandte Informatik

Prof. Dr. rer. nat. Peter Gerwinski

8. Oktober 2015

Angewandte Informatik

1 Einführung

1.1 Was ist angewandte Informatik?

1.2 Programmierung in C

2 Einführung in C

2.1 Hello, world!

2.2 Programme compilieren und ausführen

2.3 Zahlenwerte ausgeben

2.4 Elementares Rechnen

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

...

1 Einführung

Was ist C?

Etabliertes Profi-Werkzeug

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

„High-Level-Assembler“

- kein „Fallschirm“
- kompakte Schreibweise

Unix-Hintergrund

- Baukastenprinzip
- konsequente Regeln
- kein „Fallschirm“

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

2.2 Programme compilieren und ausführen

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

2 Einführung in C

2.2 Programme compilieren und ausführen

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

2 Einführung in C

2.3 Zahlenwerte ausgeben

Wert ausgeben

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

```
int main (void)
```

```
{
```

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

```
    return 0;
```

```
}
```

2 Einführung in C

2.4 Elementares Rechnen

Wert einlesen

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

```
int main (void)
```

```
{
```

```
    double a;
```

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

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

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

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

```
    return 0;
```

```
}
```

2 Einführung in C

2.4 Elementares Rechnen

Wert an Variable zuweisen

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

```
int main (void)
```

```
{
```

```
    int a;
```

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

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

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

```
    return 0;
```

```
}
```

2 Einführung in C

2.5 Verzweigungen

if-Verzweigung

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


2 Einführung in C

2.6 Schleifen

while-Schleife

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

2 Einführung in C

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 Einführung in C

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 Einführung in C

2.7 Seiteneffekte

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

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

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)
```

```
{
```

```
    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;  
}
```

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;  
}
```

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

```
$ ./side-effects-1
```

```
42
```

```
3
```

```
$
```


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.

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)

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: `=` `+=` `-=` `*=` `/=` `%=`

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

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`

```
int i;
```

```
i = 0;
```

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

```
{
```

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

```
    i += 1;
```

```
}
```

Binäre Operatoren:

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

rot = mit Seiteneffekt

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 += 1;
```

```
}
```

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

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

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 += 1;
```

```
}
```

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

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

```
i = 0;
```

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

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


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 += 1;
```

```
}
```

```
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

```
int i;
```

```
i = 0;
```

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

```
{
```

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

```
    i += 1;
```

```
}
```

```
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

2.8 Strukturierte Programmierung

```
int i;
```

```
i = 0;
```

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

```
{
```

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

```
    i += 1;
```

```
}
```

```
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

2.8 Strukturierte Programmierung

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

```
int i;
```

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

```
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

2.8 Strukturierte Programmierung

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

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

```
int i;

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

```
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

2.8 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:
```

```
int i;  
  
i = 0;           gut  
while (i < 10)  
{  
    printf ("%d\n", i);  
    i += 1;  
}
```

```
for (i = 0; i < 10; i++)  
    printf ("%d\n", i);           nur, wenn  
                                   Sie wissen,  
                                   was Sie tun  
  
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>
```

```
int answer (void)
```

```
{  
    return 42;  
}
```

```
void foo (void)
```

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

```
int main (void)
```

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

2 Einführung in C

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 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;
}
```

- Funktionsdeklaration:
Typ Name (Parameterliste)
{
 Anweisungen
}

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;
}
```

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

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;
}
```

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

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;
}
```

- 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;
}
```

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

2 Einführung in C

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 Einführung in C

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 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;
}
```

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 Zeiger zeigt auf eine Variable.

2 Einführung in C

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

2 Einführung in C

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, i = 0;  
    while (i < 5)  
        printf ("%d\n", *(p + i++));  
    return 0;  
}
```

2 Einführung in C

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, i = 0;
```

```
    while (i < 5)
```

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

```
    return 0;
```

```
}
```

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

2 Einführung in C

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, i = 0;
```

```
    while (i < 5)
```

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

```
    return 0;
```

```
}
```

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

2 Einführung in C

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, i = 0;
```

```
    while (i < 5)
```

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

```
    return 0;
```

```
}
```

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

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



2 Einführung in C

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, i = 0;  
    while (i < 5)  
        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 Einführung in C

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, i = 0;  
    while (i < 5)  
        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 Einführung in C


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, i = 0;  
    while (i < 5)  
        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

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 = 0;  
    while (i < 5)  
        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 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**.

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**.

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**.

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.

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

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

Angewandte Informatik

Prof. Dr. rer. nat. Peter Gerwinski

15. Oktober 2015

Angewandte Informatik

1 Einführung

2 Einführung in C

2.1 Hello, world!

2.2 Programme compilieren und ausführen

2.3 Zahlenwerte ausgeben

2.4 Elementares Rechnen

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

6 Ergänzungen und Ausblicke

2 Einführung in C

2.5 Verzweigungen

if-Verzweigung

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

```
if (b != 0)
    printf ("%d\n", a / b);
else
    printf ("Bitte_nicht_durch_0_dividieren.\n");
```

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

2 Einführung in C

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 Einführung in C

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 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)
```

```
{
```

```
    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;  
}
```

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

```
$ ./side-effects-1
```

```
42
```

```
3
```

```
$
```

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

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 += 1;
}
```

```
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

```
int i;
```

```
i = 0;
```

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

```
{
```

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

```
    i += 1;
```

```
}
```

```
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

2.8 Strukturierte Programmierung

```
int i;
```

```
i = 0;
```

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

```
{
```

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

```
    i += 1;
```

```
}
```

```
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

2.8 Strukturierte Programmierung

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

```
int i;

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

```
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

2.8 Strukturierte Programmierung

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

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

```
int i;

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

```
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

2.8 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:
```

```
int i;  
  
i = 0;          gut  
while (i < 10)  
{  
    printf ("%d\n", i);  
    i += 1;  
}
```

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

nur, wenn
Sie wissen,
was Sie tun

```
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>
```

```
int answer (void)
```

```
{
```

```
    return 42;
```

```
}
```

```
void foo (void)
```

```
{
```

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

```
}
```

```
int main (void)
```

```
{
```

```
    foo ();
```

```
    return 0;
```

```
}
```

2 Einführung in C

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 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;
}
```

- Funktionsdeklaration:
Typ Name (Parameterliste)
{
 Anweisungen
}

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;
}
```

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

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;
}
```

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

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;
}
```

- 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;
}
```

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

2 Einführung in C

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 Einführung in C

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 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;
}
```

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 Zeiger zeigt auf eine Variable.

2 Einführung in C

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

2 Einführung in C

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, i = 0;
    while (i < 5)
        printf ("%d\n", *(p + i++));
    return 0;
}
```

2 Einführung in C

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, i = 0;
```

```
    while (i < 5)
```

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

```
    return 0;
```

```
}
```

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

2 Einführung in C

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, i = 0;
```

```
    while (i < 5)
```

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

```
    return 0;
```

```
}
```

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

2 Einführung in C

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, i = 0;
```

```
    while (i < 5)
```

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

```
    return 0;
```

```
}
```

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

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



2 Einführung in C

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, i = 0;  
    while (i < 5)  
        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 Einführung in C


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, i = 0;  
    while (i < 5)  
        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 Einführung in C


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, i = 0;  
    while (i < 5)  
        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

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 = 0;  
    while (i < 5)  
        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 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**.

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**.

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**.

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.

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

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

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

Angewandte Informatik

Prof. Dr. rer. nat. Peter Gerwinski

29. Oktober 2015

Angewandte Informatik

1 Einführung

2 Einführung in C

...

2.11 Arrays und Strings

2.12 Strukturen

3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliothek verwenden (Beispiel: OpenGL)

3.4 Projekt organisieren: make

4 Algorithmen

5 Hardwarenahe Programmierung

...

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

Angewandte Informatik

1 Einführung

2 Einführung in C

3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliothek verwenden (Beispiel: OpenGL)

3.4 Projekt organisieren: make

4 Algorithmen

5 Hardwarenahe Programmierung

...

Angewandte Informatik

Prof. Dr. rer. nat. Peter Gerwinski

5. November 2015

Angewandte Informatik

1 Einführung

2 Einführung in C

3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliothek verwenden (Beispiel: OpenGL)

3.4 Projekt organisieren: make

4 Algorithmen

4.1 Differentialgleichungen

4.2 Rekursion

4.3 Stack und FIFO

4.4 Aufwandsabschätzungen

5 Hardwarenahe Programmierung

...

3 Bibliotheken

3.1 Der Präprozessor

#include: Text einbinden

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

#define VIER 4: Text ersetzen lassen – Konstante definieren

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

3 Bibliotheken

3.2 Bibliotheken einbinden

Inhalt der Header-Datei: externe Deklarationen

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

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

Funktion wird „anderswo“ definiert

- separater C-Quelltext: mit an `gcc` übergeben
- 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)
```

```
{
```

```
    ...
```

```
}
```

```
...
```

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

Angewandte Informatik

1 Einführung

2 Einführung in C

3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliothek verwenden (Beispiel: OpenGL)

3.4 Projekt organisieren: make

4 Algorithmen

4.1 Differentialgleichungen

4.2 Rekursion

4.3 Stack und FIFO

4.4 Aufwandsabschätzungen

5 Hardwarenahe Programmierung

...

4 Algorithmen

4.1 Differentialgleichungen

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

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

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

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

Praktikumsaufgabe: Basketball

4.2 Rekursion

Vollständige Induktion:

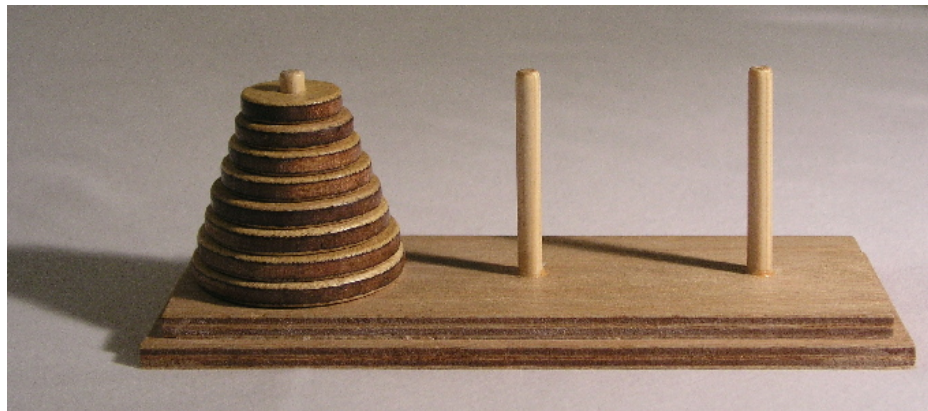
Aussage gilt für $n = 1$
Schluß von $n - 1$ auf n } Aussage gilt für alle $n \in \mathbb{N}$

4.2 Rekursion

Vollständige Induktion:

Aussage gilt für $n = 1$
Schluß von $n - 1$ auf n } Aussage gilt für alle $n \in \mathbb{N}$

Türme von Hanoi

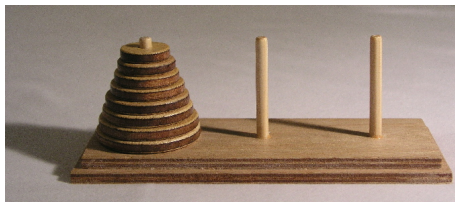


4.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{Aussage gilt für alle } n \in \mathbb{N}$

Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.

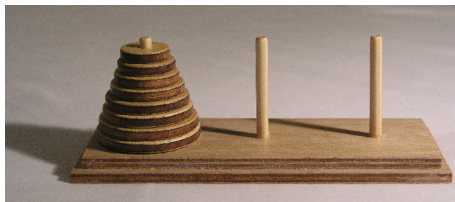


4.2 Rekursion

Vollständige Induktion:
$$\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{Aussage gilt für alle } n \in \mathbb{N}$$

Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.
- $n = 1$ Scheibe: fertig
- Wenn $n - 1$ Scheiben verschiebbar: schiebe $n - 1$ Scheiben auf Hilfsplatz, verschiebe die darunterliegende, hole $n - 1$ Scheiben von Hilfsplatz



4.2 Rekursion

Vollständige Induktion:
$$\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$$

Türme von Hanoi

- 64 Scheiben, 3 Plätze,
immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben
auf größeren liegen.
- $n = 1$ Scheibe: fertig
- Wenn $n - 1$ Scheiben verschiebbar:
schiebe $n - 1$ Scheiben auf Hilfsplatz,
verschiebe die darunterliegende,
hole $n - 1$ Scheiben von Hilfsplatz

```
void verschiebe (int n, int start, int ziel)
{
    if (n == 1)
        verschiebe_1_scheibe (start, ziel);
    else
    {
        verschiebe (1, start, hilfsplatz);
        verschiebe (n - 1, start, ziel);
        verschiebe (1, hilfsplatz, ziel);
    }
}
```

4.2 Rekursion

Floodfill

```
void fill (int x, int y, char c, char o)
{
    if (get_point (x, y) == o)
    {
        put_point (x, y, c);
        fill (x + 1, y, c, o);
        fill (x - 1, y, c, o);
        fill (x, y + 1, c, o);
        fill (x, y - 1, c, o);
    }
}
```

4.2 Rekursion

Floodfill

```
void fill (int x, int y, char c, char o)
{
    if (get_point (x, y) == o)
    {
        put_point (x, y, c);
        fill (x + 1, y, c, o);
        fill (x - 1, y, c, o);
        fill (x, y + 1, c, o);
        fill (x, y - 1, c, o);
    }
}
```

Aufgabe: Schreiben Sie eine Bibliothek für „Text-Grafik“ mit folgenden Funktionen:

- **void clear (char c)**
Bildschirm auf Zeichen **c** löschen
- **void put_point (int x, int y, char c)**
Punkt setzen
- **char get_point (int x, int y)**
Punkt lesen
- **void fill (int x, int y, char c, char o)**
Fläche in der „Farbe“ **o**, die den Punkt **(x, y)** enthält, mit der „Farbe“ **c** ausmalen
- **void display (void)**
Inhalt des Arrays auf dem Bildschirm ausgeben

Hinweis: Verwenden Sie ein Array als „Bildschirm“.

Angewandte Informatik

1 Einführung

2 Einführung in C

3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliothek verwenden (Beispiel: OpenGL)

3.4 Projekt organisieren: make

4 Algorithmen

4.1 Differentialgleichungen

4.2 Rekursion

4.3 Stack und FIFO

4.4 Aufwandsabschätzungen

5 Hardwarenahe Programmierung

...

Angewandte Informatik

Prof. Dr. rer. nat. Peter Gerwinski

12. November 2015

Sie können diese Vortragsfolien einschließlich Beispielprogramme, Skript und sonstiger Lehrmaterialien unter

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

herunterladen.

- **Files**
einzelne Dateien herunterladen
- **Download zip/tar.gz/tar.bz2/tar**
als Archiv herunterladen
- <https://gitlab.cvh-server.de/pgerwinski/ainf.git>
mit GIT herunterladen und synchronisieren

Angewandte Informatik

1 Einführung

2 Einführung in C

3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliothek verwenden (Beispiel: OpenGL)

3.4 Projekt organisieren: make

4 Algorithmen

4.1 Differentialgleichungen

4.2 Rekursion

4.3 Stack und FIFO

4.4 Aufwandsabschätzungen

5 Hardwarenahe Programmierung

...

3.4 Projekt organisieren: make

- Regeln
- Makros

3.4 Projekt organisieren: make

- Regeln

