

# Angewandte Informatik

## Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

9. Januar 2017

# Angewandte Informatik

## Hardwarenahe Programmierung

- 1 Einführung**
- 2 Einführung in C**
- 3 Bibliotheken**
- 4 Algorithmen**
- 5 Hardwarenahe Programmierung**
- 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** Virtuelle Methoden
  - 6.6** Einführung in C++

...

## 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`?

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)

# 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
- 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)

→ **kommt gleich**

# 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);
```

## 6.4 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;  
}
```



**Praktikumsversuch:  
Objektorientiertes Zeichenprogramm**



## 6.5 Virtuelle Methoden

```
void print_object (t_object *this)
```

```
{  
  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);  
}
```

if-Kette:  
wird unübersichtlich

```
void print_integer (t_object *this)
```

```
{  
  printf ("Integer:_%d\n", this->integer.content);  
}
```



Zeiger auf Funktionen

```
void print_string (t_object *this)
```

```
{  
  printf ("String:_%s\n", this->string.content);  
}
```

## 6.5 Virtuelle Methoden

Zeiger auf Funktionen

```
void (* print) (t_object *this);
```

 das, worauf print zeigt,  
ist eine Funktion

- Objekt enthält Zeiger auf Funktion
- Konstruktor initialisiert diesen Zeiger
- Aufruf: „automatisch“ die richtige Funktion
- in größeren Projekten:  
Objekt enthält Zeiger auf Tabelle von Funktionen

# 6 Objektorientierte Programmierung

## 6.6 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

## 6.6 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;  
};
```

# Angewandte Informatik

## Hardwarenahe Programmierung

- 1 Einführung**
- 2 Einführung in C**
- 3 Bibliotheken**
- 4 Algorithmen**
- 5 Hardwarenahe Programmierung**
- 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** Virtuelle Methoden
  - 6.6** Einführung in C++

...

# Angewandte Informatik

## Hardwarenahe Programmierung

- 1 Einführung
- 2 Einführung in C
- 3 Bibliotheken
- 4 Algorithmen
- 5 Hardwarenahe Programmierung
- 6 Objektorientierte Programmierung
- 7 Datenstrukturen
  - 7.1 Stack und FIFO

...

...

# 7 Datenstrukturen

## 7.1 Stack und FIFO

Im letzten Praktikumsversuch:

- Array nur zum Teil benutzt
- Variable speichert genutzte Länge
- Elemente hinten anfügen

# 7 Datenstrukturen

## 7.1 Stack und FIFO

Im letzten Praktikumsversuch:

- Array nur zum Teil benutzt
- Variable speichert genutzte Länge
- Elemente hinten anfügen  
oder entfernen



# 7 Datenstrukturen

## 7.1 Stack und FIFO

Im letzten Praktikumsversuch:

- Array nur zum Teil benutzt
- Variable speichert genutzte Länge
- Elemente hinten anfügen  
oder entfernen

→ Stack

# 7 Datenstrukturen

## 7.1 Stack und FIFO

Im letzten Praktikumsversuch:

- Array nur zum Teil benutzt
- Variable speichert genutzte Länge
- Elemente hinten anfügen  
oder entfernen

→ Stack

- hinten anfügen/entfernen:  $\mathcal{O}(1)$
- vorne oder in der Mitte  
anfügen/entfernen:  $\mathcal{O}(n)$

# 7 Datenstrukturen

## 7.1 Stack und FIFO

Im letzten Praktikumsversuch:

- Array nur zum Teil benutzt
- Variable speichert genutzte Länge
- Elemente hinten anfügen oder entfernen

→ Stack

- hinten anfügen/entfernen:  $\mathcal{O}(1)$
- vorne oder in der Mitte anfügen/entfernen:  $\mathcal{O}(n)$

Auch möglich:

- Array nur zum Teil benutzt
- 2 Variablen speichern genutzte Länge (ringförmig)
- Elemente hinten anfügen oder vorne entfernen

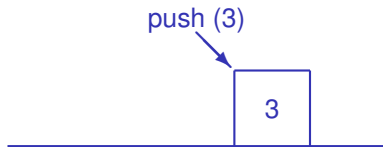
→ FIFO

- vorne oder hinten anfügen oder entfernen:  $\mathcal{O}(1)$
- in der Mitte anfügen/entfernen:  $\mathcal{O}(n)$

# 7 Datenstrukturen

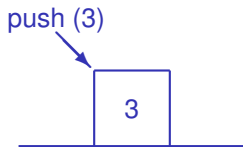
## 7.1 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

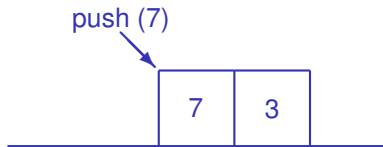


LIFO = Stack = Stapel

# 7 Datenstrukturen

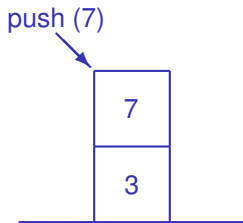
## 7.1 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



LIFO = Stack = Stapel

# 7 Datenstrukturen

## 7.1 Stack und FIFO

„First In – First Out“

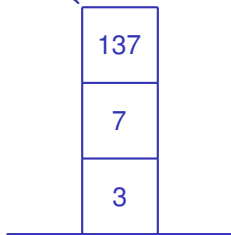
push (137)



FIFO = Queue = Reihe

„Last In – First Out“

push (137)



LIFO = Stack = Stapel

# 7 Datenstrukturen

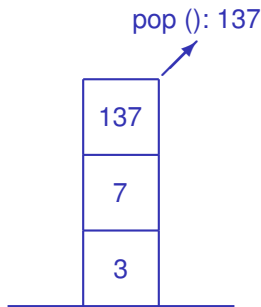
## 7.1 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

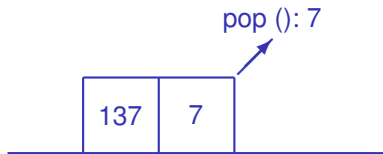


LIFO = Stack = Stapel

# 7 Datenstrukturen

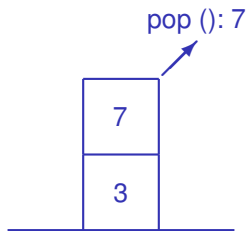
## 7.1 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



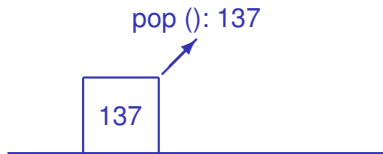
LIFO = Stack = Stapel



# 7 Datenstrukturen

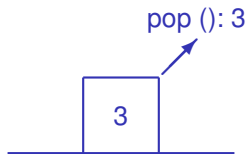
## 7.1 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



LIFO = Stack = Stapel

# Angewandte Informatik

## Hardwarenahe Programmierung

- 1 Einführung
- 2 Einführung in C
- 3 Bibliotheken
- 4 Algorithmen
- 5 Hardwarenahe Programmierung
- 6 Objektorientierte Programmierung
- 7 Datenstrukturen
  - 7.1 Stack und FIFO

...

...