

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

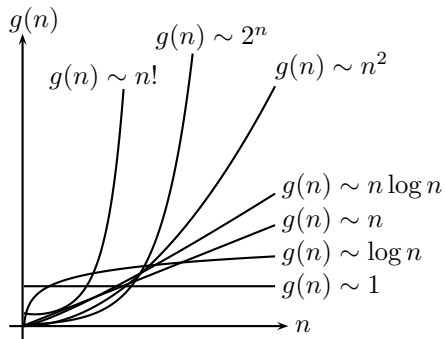
6 Objektorientierte Programmierung

7 Ergänzungen und Ausblicke

4.4 Aufwandsabschätzungen

Beispiel: Sortialgorithmen

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



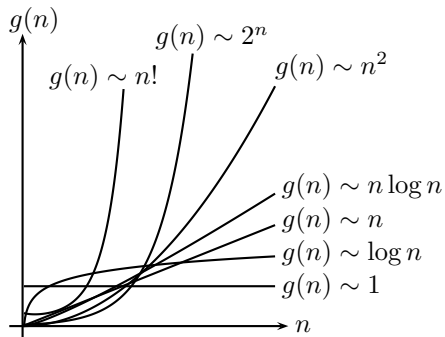
n : Eingabedaten

$g(n)$: Rechenzeit

4.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

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



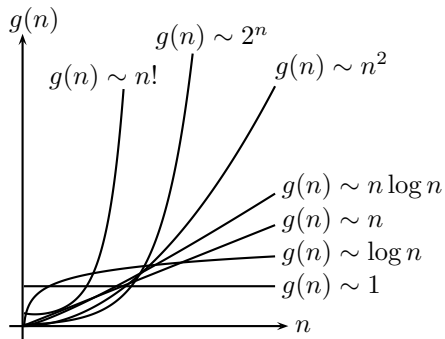
n : Eingabedaten

$g(n)$: 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: $\mathcal{O}(n!)$



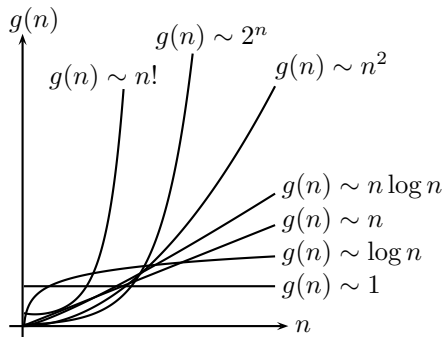
n : Eingabedaten

$g(n)$: 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 $\mathcal{O}(n^2)$



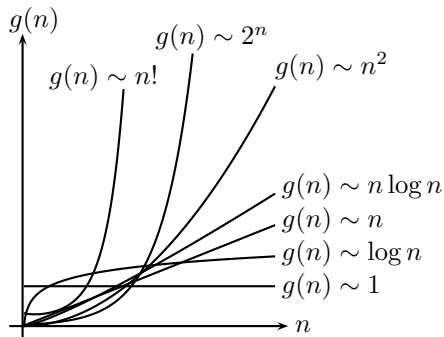
n : Eingabedaten

$g(n)$: 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:

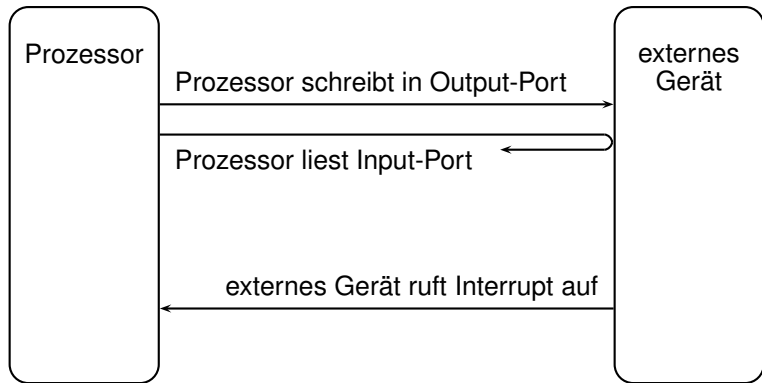
- Bit Nr. 5 gezielt auf 1 setzen: `PORTB |= 1 << 5;`
- Bit Nr. 6 gezielt auf 0 setzen: `PORTA &= ~(1 << 6);`
- Ist Bit Nr. 4 gesetzt? `if (PINC & (1 << 4) ...`
- Umschalten zwischen Ein- und Ausgabe: `DDR`
Bit = 1: Ausgabe; Bit = 0: Eingabe

**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 */;  
}
```

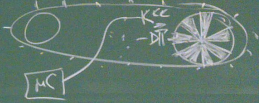
5.4 volatile-Variable

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

$$(*(\text{volatile uint8_t } *) (0 \times 18 + 0 \times 20))$$

(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 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**