```
earth-6: earth-6.c opengl-magic.h opengl-magic-test.c \  
    textured-spheres.h textured-spheres.c  
gcc -Wall earth-6.c \  
    -lGL -lGLU -lglut opengl-magic-test.c textured-spheres.c \  
    -o earth-6
```

- Makros

3.4 Projekt organisieren: make

- Regeln
- Makros

SOURCES = opengl-magic-test.c textured-spheres.c

INCLUDES = opengl-magic.h textured-spheres.h

LIBS = -lGL -lGLU -lglut

earth-5: earth-5.c \$(SOURCES) \$(INCLUDES)

gcc -Wall earth-5.c \$(SOURCES) \$(LIBS) -o earth-5

earth-6: earth-6.c \$(SOURCES) \$(INCLUDES)

gcc -Wall earth-6.c \$(SOURCES) \$(LIBS) -o earth-6

3.4 Projekt organisieren: make

- Regeln
- Makros

SOURCES = opengl-magic.c textured-spheres.c

INCLUDES = opengl-magic.h textured-spheres.h

LIBS = -IGL -IGLU -lglut

%.c \$(SOURCES) \$(INCLUDES)

gcc -Wall \$< \$(SOURCES) \$(LIBS) -o \$@

3.4 Projekt organisieren: make

- Regeln
- Makros

SOURCES = opengl-magic.c textured-spheres.c

INCLUDES = opengl-magic.h textured-spheres.h

LIBS = -IGL -IGLU -lglut

%.c \$(SOURCES) \$(INCLUDES)

gcc -Wall \$< \$(SOURCES) \$(LIBS) -o \$@

→ 3 Sprachen: C, Präprozessor, make

Angewandte Informatik

1 Einführung

2 Einführung in C

3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliothek verwenden (Beispiel: OpenGL)

3.4 Projekt organisieren: make

4 Algorithmen

4.1 Differentialgleichungen

4.2 Rekursion

4.3 Stack und FIFO

4.4 Aufwandsabschätzungen

5 Hardwarenahe Programmierung

...

4 Algorithmen

4.1 Differentialgleichungen

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

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

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

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

Praktikumsaufgabe: Basketball

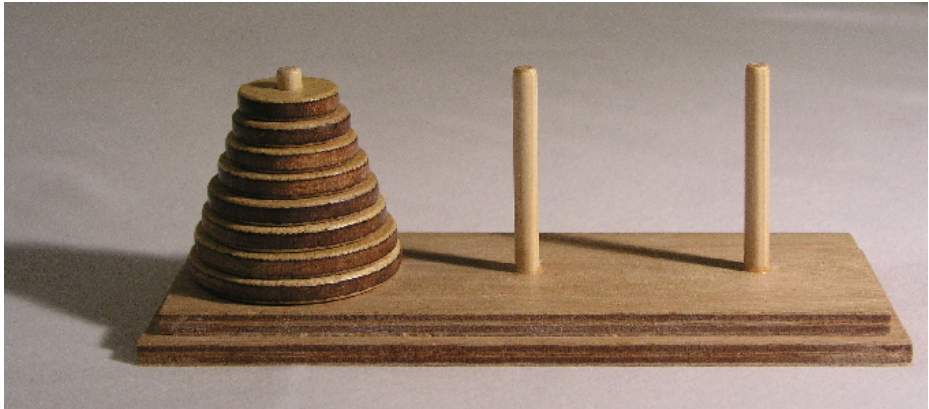
4.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

4.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

Türme von Hanoi

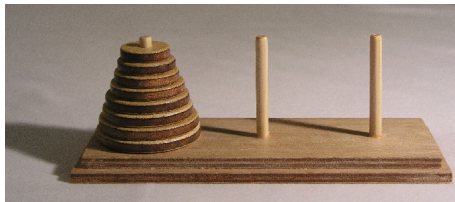


4.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{Aussage gilt für alle } n \in \mathbb{N}$

Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.

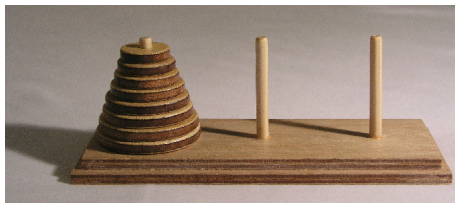


4.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.
- $n = 1$ Scheibe: fertig
- Wenn $n - 1$ Scheiben verschiebbar: schiebe $n - 1$ Scheiben auf Hilfsplatz, verschiebe die darunterliegende, hole $n - 1$ Scheiben von Hilfsplatz



4.2 Rekursion

Vollständige Induktion:
$$\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$$

Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.
- $n = 1$ Scheibe: fertig
- Wenn $n - 1$ Scheiben verschiebbar: schiebe $n - 1$ Scheiben auf Hilfsplatz, verschiebe die darunterliegende, hole $n - 1$ Scheiben von Hilfsplatz

```
void verschiebe (int n, int start, int ziel)
{
    if (n == 1)
        verschiebe_1_scheibe (start, ziel);
    else
    {
        verschiebe (1, start, hilfsplatz);
        verschiebe (n - 1, start, ziel);
        verschiebe (1, hilfsplatz, ziel);
    }
}
```

4.2 Rekursion

Floodfill

```
void fill (int x, int y, char c, char o)
{
    if (get_point (x, y) == o)
    {
        put_point (x, y, c);
        fill (x + 1, y, c, o);
        fill (x - 1, y, c, o);
        fill (x, y + 1, c, o);
        fill (x, y - 1, c, o);
    }
}
```

4.2 Rekursion

Floodfill

```
void fill (int x, int y, char c, char o)
{
    if (get_point (x, y) == o)
    {
        put_point (x, y, c);
        fill (x + 1, y, c, o);
        fill (x - 1, y, c, o);
        fill (x, y + 1, c, o);
        fill (x, y - 1, c, o);
    }
}
```

Aufgabe: Schreiben Sie eine Bibliothek für „Text-Grafik“ mit folgenden Funktionen:

- **void clear (char c)**
Bildschirm auf Zeichen **c** löschen
- **void put_point (int x, int y, char c)**
Punkt setzen
- **char get_point (int x, int y)**
Punkt lesen
- **void fill (int x, int y, char c, char o)**
Fläche in der „Farbe“ **o**, die den Punkt **(x, y)** enthält, mit der „Farbe“ **c** ausmalen
- **void display (void)**
Inhalt des Arrays auf dem Bildschirm ausgeben

Hinweis: Verwenden Sie ein Array als „Bildschirm“.

Angewandte Informatik

1 Einführung

2 Einführung in C

3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliothek verwenden (Beispiel: OpenGL)

3.4 Projekt organisieren: make

4 Algorithmen

4.1 Differentialgleichungen

4.2 Rekursion

4.3 Stack und FIFO

4.4 Aufwandsabschätzungen

5 Hardwarenahe Programmierung

...

Angewandte Informatik

Prof. Dr. rer. nat. Peter Gerwinski

19. November 2015

Sie können diese Vortragsfolien einschließlich Beispielprogramme, Skript und sonstiger Lehrmaterialien unter

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

herunterladen.

- **Files**
einzelne Dateien herunterladen
- **Download zip/tar.gz/tar.bz2/tar**
als Archiv herunterladen
- <https://gitlab.cvh-server.de/pgerwinski/ainf.git>
mit GIT herunterladen und synchronisieren

Sie können diese Vortragsfolien einschließlich Beispielprogramme, Skript und sonstiger Lehrmaterialien unter

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

herunterladen.

Achtung: Einige Dateien sind *Symlinks* – symbolische Verknüpfungen!

- Verweis auf eine andere Datei
- hier: Verweis auf gemeinsame Datei im Verzeichnis [common](#)
- Web-Interface zeigt Dateinamen als Text (statt Dateiinhalt)
- Beispiel: [earth-texture.rgb](#)
 - Textdatei mit Inhalt „[../common/earth-texture.rgb](#)“
 - bei Verwendung: keine Fehlermeldung, aber keine Textur
- Beim Herunterladen als Archiv-Datei (zip/tar.gz/tar.bz2/tar) sind die Symlinks darin korrekt gespeichert.
- ebenfalls korrekt:
[git clone https://gitlab.cvh-server.de/pgerwinski/ainf.git](#)

Ergänzungen

OpenGL

- **Doppelte Pufferung**

2 „Bildschirme“: einer zum Zeichnen; einer wird angezeigt

`opengl-magic-double.c`, `gluSwapBuffers()`

Ergänzungen

OpenGL

- **Doppelte Pufferung**

2 „Bildschirme“: einer zum Zeichnen; einer wird angezeigt
[opengl-magic-double.c](#), [gluSwapBuffers\(\)](#)

- **Im Display-Handler ([draw\(\)](#)) wirklich nur zeichnen!**

Wir haben nicht unter Kontrolle, wann, wie oft oder ob überhaupt diese Funktion aufgerufen wird.

Ergänzungen

Umgang mit Bibliotheken

- **Separates Compilieren**

```
$ gcc -c -Wall opengl-magic.c
```

```
$ gcc -c -Wall textured-spheres.c
```

```
$ gcc -Wall orbit-x.c -lGL -lGLU -lglut \  
    opengl-magic.o textured-spheres.o -o orbit-x
```

Ergänzungen

Umgang mit Bibliotheken

- **Separates Compilieren**

```
$ gcc -c -Wall opengl-magic.c
$ gcc -c -Wall textured-spheres.c
$ gcc -Wall orbit-x.c -lGL -lGLU -lglut \
    opengl-magic.o textured-spheres.o -o orbit-x
```

- Die **Link-Reihenfolge** kann eine Rolle spielen!

```
$ gcc -Wall orbit-x.c \
    opengl-magic.o textured-spheres.o \
    -lGL -lGLU -lglut -o orbit-x
```

Notfalls:

```
$ gcc -Wall orbit-x.c -lGL -lGLU -lglut \
    opengl-magic.o textured-spheres.o \
    -lGL -lGLU -lglut -o orbit-x
```

Ergänzungen

Parameter des Hauptprogramms

```
int main (int argc, char **argv)
{
    init_opengl (&argc, argv, "Orbit");
    ...
    return 0;
}
```

Ergänzungen

Parameter des Hauptprogramms

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

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


Angewandte Informatik

1 Einführung

2 Einführung in C

3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliothek verwenden (Beispiel: OpenGL)

3.4 Projekt organisieren: make

4 Algorithmen

4.1 Differentialgleichungen

4.2 Rekursion

4.3 Stack und FIFO

4.4 Aufwandsabschätzungen

5 Hardwarenahe Programmierung

...

4 Algorithmen

4.1 Differentialgleichungen

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

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

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

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

Praktikumsaufgabe: Umlaufbahn

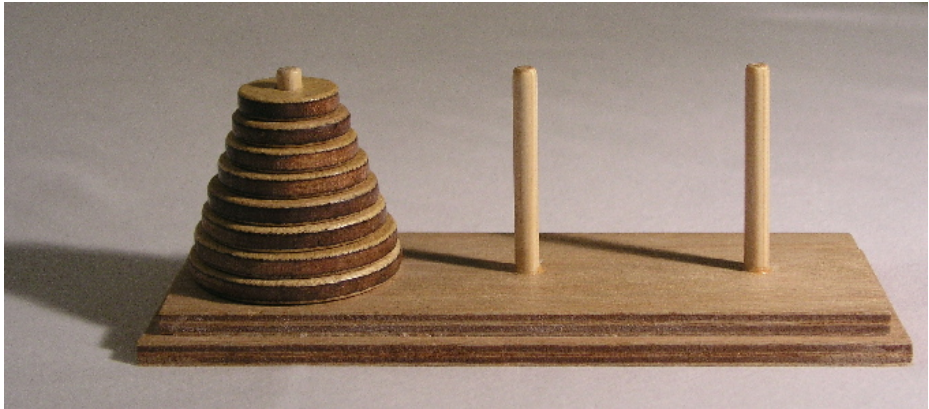
4.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

4.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

Türme von Hanoi

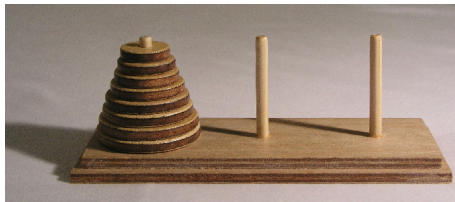


4.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{Aussage gilt für alle } n \in \mathbb{N}$

Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.

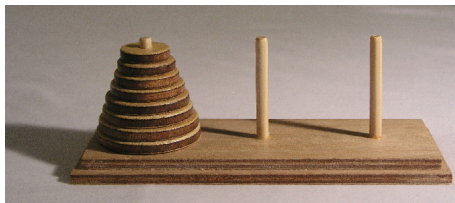


4.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.
- $n = 1$ Scheibe: fertig
- Wenn $n - 1$ Scheiben verschiebbar: schiebe $n - 1$ Scheiben auf Hilfsplatz, verschiebe die darunterliegende, hole $n - 1$ Scheiben von Hilfsplatz



4.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.
- $n = 1$ Scheibe: fertig
- Wenn $n - 1$ Scheiben verschiebbar: schiebe $n - 1$ Scheiben auf Hilfsplatz, verschiebe die darunterliegende, hole $n - 1$ Scheiben von Hilfsplatz

```
void verschiebe (int n, int start, int ziel)
{
    if (n == 1)
        verschiebe_1_scheibe (start, ziel);
    else
    {
        verschiebe (1, start, hilfsplatz);
        verschiebe (n - 1, start, ziel);
        verschiebe (1, hilfsplatz, ziel);
    }
}
```

4.2 Rekursion

Floodfill

```
void fill (int x, int y, char c, char o)
{
    if (get_point (x, y) == o)
    {
        put_point (x, y, c);
        fill (x + 1, y, c, o);
        fill (x - 1, y, c, o);
        fill (x, y + 1, c, o);
        fill (x, y - 1, c, o);
    }
}
```


4.2 Rekursion

Floodfill

