

# Angewandte Informatik

## Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

19. Dezember 2016

# Angewandte Informatik

## Hardwarenahe Programmierung

**1 Einführung**

**2 Einführung in C**

**3 Bibliotheken**

**4 Algorithmen**

**5 Hardwarenahe Programmierung**

...

5.5 Byte-Reihenfolge – Endianness

5.6 Speicherausrichtung – Alignment

**6 Objektorientierte Programmierung**

6.0 Dynamische Speicherverwaltung

6.1 Konzepte und Ziele

6.2 Beispiel: Zahlen und Buchstaben

6.3 Unions

6.4 Beispiel: graphische Benutzeroberfläche (GUI)

6.5 Einführung in C++

...

## 5.1 Bit-Operationen

Numerierung der Bits: von rechts ab 0

Bit Nr. 3 auf 1 setzen:  $a |= 1 << 3;$        $a |= 0x08;$   
Bit Nr. 4 auf 0 setzen:  $a \&= \sim(1 << 4);$        $a \&= \sim 0x10;$   
Bit Nr. 0 invertieren:  $a \wedge= 1 << 0;$        $a \wedge= 0x01;$

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

## 5.4<sup>1/2</sup> Binärdarstellung von Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

8-Bit-Zahlen ohne Vorzeichen: `uint8_t`

→ Zahlenwerte von `0x00` bis `0xff` = 0 bis 255

## 5.4<sup>1/2</sup> Binärdarstellung von Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

8-Bit-Zahlen ohne Vorzeichen: `uint8_t`

→ Zahlenwerte von `0x00` bis `0xff` = 0 bis 255

→  $255 + 1 = 0$

## 5.4<sup>1/2</sup> Binärdarstellung von Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

8-Bit-Zahlen ohne Vorzeichen: `uint8_t`

→ Zahlenwerte von `0x00` bis `0xff` = 0 bis 255

→  $255 + 1 = 0$

8-Bit-Zahlen mit Vorzeichen: `int8_t`

`0xff` = 255 ist die „natürliche“ Schreibweise für  $-1$ .

## 5.4<sup>1/2</sup> Binärdarstellung von Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

8-Bit-Zahlen ohne Vorzeichen: `uint8_t`

→ Zahlenwerte von `0x00` bis `0xff` = 0 bis 255

→  $255 + 1 = 0$

8-Bit-Zahlen mit Vorzeichen: `int8_t`

`0xff` = 255 ist die „natürliche“ Schreibweise für  $-1$ .

→ Zweierkomplement

## 5.4<sup>1/2</sup> Binärdarstellung von Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

8-Bit-Zahlen ohne Vorzeichen: `uint8_t`

→ Zahlenwerte von `0x00` bis `0xff` = 0 bis 255

→  $255 + 1 = 0$

8-Bit-Zahlen mit Vorzeichen: `int8_t`

`0xff` = 255 ist die „natürliche“ Schreibweise für  $-1$ .

→ Zweierkomplement

Oberstes Bit = 1: negativ

Oberstes Bit = 0: positiv

→  $127 + 1 = -128$



## 5.4<sup>1/2</sup> Binärdarstellung von Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

16-Bit-Zahlen ohne Vorzeichen: `uint16_t`

→ Zahlenwerte von `0x0000` bis `0xffff` = 0 bis 65535

→  $65535 + 1 = 0$

`uint8_t`

0 bis 255

$255 + 1 = 0$

16-Bit-Zahlen mit Vorzeichen: `int16_t`

`0xffff` = 65535 ist die „natürliche“ Schreibweise für  $-1$ .

→ Zweierkomplement

`int8_t`

`0xff` = 255 =  $-1$

Oberstes Bit = 1: negativ

Oberstes Bit = 0: positiv

→  $32767 + 1 = -32768$

Literatur: <http://xkcd.com/571/>

## 5.4<sup>1/2</sup> Binärdarstellung von Zahlen

Frage: Für welche Zahl steht der Speicherinhalt `0x90a3`?

## 5.4<sup>1/2</sup> Binärdarstellung von Zahlen

Frage: Für welche Zahl steht der Speicherinhalt *0x90a3*?

Antwort: Das kommt darauf an. ;—)

## 5.4<sup>1/2</sup> Binärdarstellung von Zahlen

Frage: Für welche Zahl steht der Speicherinhalt `0x90a3`?

Antwort: Das kommt darauf an. ;–)

als <code>int8_t</code> :	–93	(nur unteres Byte, Little-Endian)
als <code>uint8_t</code> :	163	(nur unteres Byte, Little-Endian)
als <code>int16_t</code> :	–28509	
als <code>uint16_t</code> :	37027	
<code>int32_t</code> oder größer:	37027	(zusätzliche Bytes mit Nullen aufgefüllt)

## 5.5 Byte-Reihenfolge – Endianness

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**: Dateiformate, Datenübertragung

## 5.6 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

## 6.0 Dynamische Speicherverwaltung

- Array: feste Anzahl von Elementen desselben Typs (z. B. 3 ganze Zahlen)
- Dynamisches Array: variable Anzahl von Elementen desselben Typs

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

...