```
void fill (int x, int y, char c, char o)
{
    if (get_point (x, y) == o)
    {
        put_point (x, y, c);
        fill (x + 1, y, c, o);
        fill (x - 1, y, c, o);
        fill (x, y + 1, c, o);
        fill (x, y - 1, c, o);
    }
}
```

Aufgabe: Schreiben Sie eine Bibliothek für „Text-Grafik“ mit folgenden Funktionen:

- **void clear (char c)**
Bildschirm auf Zeichen **c** löschen
- **void put_point (int x, int y, char c)**
Punkt setzen
- **char get_point (int x, int y)**
Punkt lesen
- **void fill (int x, int y, char c, char o)**
Fläche in der „Farbe“ **o**,
die den Punkt **(x, y)** enthält,
mit der „Farbe“ **c** ausmalen
- **void display (void)**
Inhalt des Arrays auf dem
Bildschirm ausgeben

Hinweis: Verwenden Sie ein Array als „Bildschirm“.

4.2 Rekursion

Ausgabe einer Hex-Zahl

- Den Rest der Zahl bei Division durch 16 ausgeben, ...
- ... *aber vorher* die Hex-Ziffern der durch 16 dividierten Zahl ausgeben.

4.2 Rekursion

Ausgabe einer Hex-Zahl

- Den Rest der Zahl bei Division durch 16 ausgeben, ...
- ... *aber vorher* die Hex-Ziffern der durch 16 dividierten Zahl ausgeben.

→ Array als Zwischenspeicher zum Umdrehen entfällt.

4.2 Rekursion

Ausgabe einer Hex-Zahl

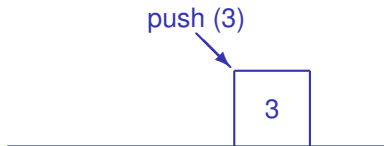
- Den Rest der Zahl bei Division durch 16 ausgeben, ...
- ... *aber vorher* die Hex-Ziffern der durch 16 dividierten Zahl ausgeben.

→ Array als Zwischenspeicher zum Umdrehen entfällt.

→ *Wirklich?*

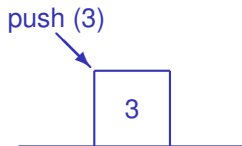
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

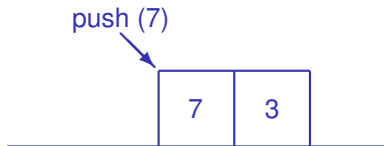
„Last In – First Out“



LIFO = Stack = Stapel

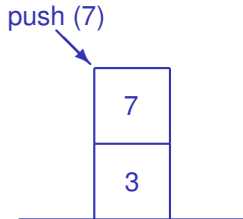
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



LIFO = Stack = Stapel

4.3 Stack und FIFO

„First In – First Out“

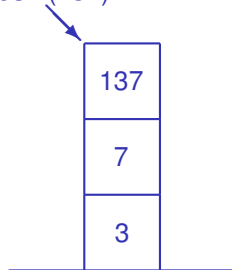
push (137)



FIFO = Queue = Reihe

„Last In – First Out“

push (137)



LIFO = Stack = Stapel

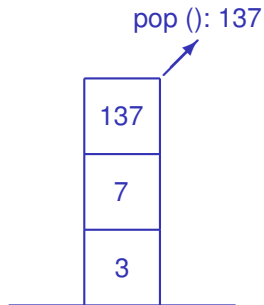
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

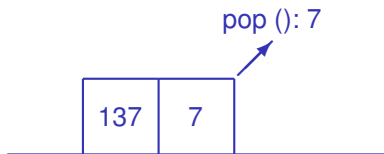
„Last In – First Out“



LIFO = Stack = Stapel

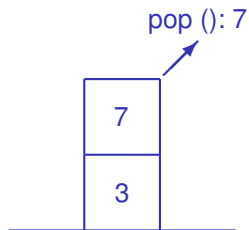
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

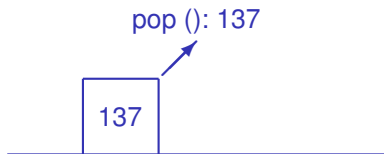
„Last In – First Out“



LIFO = Stack = Stapel

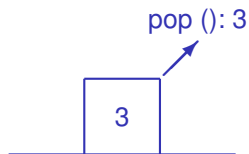
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



LIFO = Stack = Stapel

Angewandte Informatik

1 Einführung

2 Einführung in C

3 Bibliotheken

4 Algorithmen

4.1 Differentialgleichungen

4.2 Rekursion

4.3 Stack und FIFO

4.4 Aufwandsabschätzungen

5 Hardwarenahe Programmierung

...

Angewandte Informatik

Prof. Dr. rer. nat. Peter Gerwinski

26. November 2015

Angewandte Informatik

1 Einführung

2 Einführung in C

3 Bibliotheken

4 Algorithmen

4.1 Differentialgleichungen

4.2 Rekursion

4.3 Stack und FIFO

4.4 Aufwandsabschätzungen

5 Hardwarenahe Programmierung

...

Ergänzungen

OpenGL

- **Doppelte Pufferung**

2 „Bildschirme“: einer zum Zeichnen; einer wird angezeigt
[opengl-magic-double.c](#), [gluSwapBuffers\(\)](#)

- **Im Display-Handler ([draw\(\)](#)) wirklich nur zeichnen!**

Wir haben nicht unter Kontrolle, wann, wie oft oder ob überhaupt diese Funktion aufgerufen wird.

Ergänzungen

Umgang mit Bibliotheken

- **Separates Compilieren**

```
$ gcc -c -Wall opengl-magic.c
$ gcc -c -Wall textured-spheres.c
$ gcc -Wall orbit-x.c -lGL -lGLU -lglut \
    opengl-magic.o textured-spheres.o -o orbit-x
```

- Die **Link-Reihenfolge** kann eine Rolle spielen!

```
$ gcc -Wall orbit-x.c \
    opengl-magic.o textured-spheres.o \
    -lGL -lGLU -lglut -o orbit-x
```

Notfalls:

```
$ gcc -Wall orbit-x.c -lGL -lGLU -lglut \
    opengl-magic.o textured-spheres.o \
    -lGL -lGLU -lglut -o orbit-x
```

Ergänzungen

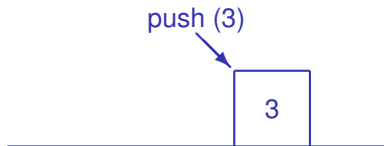
Parameter des Hauptprogramms

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

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

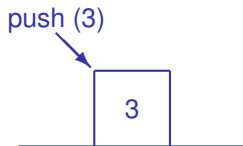

4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

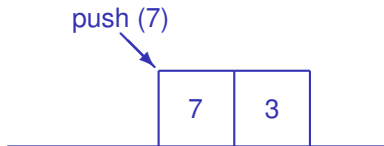
„Last In – First Out“



LIFO = Stack = Stapel

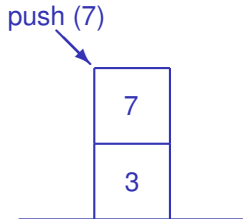
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



LIFO = Stack = Stapel

4.3 Stack und FIFO

„First In – First Out“

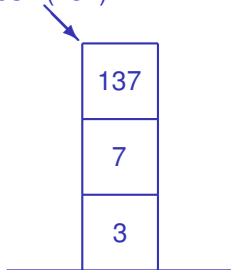
push (137)



FIFO = Queue = Reihe

„Last In – First Out“

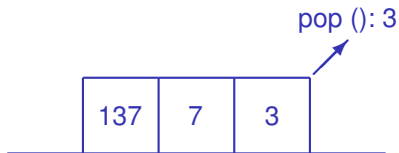
push (137)



LIFO = Stack = Stapel

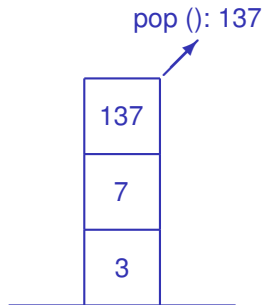
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

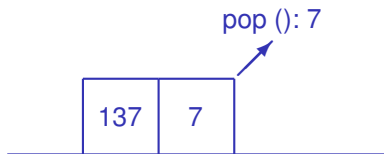
„Last In – First Out“



LIFO = Stack = Stapel

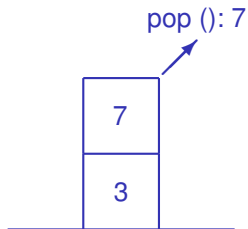
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

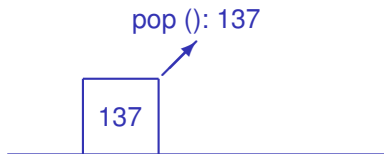
„Last In – First Out“



LIFO = Stack = Stapel

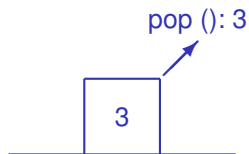
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

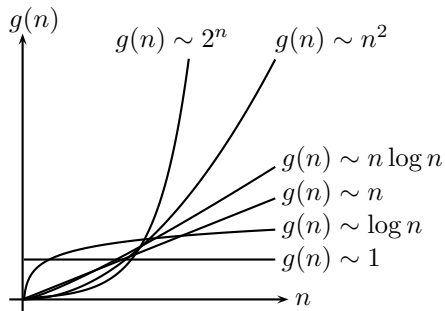


LIFO = Stack = Stapel

4.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen



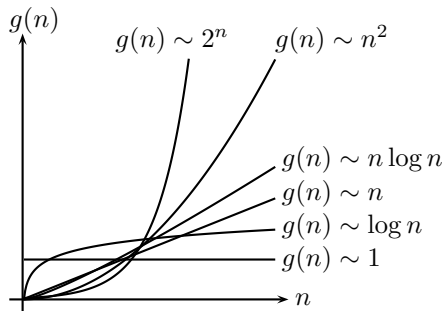
n : Eingabedaten

$g(n)$: Rechenzeit

4.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$



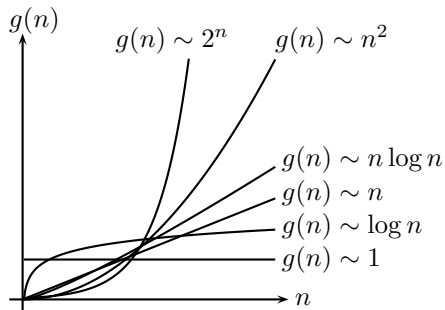
n : Eingabedaten

$g(n)$: Rechenzeit

4.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort



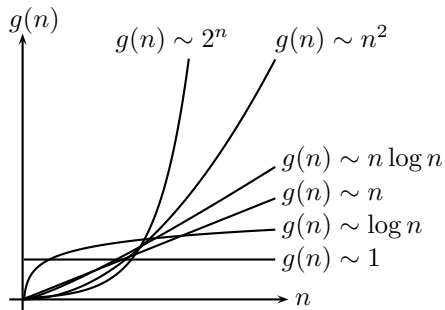
n : Eingabedaten

$g(n)$: Rechenzeit

4.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$



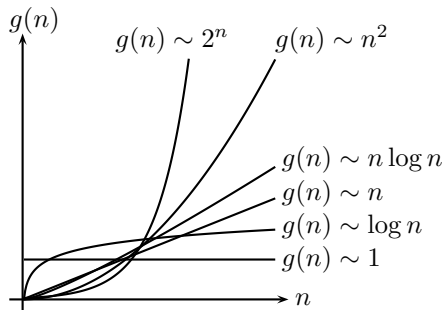
n : Eingabedaten

$g(n)$: Rechenzeit

4.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen,
abbrechen, falls schon sortiert
→ Bubblesort



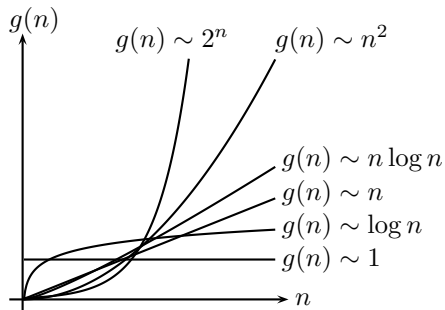
n : Eingabedaten

$g(n)$: Rechenzeit

4.4 Aufwandsabschätzungen

Beispiel: Sortialgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen, abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$



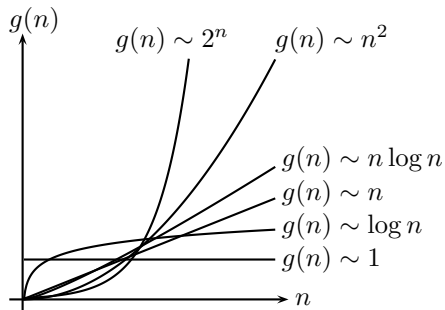
n : Eingabedaten

$g(n)$: Rechenzeit

4.4 Aufwandsabschätzungen

Beispiel: Sortialgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen, abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$
- Rekursiv sortieren
→ Quicksort



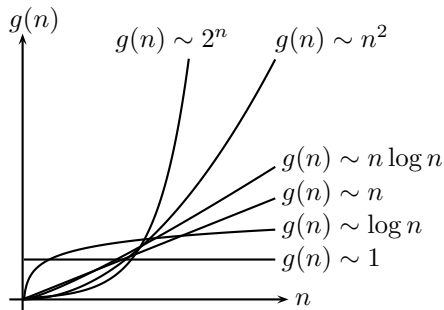
n : Eingabedaten

$g(n)$: Rechenzeit

4.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen, abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$
- Rekursiv sortieren
→ Quicksort: $\mathcal{O}(n \log n)$ bis $\mathcal{O}(n^2)$



n : Eingabedaten

$g(n)$: Rechenzeit

Angewandte Informatik

- 1 Einführung**
- 2 Einführung in C**
- 3 Bibliotheken**
- 4 Algorithmen**
 - 4.1** Differentialgleichungen
 - 4.2** Rekursion
 - 4.3** Stack und FIFO
 - 4.4** Aufwandsabschätzungen
- 5 Hardwarenahe Programmierung**
- ...

Angewandte Informatik

Prof. Dr. rer. nat. Peter Gerwinski

3. Dezember 2015

Angewandte Informatik

1 Einführung

2 Einführung in C

3 Bibliotheken

4 Algorithmen

4.1 Differentialgleichungen

4.2 Rekursion

4.3 Stack und FIFO

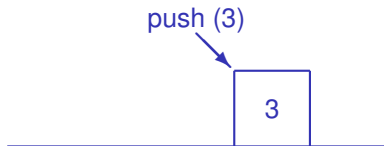
4.4 Aufwandsabschätzungen

5 Hardwarenahe Programmierung

...

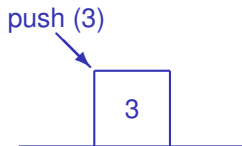
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

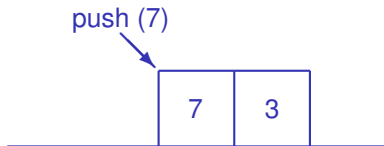
„Last In – First Out“



LIFO = Stack = Stapel

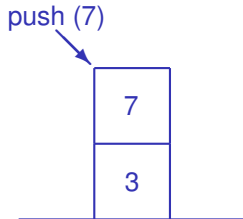
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



LIFO = Stack = Stapel

4.3 Stack und FIFO

„First In – First Out“

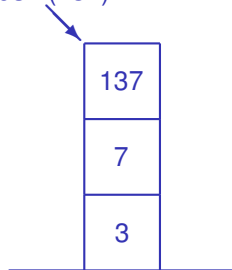
push (137)



FIFO = Queue = Reihe

„Last In – First Out“

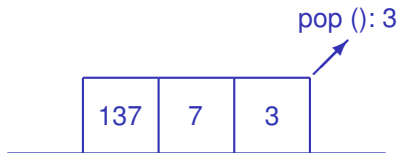
push (137)



LIFO = Stack = Stapel

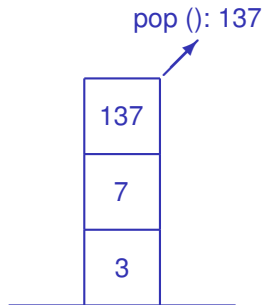
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

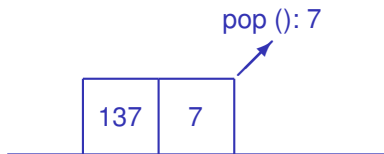
„Last In – First Out“



LIFO = Stack = Stapel

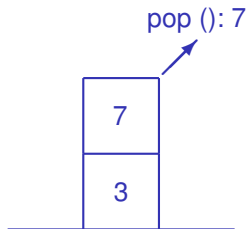
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

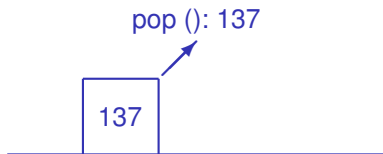
„Last In – First Out“



LIFO = Stack = Stapel

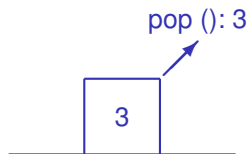
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



LIFO = Stack = Stapel

Angewandte Informatik

1 Einführung

2 Einführung in C

3 Bibliotheken

4 Algorithmen

4.1 Differentialgleichungen

4.2 Rekursion

4.3 Stack und FIFO

4.4 Aufwandsabschätzungen

5 Hardwarenahe Programmierung

5.1 Bit-Operationen

5.2 I/O-Ports

5.3 Interrupts

5.4 volatile-Variable

5.5 Software-Interrupts

...

5 Hardwarenahe Programmierung

5 Hardwarenahe Programmierung

5.1 Bit-Operationen

5 Hardwarenahe Programmierung

5.1 Bit-Operationen

5.1.1 Zahlensysteme

5 Hardwarenahe Programmierung

5.1 Bit-Operationen

5.1.1 Zahlensysteme

Basis		Beispiel
2	Binärsystem	1 0000 0011
8	Oktalsystem	0403
10	Dezimalsystem	259
16	Hexadezimalsystem	0x103
256	IP-Adressen (IPv4)	0.0.1.3

5 Hardwarenahe Programmierung

5.1 Bit-Operationen

5.1.1 Zahlensysteme

Oktal- und Hexadezimal-Zahlen lassen sich ziffernweise in Binär-Zahlen umrechnen:

000	0	0000	0	1000	8
001	1	0001	1	1001	9
010	2	0010	2	1010	A
011	3	0011	3	1011	B
100	4	0100	4	1100	C
101	5	0101	5	1101	D
110	6	0110	6	1110	E
111	7	0111	7	1111	F

5.1.2 Bit-Operationen in C

C-Operator	Verknüpfung	Anwendung
&	Und	Bits gezielt löschen
	Oder	Bits gezielt setzen
^	Exklusiv-Oder	Bits gezielt invertieren
~	Nicht	Alle Bits invertieren
<<	Verschiebung nach links	Maske generieren
>>	Verschiebung nach rechts	Bits isolieren

Aufgabe: Schreiben Sie C-Funktionen, die ein „Array von Bits“ realisieren, z. B.

void set_bit (int i);	Bei Index <i>i</i> auf 1 setzen
void clear_bit (int i);	Bei Index <i>i</i> auf 0 setzen
int get_bit (int i);	Bei Index <i>i</i> lesen

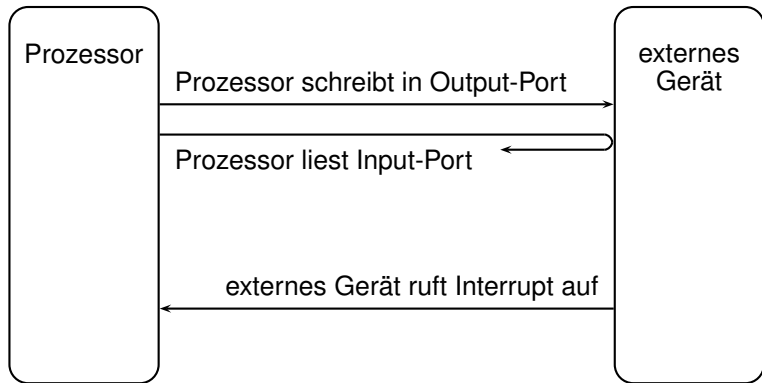
Hinweise:

- Die Größe des Bit-„Arrays“ (z. B. 1000) dürfen Sie als *vorher bekannt* voraussetzen.
- Sie benötigen ein Array, z. B. von **char**- oder **int**-Variablen.
- Sie benötigen eine Division (/) sowie den Divisionsrest (Modulo: %).

5.2 I/O-Ports

5.3 Interrupts

Kommunikation mit externen Geräten



5.2 I/O-Ports

In Output-Port schreiben = Leitungen ansteuern

Datei: `RP6Base/RP6Base_Examples/RP6Examples_20080915/
RP6Lib/RP6base/RP6RobotBaseLib.c`

Suchbegriff: `setMotorDir`

```
void setMotorDir(uint8_t left_dir, uint8_t right_dir)
```

```
{
```

```
    /* ... */
```

```
    if(left_dir)
```

```
        PORTC |= DIR_L;
```

```
    else
```

```
        PORTC &= ~DIR_L;
```

```
    if(right_dir)
```

```
        PORTC |= DIR_R;
```

```
    else
```

```
        PORTC &= ~DIR_R;
```

```
}
```

Manipulation einzelner Bits

→ Steuerung der Motordrehrichtung

Output-Port

5.3 Interrupts

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

Datei: [RP6Base/RP6Base_Examples/RP6Examples_20080915/
RP6Lib/RP6base/RP6RobotBaseLib.c](#)
Suchbegriff: ISR

„Dies ist ein Interrupt-Handler.“

Interrupt-Vektor 0 darauf zeigen lassen

Schreibweise herstellerspezifisch!

```
ISR (INT0_vect)
{
    mleft_dist++;
    mleft_counter++;
    /* ... */
}
```

Aufruf durch Sensor an Encoder-Scheibe
→ Entfernungsmessung

5.4 volatile-Variable

```
volatile uint16_t mleft_counter;
```

```
/* ... */
```

„Immer lesen und schreiben. Nicht wegoptimieren.“

```
volatile uint16_t mleft_dist;
```

```
/* ... */
```

```
ISR (INT0_vect)
```

```
{  
    mleft_dist++;  
    mleft_counter++;  
    /* ... */  
}
```

```
int main (void)
```

```
{  
    int prev_mleft_dist = mleft_dist;  
    while (mleft_dist == prev_mleft_dist)  
        /* just wait */;  
}
```

5.5 Software-Interrupts

```
mov ax, 0012
```

```
int 10
```

5.5 Software-Interrupts

`mov ax, 0012` ← Parameter in Prozessorregister

`int 10` ← Funktionsaufruf über Interrupt-Vektor

5.5 Software-Interrupts

`mov ax, 0012` ← Parameter in Prozessorregister
`int 10` ← Funktionsaufruf über Interrupt-Vektor

Beispiel: VGA-Grafikkarte

- Modus setzen: `mov ah, 00`
- Grafikmodus: `mov al, 12`
- Textmodus: `mov al, 03`

5.5 Software-Interrupts

`mov ax, 0012` ← Parameter in Prozessorregister
`int 10` ← Funktionsaufruf über Interrupt-Vektor

Beispiel: VGA-Grafikkarte

- Modus setzen: `mov ah, 00`
- Grafikmodus: `mov al, 12`
- Textmodus: `mov al, 03`

Verschiedene Farben: Output-Ports

- *Graphics Register*: Index `03CE`, Daten `03CF`
- Index 0: *Set/Reset Register*
- Index 1: *Enable Set/Reset Register*
- Index 8: *Bit Mask Register*
- Jedes Bit steht für Schreibzugriff auf eine Speicherbank.
- 4 Speicherbänke → 16 Farben

Angewandte Informatik

1 Einführung

2 Einführung in C

3 Bibliotheken

4 Algorithmen

4.1 Differentialgleichungen

4.2 Rekursion

4.3 Stack und FIFO

4.4 Aufwandsabschätzungen

5 Hardwarenahe Programmierung

5.1 Bit-Operationen

5.2 I/O-Ports

5.3 Interrupts

5.4 volatile-Variable

5.5 Software-Interrupts

...

Angewandte Informatik

Prof. Dr. rer. nat. Peter Gerwinski

10. Dezember 2015

Angewandte Informatik

1 Einführung

2 Einführung in C

3 Bibliotheken

4 Algorithmen

4.1 Differentialgleichungen

4.2 Rekursion

4.3 Stack und FIFO

4.4 Aufwandsabschätzungen

5 Hardwarenahe Programmierung

5.1 Bit-Operationen

5.2 I/O-Ports

5.3 Interrupts

5.4 volatile-Variable

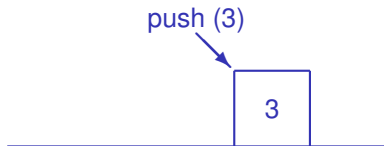
5.5 Software-Interrupts

...

...

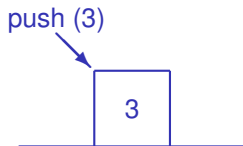
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

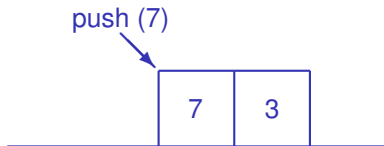
„Last In – First Out“



LIFO = Stack = Stapel

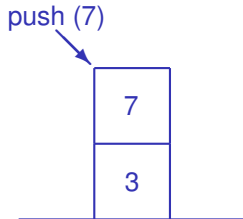
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



LIFO = Stack = Stapel

4.3 Stack und FIFO

„First In – First Out“

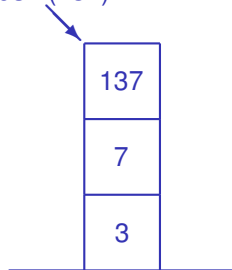
push (137)



FIFO = Queue = Reihe

„Last In – First Out“

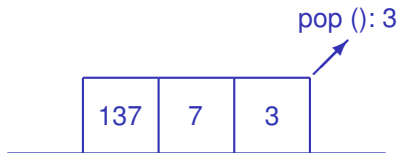
push (137)



LIFO = Stack = Stapel

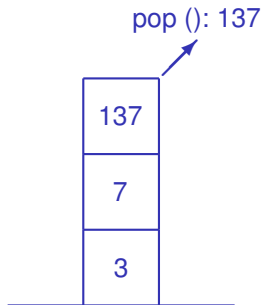
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

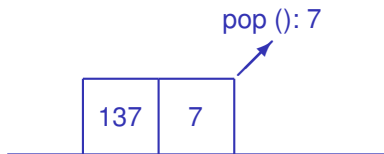
„Last In – First Out“



LIFO = Stack = Stapel

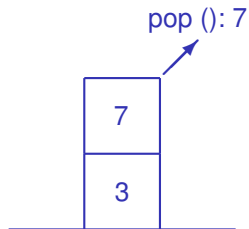
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

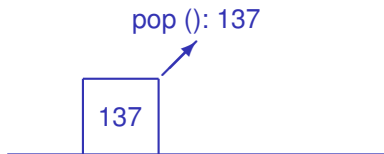
„Last In – First Out“



LIFO = Stack = Stapel

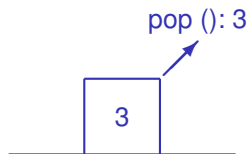
4.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



LIFO = Stack = Stapel

5 Hardwarenahe Programmierung

5.1 Bit-Operationen

5.1.1 Zahlensysteme

Oktal- und Hexadezimal-Zahlen lassen sich ziffernweise in Binär-Zahlen umrechnen:

000	0	0000	0	1000	8
001	1	0001	1	1001	9
010	2	0010	2	1010	A
011	3	0011	3	1011	B
100	4	0100	4	1100	C
101	5	0101	5	1101	D
110	6	0110	6	1110	E
111	7	0111	7	1111	F

Auswendig lernen!

5.1.2 Bit-Operationen in C

C-Operator	Verknüpfung	Anwendung
&	Und	Bits gezielt löschen
	Oder	Bits gezielt setzen
^	Exklusiv-Oder	Bits gezielt invertieren
~	Nicht	Alle Bits invertieren
<<	Verschiebung nach links	Maske generieren
>>	Verschiebung nach rechts	Bits isolieren

Beispiele:

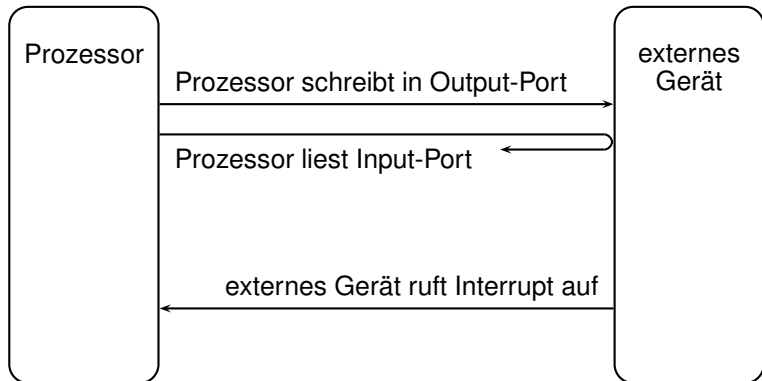
- Bit Nr. 5 gezielt auf 1 setzen: `PORTB |= 1 << 5;`
- Bit Nr. 6 gezielt auf 0 setzen: `PORTA &= ~(1 << 6);`
- Ist Bit Nr. 4 gesetzt? `if (PINC & (1 << 4) ...`
- Umschalten zwischen Ein- und Ausgabe: `DDR`
Bit = 1: Ausgabe; Bit = 0: Eingabe

**Details abhängig von
Prozessor und Compiler!**

5.2 I/O-Ports

5.3 Interrupts

Kommunikation mit externen Geräten



5.2 I/O-Ports

In Output-Port schreiben = Leitungen ansteuern

Datei: `RP6Base/RP6Base_Examples/RP6Examples_20080915/
RP6Lib/RP6base/RP6RobotBaseLib.c`

Suchbegriff: `setMotorDir`

```
void setMotorDir(uint8_t left_dir, uint8_t right_dir)
```

```
{
```

```
    /* ... */
```

```
    if(left_dir)
```

```
        PORTC |= DIR_L;
```

```
    else
```

```
        PORTC &= ~DIR_L;
```

```
    if(right_dir)
```

```
        PORTC |= DIR_R;
```

```
    else
```

```
        PORTC &= ~DIR_R;
```

```
}
```

Manipulation einzelner Bits

→ Steuerung der Motordrehrichtung

Output-Port

5.2 I/O-Ports

In Output-Port schreiben = Leitungen ansteuern

Datei: `RP6Base/RP6Base_Examples/RP6Examples_20080915/
RP6Lib/RP6base/RP6RobotBaseLib.c`

Suchbegriff: `updateStatusLEDs`

```
DDRB &= ~0x83;  ← Data Direction Register: auf Input(!) setzen
PORTB &= ~0x83; ← internen Pull-Up-Widerstand ausschalten
if(statusLEDs.LED4){ DDRB |= SL4; PORTB |= SL4; }
if(statusLEDs.LED5){ DDRB |= SL5; PORTB |= SL5; }
if(statusLEDs.LED6){ DDRB |= SL6; PORTB |= SL6; }
DDRC &= ~0x70;
PORTC &= ~0x70;
DDRC |= ((statusLEDs.byte << 4) & 0x70);
PORTC |= ((statusLEDs.byte << 4) & 0x70);
```

Manipulation einzelner Bits

3 Bits gemeinsam

union statusLEDs: Bit-Felder vs. Byte

5.3 Interrupts

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

Datei: [RP6Base/RP6Base_Examples/RP6Examples_20080915/
RP6Lib/RP6base/RP6RobotBaseLib.c](#)
Suchbegriff: ISR

„Dies ist ein Interrupt-Handler.“

Interrupt-Vektor 0 darauf zeigen lassen

Schreibweise abhängig von Prozessor und Compiler!