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

# 6 Objektorientierte Programmierung

## 6.0 Dynamische Speicherverwaltung

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

```
...
```

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

```
...
```

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



## 6 Objektorientierte Programmierung

### 6.1 Konzepte und Ziele

- Array: feste Anzahl von Elementen desselben Typs (z. B. 3 ganze Zahlen)
- 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

- Array: feste Anzahl von Elementen desselben Typs (z. B. 3 ganze Zahlen)
- Dynamisches Array: variable Anzahl von Elementen desselben Typs
- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern
- Funktionen, die mit dem Objekt arbeiten: *Methoden*
- Was die Funktion bewirkt, hängt vom Typ des Objekts ab
- Realisierung über endlose **if**-Ketten

# 6 Objektorientierte Programmierung

## 6.1 Konzepte und Ziele

- Array: feste Anzahl von Elementen desselben Typs (z. B. 3 ganze Zahlen)
- Dynamisches Array: variable Anzahl von Elementen desselben Typs
- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern
- Funktionen, die mit dem Objekt arbeiten: *Methoden*
- ~~Was die Funktion bewirkt~~ Welche Funktion aufgerufen wird, hängt vom Typ des Objekts ab: *virtuelle Methode*
- Realisierung über ~~endlose if-Ketten~~ Zeiger, die im Objekt gespeichert sind (Genaugenommen: Tabelle von Zeigern)

→ **nächstes Jahr**

# 6 Objektorientierte Programmierung

## 6.1 Konzepte und Ziele

- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern
- *Methoden* und *virtuelle Methoden*
- 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*
- Zeiger auf die Basisklasse dürfen auf Objekte der *abgeleiteten Klasse* zeigen  
→ *Polymorphie*

# 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;
```

## 6.3 Unions

Variable teilen sich denselben Speicherplatz.

```
typedef union
```

```
{  
    int8_t i;  
    uint8_t u;  
} num8_t;
```



## 6.3 Unions

Variable teilen sich denselben Speicherplatz.

**typedef union**

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

**typedef struct**

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

**typedef struct**

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

```
if (this->base.type == T_INTEGER)  
    printf ("Integer:_%d\n", this->integer.content);  
else if (this->base.type == T_STRING)  
    printf ("String:_\"%s\"\n", this->string.content);
```

# 6 Objektorientierte Programmierung

## 6.4 Beispiel: graphische Benutzeroberfläche (GUI)

```
#include <gtk/gtk.h>
```

```
int main (int argc, char **argv)
```

```
{  
    gtk_init (&argc, &argv);  
    GtkWidget *window = gtk_window_new (GTK_WINDOW_TOPLEVEL);  
    gtk_window_set_title (GTK_WINDOW (window), "Hello");  
    g_signal_connect (window, "destroy", G_CALLBACK (gtk_main_quit), NULL);  
    GtkWidget *vbox = gtk_box_new (GTK_ORIENTATION_VERTICAL, 5);  
    gtk_container_add (GTK_CONTAINER (window), vbox);  
    gtk_container_set_border_width (GTK_CONTAINER (vbox), 10);  
    GtkWidget *label = gtk_label_new ("Hello, world!");  
    gtk_container_add (GTK_CONTAINER (vbox), label);  
    GtkWidget *button = gtk_button_new_with_label ("Quit");  
    g_signal_connect (button, "clicked", G_CALLBACK (gtk_main_quit), NULL);  
    gtk_container_add (GTK_CONTAINER (vbox), button);  
    gtk_widget_show (button);  
    gtk_widget_show (label);  
    gtk_widget_show (vbox);  
    gtk_widget_show (window);  
    gtk_main ();  
    return 0;  
}
```



# Angewandte Informatik

## Hardwarenahe Programmierung

**1 Einführung**

**2 Einführung in C**

**3 Bibliotheken**

**4 Algorithmen**

**5 Hardwarenahe Programmierung**

**5.4<sup>1/2</sup> Binärdarstellung von Zahlen**

**5.5** Byte-Reihenfolge – Endianness

**5.6** Speicherausrichtung – Alignment

**6 Objektorientierte Programmierung**

**6.0** Dynamische Speicherverwaltung

**6.1** Konzepte und Ziele

**6.2** Beispiel: Zahlen und Buchstaben

**6.3** Unions

**6.4** Beispiel: graphische Benutzeroberfläche (GUI)

**6.5** Einführung in C++

...