```
ISR (INT0_vect)
{
    mleft_dist++;
    mleft_counter++;
    /* ... */
}
```

Aufruf durch Sensor an Encoder-Scheibe
→ Entfernungsmessung

5.4 volatile-Variable

```
volatile uint16_t mleft_counter;
```

```
/* ... */
```

„Immer lesen und schreiben. Nicht wegoptimieren.“

```
volatile uint16_t mleft_dist;
```

```
/* ... */
```

```
ISR (INT0_vect)
```

```
{  
    mleft_dist++;  
    mleft_counter++;  
    /* ... */  
}
```

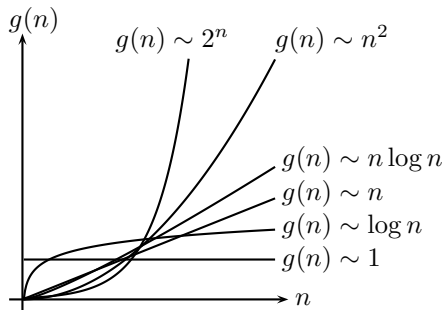
```
int main (void)
```

```
{  
    int prev_mleft_dist = mleft_dist;  
    while (mleft_dist == prev_mleft_dist)  
        /* just wait */;  
}
```

5.5 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen



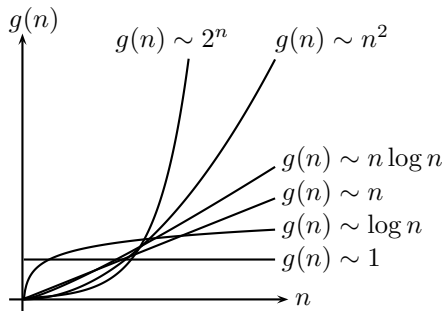
n : Eingabedaten

$g(n)$: Rechenzeit

5.5 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$



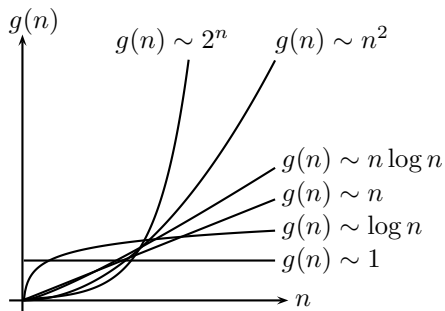
n : Eingabedaten

$g(n)$: Rechenzeit

5.5 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort



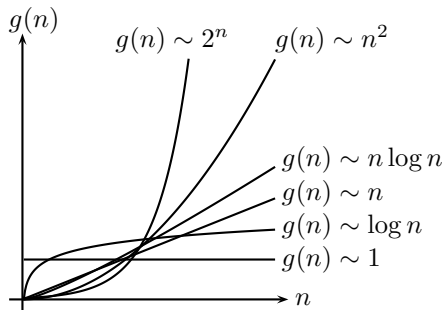
n : Eingabedaten

$g(n)$: Rechenzeit

5.5 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$



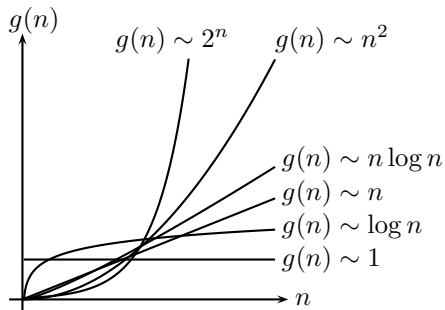
n : Eingabedaten

$g(n)$: Rechenzeit

5.5 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen,
abbrechen, falls schon sortiert
→ Bubblesort



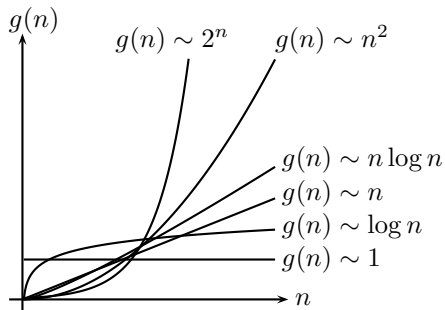
n : Eingabedaten

$g(n)$: Rechenzeit

5.5 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen, abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$



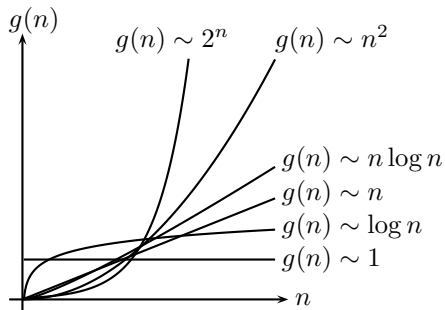
n : Eingabedaten

$g(n)$: Rechenzeit

5.5 Aufwandsabschätzungen

Beispiel: Sortialgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen, abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$
- Rekursiv sortieren
→ Quicksort



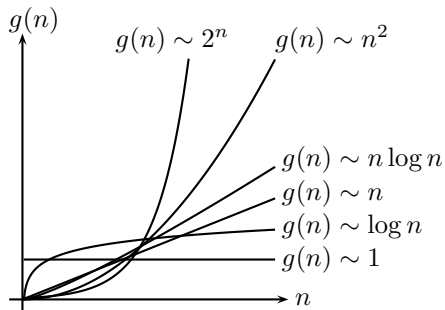
n : Eingabedaten

$g(n)$: Rechenzeit

5.5 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen, abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$
- Rekursiv sortieren
→ Quicksort: $\mathcal{O}(n \log n)$ bis $\mathcal{O}(n^2)$



n : Eingabedaten

$g(n)$: Rechenzeit

Angewandte Informatik

1 Einführung

2 Einführung in C

3 Bibliotheken

4 Algorithmen

4.1 Differentialgleichungen

4.2 Rekursion

4.3 Stack und FIFO

4.4 Aufwandsabschätzungen

5 Hardwarenahe Programmierung

5.1 Bit-Operationen

5.2 I/O-Ports

5.3 Interrupts

5.4 volatile-Variable

5.5 Software-Interrupts

...

...

Angewandte Informatik

Prof. Dr. rer. nat. Peter Gerwinski

17. Dezember 2015

Angewandte Informatik

1 Einführung

2 Einführung in C

3 Bibliotheken

4 Algorithmen

4.1 Differentialgleichungen

4.2 Rekursion

4.3 Stack und FIFO

4.4 Aufwandsabschätzungen

5 Hardwarenahe Programmierung

5.1 Bit-Operationen

5.2 I/O-Ports

5.3 Interrupts

5.4 volatile-Variable

5.5 Software-Interrupts

...

...

5 Hardwarenahe Programmierung

5.1 Bit-Operationen

5.1.1 Zahlensysteme

Oktal- und Hexadezimal-Zahlen lassen sich ziffernweise
in Binär-Zahlen umrechnen:

000	0	0000	0	1000	8
001	1	0001	1	1001	9
010	2	0010	2	1010	A
011	3	0011	3	1011	B
100	4	0100	4	1100	C
101	5	0101	5	1101	D
110	6	0110	6	1110	E
111	7	0111	7	1111	F

Auswendig lernen!

5.1.2 Bit-Operationen in C

C-Operator	Verknüpfung	Anwendung
&	Und	Bits gezielt löschen
	Oder	Bits gezielt setzen
^	Exklusiv-Oder	Bits gezielt invertieren
~	Nicht	Alle Bits invertieren
<<	Verschiebung nach links	Maske generieren
>>	Verschiebung nach rechts	Bits isolieren

Beispiele:

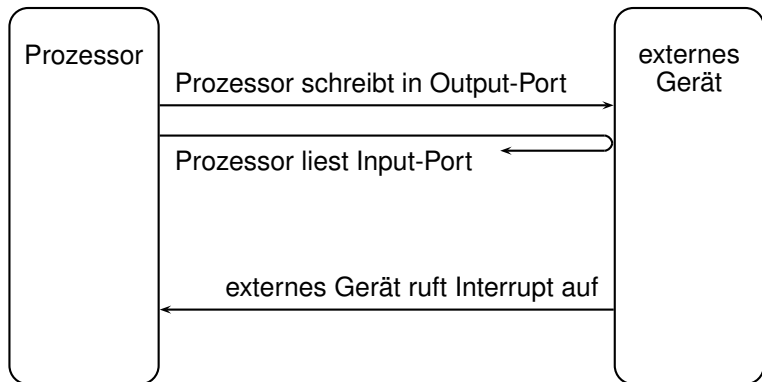
- Bit Nr. 5 gezielt auf 1 setzen: `PORTB |= 1 << 5;`
- Bit Nr. 6 gezielt auf 0 setzen: `PORTA &= ~(1 << 6);`
- Ist Bit Nr. 4 gesetzt? `if (PINC & (1 << 4) ...`
- Umschalten zwischen Ein- und Ausgabe: `DDR`
Bit = 1: Ausgabe; Bit = 0: Eingabe

**Details abhängig von
Prozessor und Compiler!**

5.2 I/O-Ports

5.3 Interrupts

Kommunikation mit externen Geräten



5.2 I/O-Ports

In Output-Port schreiben = Leitungen ansteuern

Datei: `RP6Base/RP6Base_Examples/RP6Examples_20080915/
RP6Lib/RP6base/RP6RobotBaseLib.c`

Suchbegriff: `setMotorDir`

```
void setMotorDir(uint8_t left_dir, uint8_t right_dir)
```

```
{
```

```
    /* ... */
```

```
    if(left_dir)
```

```
        PORTC |= DIR_L;
```

```
    else
```

```
        PORTC &= ~DIR_L;
```

```
    if(right_dir)
```

```
        PORTC |= DIR_R;
```

```
    else
```

```
        PORTC &= ~DIR_R;
```

```
}
```

Manipulation einzelner Bits

→ Steuerung der Motordrehrichtung

Output-Port

5.2 I/O-Ports

In Output-Port schreiben = Leitungen ansteuern

Datei: `RP6Base/RP6Base_Examples/RP6Examples_20080915/
RP6Lib/RP6base/RP6RobotBaseLib.c`

Suchbegriff: `updateStatusLEDs`

```
DDRB &= ~0x83;  ← Data Direction Register: auf Input(!) setzen
PORTB &= ~0x83; ← internen Pull-Up-Widerstand ausschalten
if(statusLEDs.LED4){ DDRB |= SL4; PORTB |= SL4; }
if(statusLEDs.LED5){ DDRB |= SL5; PORTB |= SL5; }
if(statusLEDs.LED6){ DDRB |= SL6; PORTB |= SL6; }
DDRC &= ~0x70;
PORTC &= ~0x70;
DDRC |= ((statusLEDs.byte << 4) & 0x70);
PORTC |= ((statusLEDs.byte << 4) & 0x70);
```

Manipulation einzelner Bits

3 Bits gemeinsam

union statusLEDs: Bit-Felder vs. Byte

5.3 Interrupts

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

Datei: [RP6Base/RP6Base_Examples/RP6Examples_20080915/
RP6Lib/RP6base/RP6RobotBaseLib.c](#)
Suchbegriff: ISR

„Dies ist ein Interrupt-Handler.“

Interrupt-Vektor 0 darauf zeigen lassen

Schreibweise abhängig von Prozessor und Compiler!

```
ISR (INT0_vect)
{
    mleft_dist++;
    mleft_counter++;
    /* ... */
}
```

Aufruf durch Sensor an Encoder-Scheibe
→ Entfernungsmessung

5.4 volatile-Variable

```
volatile uint16_t mleft_counter;
```

```
/* ... */
```

„Immer lesen und schreiben. Nicht wegoptimieren.“

```
volatile uint16_t mleft_dist;
```

```
/* ... */
```

```
ISR (INT0_vect)
```

```
{  
    mleft_dist++;  
    mleft_counter++;  
    /* ... */  
}
```

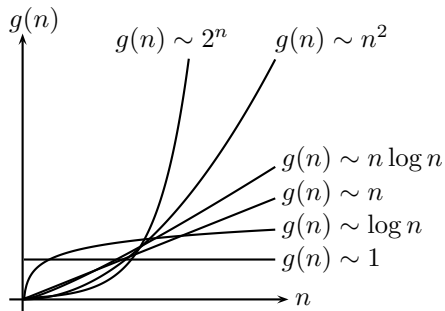
```
int main (void)
```

```
{  
    int prev_mleft_dist = mleft_dist;  
    while (mleft_dist == prev_mleft_dist)  
        /* just wait */;  
}
```


5.5 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen



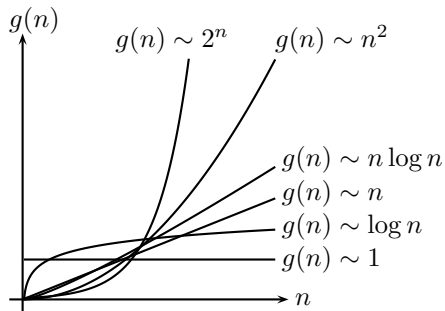
n : Eingabedaten

$g(n)$: Rechenzeit

5.5 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$



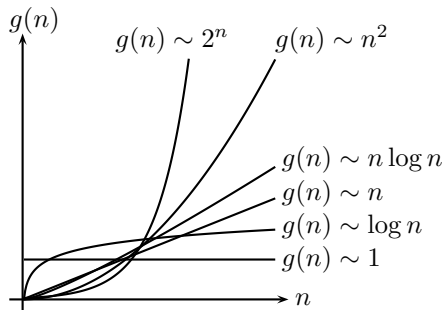
n : Eingabedaten

$g(n)$: Rechenzeit

5.5 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort



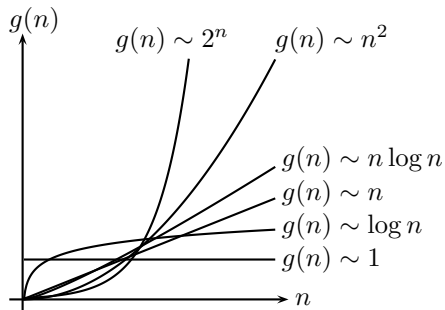
n : Eingabedaten

$g(n)$: Rechenzeit

5.5 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$



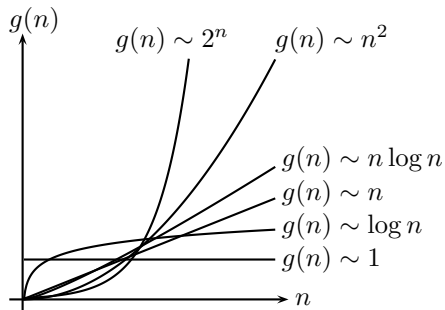
n : Eingabedaten

$g(n)$: Rechenzeit

5.5 Aufwandsabschätzungen

Beispiel: Sortialgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen,
abbrechen, falls schon sortiert
→ Bubblesort



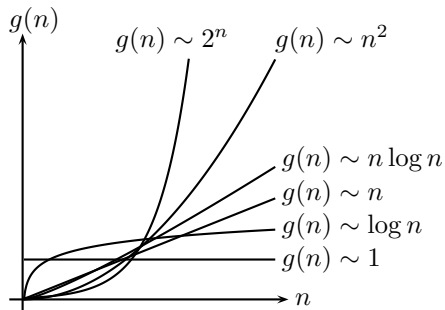
n : Eingabedaten

$g(n)$: Rechenzeit

5.5 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen, abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$



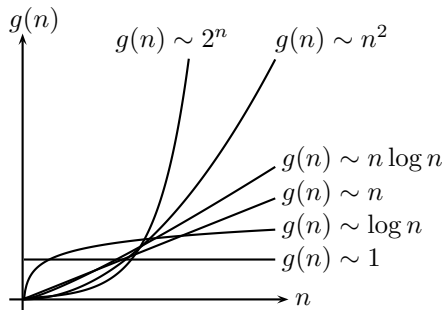
n : Eingabedaten

$g(n)$: Rechenzeit

5.5 Aufwandsabschätzungen

Beispiel: Sortialgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen, abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$
- Rekursiv sortieren
→ Quicksort



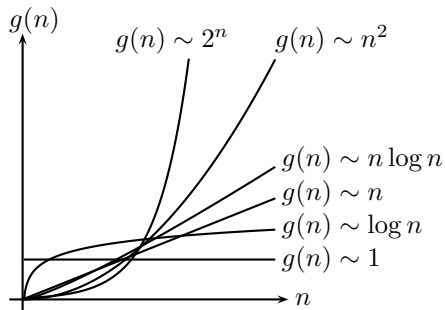
n : Eingabedaten

$g(n)$: Rechenzeit

5.5 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen, abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$
- Rekursiv sortieren
→ Quicksort: $\mathcal{O}(n \log n)$ bis $\mathcal{O}(n^2)$



n : Eingabedaten

$g(n)$: Rechenzeit

Angewandte Informatik

1 Einführung

2 Einführung in C

3 Bibliotheken

4 Algorithmen

4.1 Differentialgleichungen

4.2 Rekursion

4.3 Stack und FIFO

4.4 Aufwandsabschätzungen

5 Hardwarenahe Programmierung

5.1 Bit-Operationen

5.2 I/O-Ports

5.3 Interrupts

5.4 volatile-Variable

5.5 Software-Interrupts

...

...

Angewandte Informatik

Prof. Dr. rer. nat. Peter Gerwinski

7. Januar 2016

Angewandte Informatik

- 1 Einführung**
- 2 Einführung in C**
- 3 Bibliotheken**
- 4 Algorithmen**
 - 4.1 Differentialgleichungen
 - 4.2 Rekursion
 - 4.3 Stack und FIFO
 - 4.4 Aufwandsabschätzungen
 - 4.4 Dynamische Speicherverwaltung
- 5 Hardwarenahe Programmierung**
 - ...
 - 5.4 volatile-Variable
 - 5.5 Software-Interrupts
 - 5.6 Byte-Reihenfolge – Endianness
 - 5.6 Speicherausrichtung – Alignment
- 6 Objektorientierte Programmierung**
- 7 Ergänzungen und Ausblicke**

5 Hardwarenahe Programmierung

5.1 Bit-Operationen

5.1.1 Zahlensysteme

Oktal- und Hexadezimal-Zahlen lassen sich ziffernweise in Binär-Zahlen umrechnen:

000	0	0000	0	1000	8
001	1	0001	1	1001	9
010	2	0010	2	1010	A
011	3	0011	3	1011	B
100	4	0100	4	1100	C
101	5	0101	5	1101	D
110	6	0110	6	1110	E
111	7	0111	7	1111	F

Auswendig lernen!

5.1.2 Bit-Operationen in C

C-Operator	Verknüpfung	Anwendung
&	Und	Bits gezielt löschen
	Oder	Bits gezielt setzen
^	Exklusiv-Oder	Bits gezielt invertieren
~	Nicht	Alle Bits invertieren
<<	Verschiebung nach links	Maske generieren
>>	Verschiebung nach rechts	Bits isolieren

Beispiele:

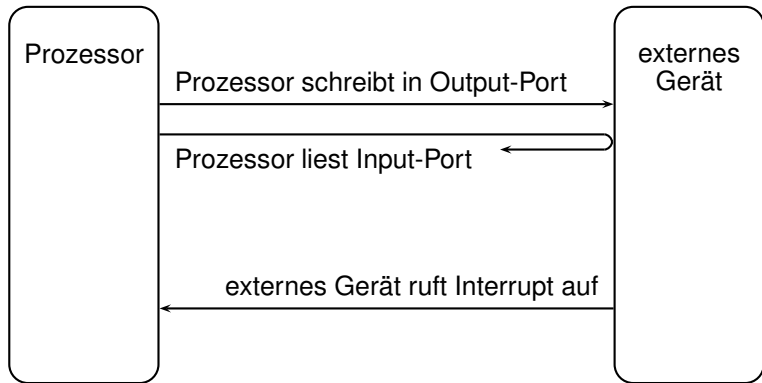
- Bit Nr. 5 gezielt auf 1 setzen: `PORTB |= 1 << 5;`
- Bit Nr. 6 gezielt auf 0 setzen: `PORTA &= ~(1 << 6);`
- Ist Bit Nr. 4 gesetzt? `if (PINC & (1 << 4) ...`
- Umschalten zwischen Ein- und Ausgabe: `DDR`
Bit = 1: Ausgabe; Bit = 0: Eingabe

**Details abhängig von
Prozessor und Compiler!**

5.2 I/O-Ports

5.3 Interrupts

Kommunikation mit externen Geräten



5.4 volatile-Variable

```
volatile uint16_t mleft_counter;
```

```
/* ... */
```

„Immer lesen und schreiben. Nicht wegoptimieren.“

```
volatile uint16_t mleft_dist;
```

```
/* ... */
```

```
ISR (INT0_vect)
```

```
{  
    mleft_dist++;  
    mleft_counter++;  
    /* ... */  
}
```

```
int main (void)
```


```
{  
    int prev_mleft_dist = mleft_dist;  
    while (mleft_dist == prev_mleft_dist)  
        /* just wait */;  
}
```

5.4 volatile-Variable

PORTA, PORTB, DDRA usw. sind **volatile**-Variable an numerisch vorgegebenen Speicheradressen (z. B. **0x38** für PORTB).

$(*(\text{volatile uint8_t} *) (0 \times 18 + 0 \times 20))$
(explizite Typumwandlg.) 0x38
in einen Zeiger
auf 8-Bit-Zahlen ohne Vorzeichen
ohne Zugriffsoptimierung

⇒ PORTB ≡ Speicherzelle Nr. 38₁₆



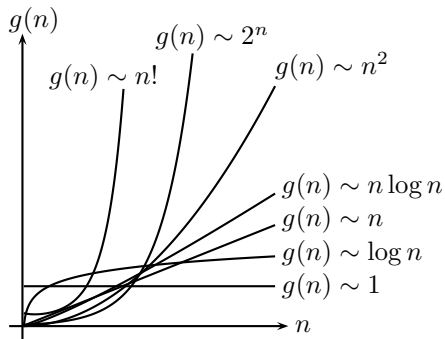
Zeiger dereferenzieren: Speicherzelle Nr. x direkt beschreiben →

Unwandlung einer Zahl in einen Zeiger
= Speicherzelle Nr. x
direkt ansprechen
hier: x = 0x38

5.5 Aufwandsabschätzungen

Beispiel: Sortialgorithmen

- Maximum suchen: $\mathcal{O}(n)$



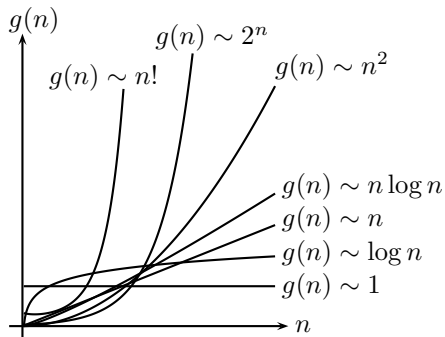
n : Eingabedaten

$g(n)$: Rechenzeit

5.5 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$



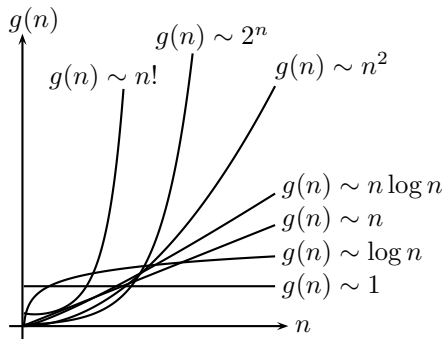
n : Eingabedaten

$g(n)$: Rechenzeit

5.5 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- zufällig mischen, bis sortiert
→ Monkeysort: $\mathcal{O}(n!)$



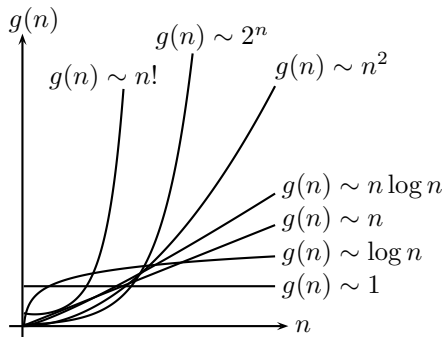
n : Eingabedaten

$g(n)$: Rechenzeit

5.5 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- zufällig mischen, bis sortiert
→ Monkeysort: $\mathcal{O}(n!)$
- Während Maximumsuche prüfen, abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$



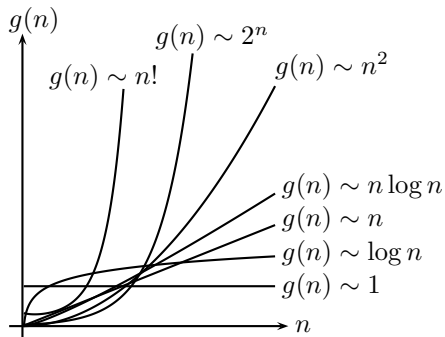
n : Eingabedaten

$g(n)$: Rechenzeit

5.5 Aufwandsabschätzungen

Beispiel: Sortialgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- zufällig mischen, bis sortiert
→ Monkeysort: $\mathcal{O}(n!)$
- Während Maximumsuche prüfen, abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$
- Rekursiv sortieren
→ Quicksort: $\mathcal{O}(n \log n)$ bis $\mathcal{O}(n^2)$



n : Eingabedaten

$g(n)$: Rechenzeit

5.6 Dynamische Speicherverwaltung

```
char *name[] = { "Anna", "Berthold", "Caesar" };
```

```
...
```

```
name[3] = "Dieter";
```

5.6 Dynamische Speicherverwaltung

```
#include <stdlib.h>
```

```
...
```

```
char **name = malloc (3 * sizeof (char *));  
    /* Speicherplatz für 3 Zeiger anfordern */
```

```
...
```

```
free (name)  
    /* Speicherplatz freigeben */
```

Angewandte Informatik

- 1 Einführung**
- 2 Einführung in C**
- 3 Bibliotheken**
- 4 Algorithmen**
 - 4.1** Differentialgleichungen
 - 4.2** Rekursion
 - 4.3** Stack und FIFO
 - 4.4** Aufwandsabschätzungen
 - 4.4** Dynamische Speicherverwaltung
- 5 Hardwarenahe Programmierung**
 - ...
 - 5.4** volatile-Variable
 - 5.5** Software-Interrupts
 - 5.6** Byte-Reihenfolge – Endianness
 - 5.6** Speicherausrichtung – Alignment
- 6 Objektorientierte Programmierung**
- 7 Ergänzungen und Ausblicke**

Angewandte Informatik

Prof. Dr. rer. nat. Peter Gerwinski

14. Januar 2016

Angewandte Informatik

1 Einführung

2 Einführung in C

3 Bibliotheken

4 Algorithmen

4.1 Differentialgleichungen

4.2 Rekursion

4.3 Stack und FIFO

4.4 Aufwandsabschätzungen

4.4 Dynamische Speicherverwaltung

5 Hardwarenahe Programmierung

...

5.4 volatile-Variable

5.5 Software-Interrupts

5.6 Byte-Reihenfolge – Endianness

5.6 Speicherausrichtung – Alignment

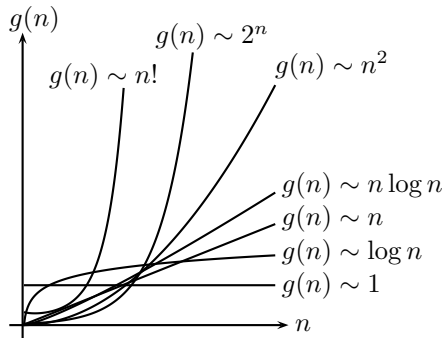
6 Objektorientierte Programmierung

7 Ergänzungen und Ausblicke

4.4 Aufwandsabschätzungen

Beispiel: Sortialgorithmen

- Maximum suchen: $\mathcal{O}(n)$



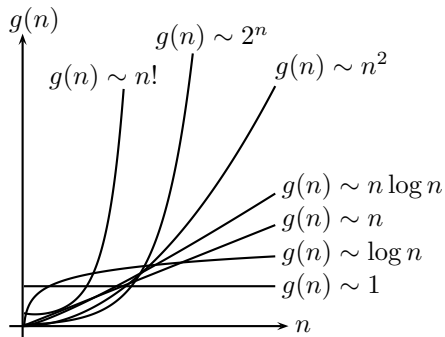
n : Eingabedaten

$g(n)$: Rechenzeit

4.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$



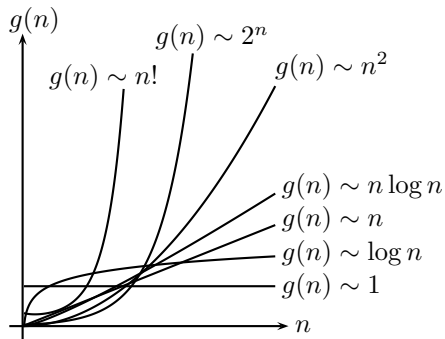
n : Eingabedaten

$g(n)$: Rechenzeit

4.4 Aufwandsabschätzungen

Beispiel: Sortialgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- zufällig mischen, bis sortiert
→ Monkeysort: $\mathcal{O}(n!)$



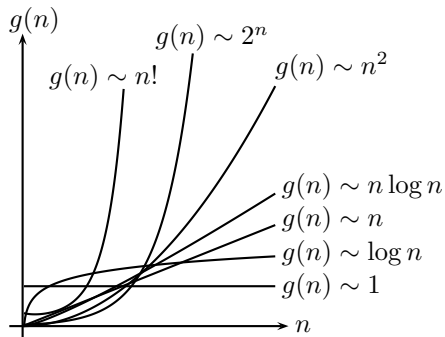
n : Eingabedaten

$g(n)$: Rechenzeit

4.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- zufällig mischen, bis sortiert
→ Monkeysort: $\mathcal{O}(n!)$
- Während Maximumsuche prüfen, abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$



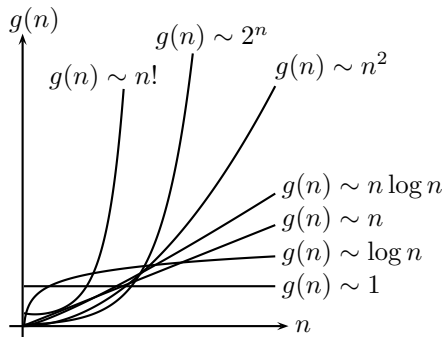
n : Eingabedaten

$g(n)$: Rechenzeit

4.4 Aufwandsabschätzungen

Beispiel: Sortialgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- zufällig mischen, bis sortiert
→ Monkeysort: $\mathcal{O}(n!)$
- Während Maximumsuche prüfen, abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$
- Rekursiv sortieren
→ Quicksort: $\mathcal{O}(n \log n)$ bis $\mathcal{O}(n^2)$



n : Eingabedaten

$g(n)$: Rechenzeit

4.5 Dynamische Speicherverwaltung

```
char *name[] = { "Anna", "Berthold", "Caesar" };
```

```
...
```

```
name[3] = "Dieter";
```


4.5 Dynamische Speicherverwaltung

```
#include <stdlib.h>
```

```
...
```

```
char **name = malloc (3 * sizeof (char *));  
    /* Speicherplatz für 3 Zeiger anfordern */
```

```
...
```

```
free (name)  
    /* Speicherplatz freigeben */
```

5 Hardwarenahe Programmierung

5.1 Bit-Operationen

5.1.1 Zahlensysteme

Oktal- und Hexadezimal-Zahlen lassen sich ziffernweise in Binär-Zahlen umrechnen:

000	0	0000	0	1000	8
001	1	0001	1	1001	9
010	2	0010	2	1010	A
011	3	0011	3	1011	B
100	4	0100	4	1100	C
101	5	0101	5	1101	D
110	6	0110	6	1110	E
111	7	0111	7	1111	F

Auswendig lernen!

5.1.2 Bit-Operationen in C

C-Operator	Verknüpfung	Anwendung
&	Und	Bits gezielt löschen
	Oder	Bits gezielt setzen
^	Exklusiv-Oder	Bits gezielt invertieren
~	Nicht	Alle Bits invertieren
<<	Verschiebung nach links	Maske generieren
>>	Verschiebung nach rechts	Bits isolieren

Beispiele:

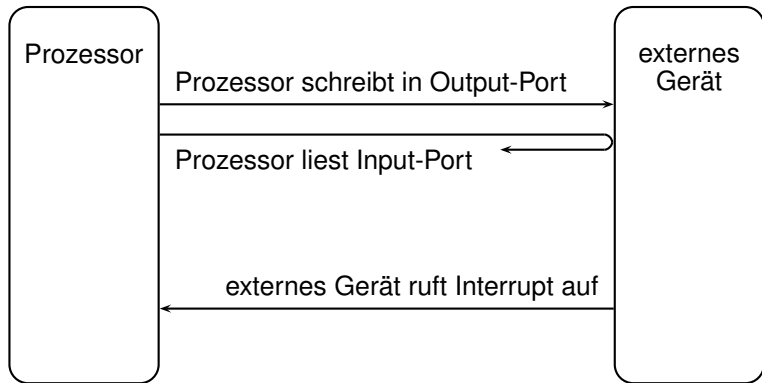
- Bit Nr. 5 gezielt auf 1 setzen: `PORTB |= 1 << 5;`
- Bit Nr. 6 gezielt auf 0 setzen: `PORTA &= ~(1 << 6);`
- Ist Bit Nr. 4 gesetzt? `if (PINC & (1 << 4) ...`
- Umschalten zwischen Ein- und Ausgabe: `DDR`
Bit = 1: Ausgabe; Bit = 0: Eingabe

**Details abhängig von
Prozessor und Compiler!**

5.2 I/O-Ports

5.3 Interrupts

Kommunikation mit externen Geräten



5.4 volatile-Variable

```
volatile uint16_t mleft_counter;
```

```
/* ... */
```

„Immer lesen und schreiben. Nicht wegoptimieren.“

```
volatile uint16_t mleft_dist;
```

```
/* ... */
```

```
ISR (INT0_vect)
```

```
{  
    mleft_dist++;  
    mleft_counter++;  
    /* ... */  
}
```

```
int main (void)
```


```
{  
    int prev_mleft_dist = mleft_dist;  
    while (mleft_dist == prev_mleft_dist)  
        /* just wait */;  
}
```

5.4 volatile-Variable

PORTA, PORTB, DDRA usw. sind **volatile**-Variable an numerisch vorgegebenen Speicheradressen (z. B. 0x38 für PORTB).

$(\ast(\text{volatile uint8_t } \ast)(0 \times 18 + 0 \times 20))$
(explizite Typumwandlg.) 0x38
in einen Zeiger
auf 8-Bit-Zahlen ohne Vorzeichen
ohne Zugriffsoptimierung

$\Rightarrow \text{PORTB} \equiv \text{Speicherzelle Nr. } 38_{16}$



Zeiger dereferenzieren: Speicherzelle Nr. x direkt beschreiben \rightarrow Unwandlung einer Zahl in einen Zeiger = Speicherzelle Nr. x direkt ansprechen
hier: x = 0x38

5.5 Software-Interrupts

```
mov ax, 0012
```

```
int 10
```

5.5 Software-Interrupts

`mov ax, 0012` ← Parameter in Prozessorregister

`int 10` ← Funktionsaufruf über Interrupt-Vektor

5.5 Software-Interrupts

`mov ax, 0012` ← Parameter in Prozessorregister
`int 10` ← Funktionsaufruf über Interrupt-Vektor

Beispiel: VGA-Grafikkarte

- Modus setzen: `mov ah, 00`
- Grafikmodus: `mov al, 12`
- Textmodus: `mov al, 03`

5.5 Software-Interrupts

`mov ax, 0012` ← Parameter in Prozessorregister
`int 10` ← Funktionsaufruf über Interrupt-Vektor

Beispiel: VGA-Grafikkarte

- Modus setzen: `mov ah, 00`
- Grafikmodus: `mov al, 12`
- Textmodus: `mov al, 03`

Verschiedene Farben: Output-Ports

- *Graphics Register*: Index `03CE`, Daten `03CF`
- Index 0: *Set/Reset Register*
- Index 1: *Enable Set/Reset Register*
- Index 8: *Bit Mask Register*
- Jedes Bit steht für Schreibzugriff auf eine Speicherbank.
- 4 Speicherbänke → 16 Farben

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.
Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen
Welche Bits liegen wo?

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\ 0100\ 0000\ 0011_2 = 0403_{16}$$

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\ 0100\ 0000\ 0011_2 = 0403_{16}$$

Speicherzellen:

04	03
----	----

 Big-Endian „großes Ende zuerst“

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\ 0100\ 0000\ 0011_2 = 0403_{16}$$

Speicherzellen:

04	03
----	----

Big-Endian „großes Ende zuerst“
für Menschen leichter lesbar

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\ 0100\ 0000\ 0011_2 = 0403_{16}$$

Speicherzellen:

04	03
----	----

 Big-Endian „großes Ende zuerst“
für Menschen leichter lesbar

03	04
----	----

 Little-Endian „kleines Ende zuerst“

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\ 0100\ 0000\ 0011_2 = 0403_{16}$$

Speicherzellen:

04	03
----	----

Big-Endian „großes Ende zuerst“
für Menschen leichter lesbar

03	04
----	----

Little-Endian „kleines Ende zuerst“
bei Additionen effizienter

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\ 0100\ 0000\ 0011_2 = 0403_{16}$$

Speicherzellen:

04	03
----	----

 Big-Endian „großes Ende zuerst“
für Menschen leichter lesbar

03	04
----	----

 Little-Endian „kleines Ende zuerst“
bei Additionen effizienter

→ Geschmackssache

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\ 0100\ 0000\ 0011_2 = 0403_{16}$$

Speicherzellen:

04	03
----	----

 Big-Endian „großes Ende zuerst“
für Menschen leichter lesbar

03	04
----	----

 Little-Endian „kleines Ende zuerst“
bei Additionen effizienter

→ Geschmackssache

... **außer bei Datenaustausch!**

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.
Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

—→ Geschmackssache

... **außer bei Datenaustausch!**

- Dateiformate
- Datenübertragung

5.6 Byte-Reihenfolge – Endianness

5.6.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian

5.6 Byte-Reihenfolge – Endianness

5.6.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian
- ältere AIFF-Dateien (.aiff): Big-Endian
- neuere AIFF-Dateien (.aiff): Little-Endian

5.6 Byte-Reihenfolge – Endianness

5.6.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian
- ältere AIFF-Dateien (.aiff): Big-Endian
- neuere AIFF-Dateien (.aiff): Little-Endian

Grafik-Formate: Reihenfolge der Bits in den Bytes

- PBM-Dateien: Big-Endian
- XBM-Dateien: Little-Endian

5.6 Byte-Reihenfolge – Endianness

5.6.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian
- ältere AIFF-Dateien (.aiff): Big-Endian
- neuere AIFF-Dateien (.aiff): Little-Endian

Grafik-Formate: Reihenfolge der Bits in den Bytes

- PBM-Dateien: Big-Endian, MSB first
- XBM-Dateien: Little-Endian, LSB first

MSB/LSB = most/least significant bit

5.6 Byte-Reihenfolge – Endianness

5.6.3 Datenübertragung

- RS-232 (serielle Schnittstelle): LSB first
- I²C: MSB first
- USB: beides

5.6 Byte-Reihenfolge – Endianness

5.6.3 Datenübertragung

- RS-232 (serielle Schnittstelle): LSB first
- I²C: MSB first
- USB: beides
- Ethernet: LSB first
- TCP/IP (Internet): Big-Endian

5.7 Speicherausrichtung – Alignment

5.7 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

5.7 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter

5.7 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- ... oder sogar nur dann erlaubt

5.7 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- ... oder sogar nur dann erlaubt

→ Compiler optimiert Speicherausrichtung

5.7 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- ... oder sogar nur dann erlaubt

→ Compiler optimiert Speicherausrichtung

```
uint8_t a;  
uint8_t dummy;  
uint16_t b;  
uint8_t c;
```

5.7 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- ... oder sogar nur dann erlaubt

→ Compiler optimiert Speicherausrichtung

```
uint8_t a;  
uint8_t dummy;    uint8_t a;  
uint16_t b;        uint8_t c;  
uint8_t c;         uint16_t b;
```


5.7 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- ... oder sogar nur dann erlaubt

→ Compiler optimiert Speicherausrichtung

```
uint8_t a;  
uint8_t dummy;  
uint16_t b;  
uint8_t c;  
  
uint8_t a;  
uint8_t c;  
uint16_t b;
```

Fazit:

- Adressen von Variablen sind systemabhängig

5.7 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- ... oder sogar nur dann erlaubt

→ Compiler optimiert Speicherausrichtung

```
uint8_t a;  
uint8_t dummy;  
uint16_t b;  
uint8_t c;  
  
uint8_t a;  
uint8_t c;  
uint16_t b;
```

Fazit:

- Adressen von Variablen sind systemabhängig
- Bei Definition von Datenformaten Alignment beachten → effizienter

Angewandte Informatik

- 1 Einführung**
- 2 Einführung in C**
- 3 Bibliotheken**
- 4 Algorithmen**
- 5 Hardwarenahe Programmierung**
 - ...
 - 5.4** volatile-Variable
 - 5.5** Software-Interrupts
 - 5.6** Byte-Reihenfolge – Endianness
 - 5.6** Speicherausrichtung – Alignment
- 6 Objektorientierte Programmierung**
 - 6.1** Konzepte und Ziele
 - 6.2** Beispiel: Zahlen und Buchstaben
 - 6.3** Einführung in C++
- 7 Ergänzungen und Ausblicke**

6 Objektorientierte Programmierung

6.1 Konzepte und Ziele

- Array: feste Anzahl von Elementen desselben Typs (z. B.: 3 Zeiger)
- Dynamisches Array: variable Anzahl von Elementen desselben Typs
- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern

6 Objektorientierte Programmierung

6.1 Konzepte und Ziele

- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern
- Zeiger auf verschiedene Strukturen mit einem gemeinsamen Anteil von Datenfeldern
→ „verwandte“ *Objekte*, *Klassen* von Objekten
- Struktur, die *nur* den gemeinsamen Anteil enthält
→ „Vorfahr“, *Basisklasse*, *Vererbung*
- Explizite Typumwandlung eines Zeigers auf die Basisklasse in einen Zeiger auf die *abgeleitete Klasse*
→ Man kann ein Array unterschiedlicher Objekte in einer Schleife abarbeiten.
→ *Polymorphie*

6 Objektorientierte Programmierung

6.1 Konzepte und Ziele

- *Objekte, Klassen, Basisklassen, abgeleitete Klassen*
- *Vererbung, Polymorphie*
- Funktionen, die mit dem Objekt arbeiten: *Methoden*
- Aufgerufene Funktion hängt vom Typ des Objekts ab: *virtuelle Methode*
- Realisierung über Zeiger, die im Objekt gespeichert sind
(Genaugenommen: Tabelle von Zeigern)

6 Objektorientierte Programmierung

6.2 Beispiel: Zahlen und Buchstaben

```
typedef struct
{
    int type;
} t_base;
```

```
typedef struct
{
    int type;
    int content;
} t_integer;
```

```
typedef struct
{
    int type;
    char *content;
} t_string;
```

```
typedef struct
```

```
{  
    int type;  
} t_base;
```

```
typedef struct
```

```
{  
    int type;  
    int content;  
} t_integer;
```

```
typedef struct
```

```
{  
    int type;  
    char *content;  
} t_string;
```

```
t_integer i = { 1, 42 };
```

```
t_string s = { 2, "Hello,_world!" };
```

```
t_base *object[] = { (t_base *) &i, (t_base *) &s };
```


explizite

Typumwandlung

6 Objektorientierte Programmierung

6.2 Beispiel: Zahlen und Buchstaben

Weitere Beispiele:

- Editor für graphische Objekte
- Datenbank-Software
- graphische Benutzeroberfläche (GUI)

6 Objektorientierte Programmierung

6.3 Einführung in C++

```
typedef struct  
{  
    int type;  
} t_base;
```

```
typedef struct  
{  
    int type;  
    int content;  
} t_integer;
```

```
typedef struct  
{  
    int type;  
    char *content;  
} t_string;
```

6 Objektorientierte Programmierung

6.3 Einführung in C++

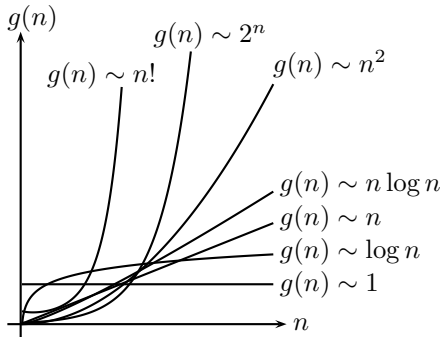
```
struct TBase  
{  
};
```

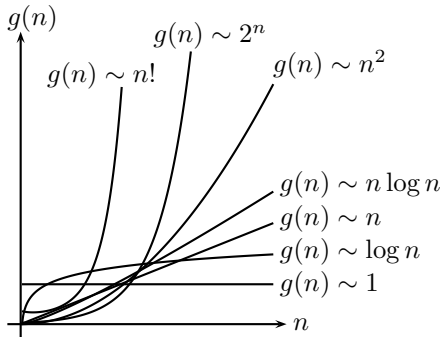
```
struct TInteger: public TBase  
{  
    int content;  
};
```

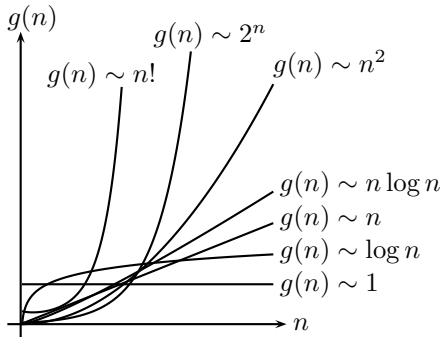
```
struct TString: public TBase  
{  
    char *content;  
};
```

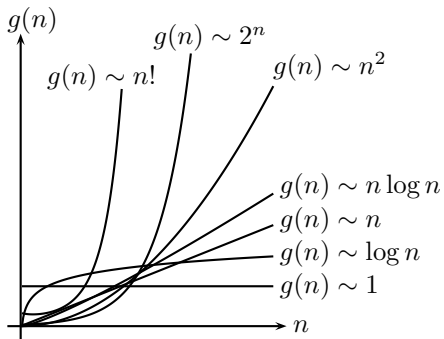
Angewandte Informatik

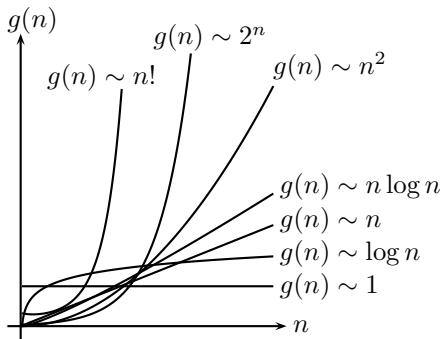
- 1 Einführung**
- 2 Einführung in C**
- 3 Bibliotheken**
- 4 Algorithmen**
- 5 Hardwarenahe Programmierung**
 - ...
 - 5.4 volatile-Variable
 - 5.5 Software-Interrupts
 - 5.6 Byte-Reihenfolge – Endianness
 - 5.6 Speicherausrichtung – Alignment
- 6 Objektorientierte Programmierung**
 - 6.1 Konzepte und Ziele
 - 6.2 Beispiel: Zahlen und Buchstaben
 - 6.3 Einführung in C++
- 7 Ergänzungen und Ausblicke**

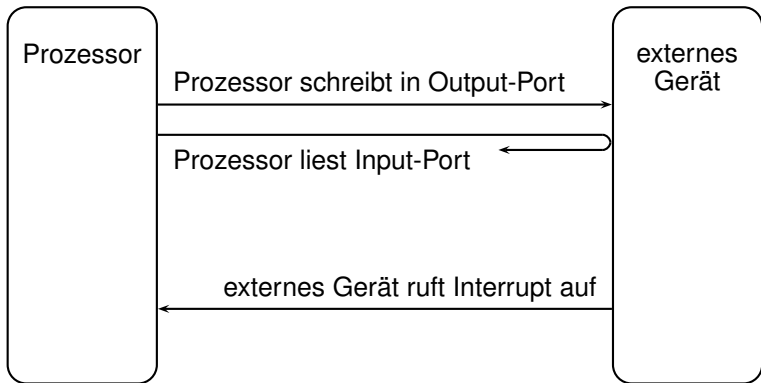












Angewandte Informatik

Prof. Dr. rer. nat. Peter Gerwinski

21. Januar 2016

Angewandte Informatik

- 1 Einführung**
- 2 Einführung in C**
- 3 Bibliotheken**
- 4 Algorithmen**
- 5 Hardwarenahe Programmierung**
 - ...
 - 5.4** volatile-Variable
 - 5.5** Software-Interrupts
 - 5.6** Byte-Reihenfolge – Endianness
 - 5.6** Speicherausrichtung – Alignment
- 6 Objektorientierte Programmierung**
 - 6.1** Konzepte und Ziele
 - 6.2** Beispiel: Zahlen und Buchstaben
 - 6.3** Einführung in C++
- 7 Ergänzungen und Ausblicke**

5.5 Software-Interrupts

`mov ax, 0012` ← Parameter in Prozessorregister
`int 10` ← Funktionsaufruf über Interrupt-Vektor

Beispiel: VGA-Grafikkarte

- Modus setzen: `mov ah, 00`
- Grafikmodus: `mov al, 12`
- Textmodus: `mov al, 03`

Verschiedene Farben: Output-Ports

- *Graphics Register*: Index `03CE`, Daten `03CF`
- Index 0: *Set/Reset Register*
- Index 1: *Enable Set/Reset Register*
- Index 8: *Bit Mask Register*
- Jedes Bit steht für Schreibzugriff auf eine Speicherbank.
- 4 Speicherbänke → 16 Farben

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\ 0100\ 0000\ 0011_2 = 0403_{16}$$

Speicherzellen:

04	03
----	----

 Big-Endian „großes Ende zuerst“
für Menschen leichter lesbar

03	04
----	----

 Little-Endian „kleines Ende zuerst“
bei Additionen effizienter

→ Geschmackssache

... **außer bei Datenaustausch!**

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.
Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

—→ Geschmackssache

... **außer bei Datenaustausch!**

- Dateiformate
- Datenübertragung

5.6 Byte-Reihenfolge – Endianness

5.6.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian
- ältere AIFF-Dateien (.aiff): Big-Endian
- neuere AIFF-Dateien (.aiff): Little-Endian

Grafik-Formate: Reihenfolge der Bits in den Bytes

- PBM-Dateien: Big-Endian, MSB first
- XBM-Dateien: Little-Endian, LSB first

MSB/LSB = most/least significant bit

5.6 Byte-Reihenfolge – Endianness

5.6.3 Datenübertragung

- RS-232 (serielle Schnittstelle): LSB first
- I²C: MSB first
- USB: beides
- Ethernet: LSB first
- TCP/IP (Internet): Big-Endian

5.7 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- ... oder sogar nur dann erlaubt

→ Compiler optimiert Speicherausrichtung

```
uint8_t a;  
uint8_t dummy;  
uint16_t b;  
uint8_t c;  
  
uint8_t a;  
uint8_t c;  
uint16_t b;
```

Fazit:

- Adressen von Variablen sind systemabhängig
- Bei Definition von Datenformaten Alignment beachten → effizienter

6 Objektorientierte Programmierung

6.1 Konzepte und Ziele

- Array: feste Anzahl von Elementen desselben Typs (z. B.: 3 Zeiger)
- Dynamisches Array: variable Anzahl von Elementen desselben Typs
- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern

6 Objektorientierte Programmierung

6.1 Konzepte und Ziele

- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern
- Zeiger auf verschiedene Strukturen mit einem gemeinsamen Anteil von Datenfeldern
→ „verwandte“ *Objekte*, *Klassen* von Objekten
- Struktur, die *nur* den gemeinsamen Anteil enthält
→ „Vorfahr“, *Basisklasse*, *Vererbung*
- Explizite Typumwandlung eines Zeigers auf die Basisklasse in einen Zeiger auf die *abgeleitete Klasse*
→ Man kann ein Array unterschiedlicher Objekte in einer Schleife abarbeiten.
→ *Polymorphie*

6 Objektorientierte Programmierung

6.1 Konzepte und Ziele

- *Objekte, Klassen, Basisklassen, abgeleitete Klassen*
- *Vererbung, Polymorphie*
- Funktionen, die mit dem Objekt arbeiten: *Methoden*
- Aufgerufene Funktion hängt vom Typ des Objekts ab: *virtuelle Methode*
- Realisierung über Zeiger, die im Objekt gespeichert sind
(Genaugenommen: Tabelle von Zeigern)

6 Objektorientierte Programmierung

6.2 Beispiel: Zahlen und Buchstaben

```
typedef struct
{
    int type;
} t_base;
```

```
typedef struct
{
    int type;
    int content;
} t_integer;
```

```
typedef struct
{
    int type;
    char *content;
} t_string;
```

```
typedef struct
{
    int type;
} t_base;
```

```
typedef struct
{
    int type;
    int content;
} t_integer;
```

```
typedef struct
{
    int type;
    char *content;
} t_string;
```

```
t_integer i = { 1, 42 };
t_string s = { 2, "Hello,_world!" };
```

```
t_base *object[] = { (t_base *) &i, (t_base *) &s };
```


explizite

Typumwandlung

```
typedef struct
```

```
{  
    int type;  
} t_base;
```

```
typedef struct
```

```
{  
    int type;  
    int content;  
} t_integer;
```

```
typedef struct
```

```
{  
    int type;  
    char *content;  
} t_string;
```

```
typedef union
```

```
{  
    t_base base;  
    t_integer integer;  
    t_string string;  
} t_object;
```



```
typedef struct
{
    void (* print) (union t_object *this);
} t_base;
```

```
typedef struct
{
    void (* print) (...);
    int content;
} t_integer;
```

```
typedef struct
{
    void (* print) (union t_object *this);
    char *content;
} t_string;
```

```
typedef union
{
    t_base base;
    t_integer integer;
    t_string string;
} t_object;
```

```
object[i]—>base.print (object[i]);
```

6 Objektorientierte Programmierung

6.2 Beispiel: Zahlen und Buchstaben

Weitere Beispiele:

- Editor für graphische Objekte
- Datenbank-Software
- graphische Benutzeroberfläche (GUI)

6 Objektorientierte Programmierung

6.3 Einführung in C++

```
typedef struct  
{  
    void (* print) (union t_object *this);  
} t_base;
```

```
typedef struct  
{  
    void (* print) (...);  
    int content;  
} t_integer;
```

```
typedef struct  
{  
    void (* print) (union t_object *this);  
    char *content;  
} t_string;
```

6 Objektorientierte Programmierung

6.3 Einführung in C++

```
struct TBase  
{  
    virtual void print (void);  
};
```

```
struct TInteger: public TBase  
{  
    virtual void print (void);  
    int content;  
};
```

```
struct TString: public TBase  
{  
    virtual void print (void);  
    char *content;  
};
```

7 Ergänzungen und Ausblicke

7.1 String-Operationen

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

```
if (strcmp (s1, s2) < 0)  
    printf ("%s_ist_alphabetisch_kleiner_als_%s.\n", s1, s2);
```

```
strlen ()
```

```
strcpy ()
```

```
strncpy ()
```

```
...
```

7 Ergänzungen und Ausblicke

7.2 Dateien

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

```
FILE *f = fopen ("fhello.txt", "w");  
fprintf (f, "Hello, world!\n");  
fclose (f);
```

7 Ergänzungen und Ausblicke

7.3 Wie geht es weiter?

- das Gelernte anwenden
- weitere Algorithmen und Bibliotheken
- Vertiefung C++
- ...

Empfehlung:

- lesen, was andere geschrieben haben
- Fehler verbessern
- eigene Vorstellungen realisieren

Angewandte Informatik

- 1 Einführung**
- 2 Einführung in C**
- 3 Bibliotheken**
- 4 Algorithmen**
- 5 Hardwarenahe Programmierung**
 - ...
 - 5.4** volatile-Variable
 - 5.5** Software-Interrupts
 - 5.6** Byte-Reihenfolge – Endianness
 - 5.6** Speicherausrichtung – Alignment
- 6 Objektorientierte Programmierung**
 - 6.1** Konzepte und Ziele
 - 6.2** Beispiel: Zahlen und Buchstaben
 - 6.3** Einführung in C++
- 7 Ergänzungen und Ausblicke**

Angewandte Informatik

- 1 Einführung**
- 2 Einführung in C**
- 3 Bibliotheken**
- 4 Algorithmen**
- 5 Hardwarenahe Programmierung**
 - ...
 - 5.4** volatile-Variable
 - 5.5** Software-Interrupts
 - 5.6** Byte-Reihenfolge – Endianness
 - 5.6** Speicherausrichtung – Alignment
- 6 Objektorientierte Programmierung**
 - 6.1** Konzepte und Ziele
 - 6.2** Beispiel: Zahlen und Buchstaben
 - 6.3** Einführung in C++
- 7 Ergänzungen und Ausblicke**