

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

14. Januar 2019

Hardwarenahe Programmierung

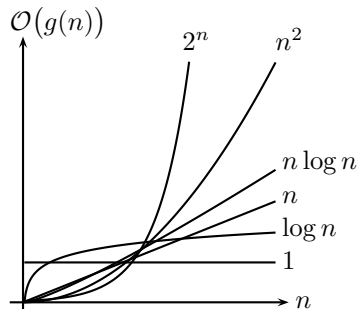
<https://gitlab.cvh-server.de/pgerwinski/hp.git>

- 1 Einführung**
- 2 Einführung in C**
- 3 Bibliotheken**
- 4 Hardwarenahe Programmierung**
- 5 Algorithmen**
 - 5.1 Differentialgleichungen
 - 5.2 Rekursion
 - 5.3 Aufwandsabschätzungen
- 6 Objektorientierte Programmierung**
 - 6.0 Dynamische Speicherverwaltung
 - 6.1 Konzepte und Ziele
 - 6.2 Beispiel: Zahlen und Buchstaben
 - 6.3 Unions
 - ...
- 7 Datenstrukturen**

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

- Türme von Hanoi: $\mathcal{O}(2^n)$
- Minimum suchen: $\mathcal{O}(n)$
- ... mit Schummeln: $\mathcal{O}(1)$
- Minimum an den Anfang tauschen, nächstes Minimum suchen:
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Minimumsuche prüfen und abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$
- Rekursiv sortieren
→ Quicksort: $\mathcal{O}(n \log n)$ bis $\mathcal{O}(n^2)$

Faustregel:
Schachtelung der Schleifen zählen
 x Schleifen → $\mathcal{O}(n^x)$

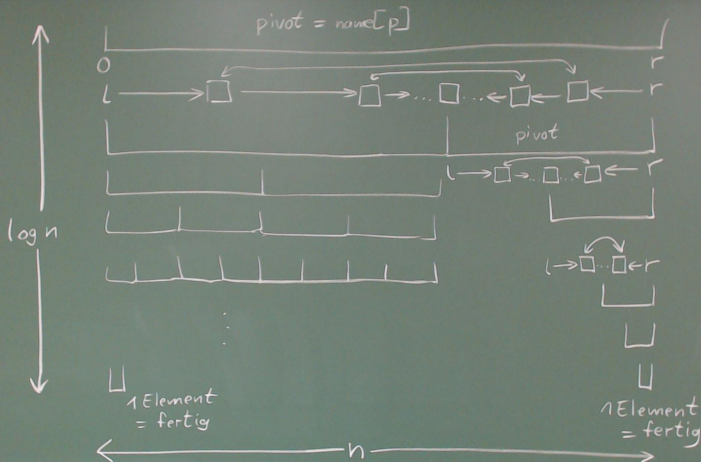


n : Eingabedaten

$g(n)$: Rechenzeit

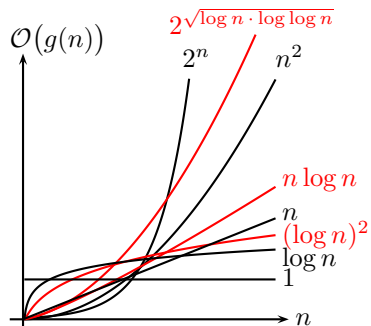
5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Quicksort



5.3 Aufwandsabschätzungen – Komplexitätsanalyse

- Türme von Hanoi: $\mathcal{O}(2^n)$
- Minimum suchen: $\mathcal{O}(n)$
- ... mit Schummeln: $\mathcal{O}(1)$
- Minimum an den Anfang tauschen, nächstes Minimum suchen:
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Minimumsuche prüfen und 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

Faustregel:

Schachtelung der Schleifen zählen

x Schleifen $\rightarrow \mathcal{O}(n^x)$

RSA: Schlüsselerzeugung (Berechnung von d): $\mathcal{O}((\log n)^2)$,

Ver- und Entschlüsselung (Exponentiation): $\mathcal{O}(n \log n)$,

Verschlüsselung brechen (Primfaktorzerlegung): $\mathcal{O}(2^{\sqrt{\log n \cdot \log \log n}})$

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

6.3 Unions

Variable teilen sich denselben Speicherplatz.

```
typedef union
```

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

```
int main (void)
```

```
{  
    num8_t test;  
    test.i = -1;  
    printf ("%d\n", test.u);  
    return 0;  
}
```

6.3 Unions

Variable teilen sich denselben Speicherplatz.

```
typedef union
```

```
{  
    char s[8];  
    uint64_t x;  
} num_char_t;
```

```
int main (void)
```

```
{  
    num_char_t test = { "Hello!" };  
    printf ("%lx\n", test.x);  
    return 0;  
}
```

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;  
} t_base;
```

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 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);  
}
```


6.4 Virtuelle Methoden

Zeiger auf Funktionen

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

 das, worauf print zeigt,
ist eine Funktion

- Objekt enthält Zeiger auf Funktion

```
typedef struct
```

```
{  
    void (* print) (union t_object *this);  
    int content;  
} t_integer;
```

6.4 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

```
t_object *new_integer (int i)
{
    t_object *p = malloc (sizeof (t_integer));
    p->integer.print = print_integer;
    p->integer.content = i;
    return p;
}
```

```
typedef struct
{
    void (* print) (union t_object *this);
    int content;
} t_integer;
```

6.4 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

```
for (int i = 0; object[i]; i++)  
    object[i]—>base.print (object[i]);
```

```
typedef struct  
{  
    void (* print) (union t_object *this);  
    int content;  
} t_integer;
```

6.4 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

```
typedef struct
```

```
{
```

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

```
    int content;
```

```
} t_integer;
```

6.5 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**

6.6 Ausblick: 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;
```

6.6 Ausblick: 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;  
};
```

Hardwarenahe Programmierung

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

- 1 Einführung**
- 2 Einführung in C**
- 3 Bibliotheken**
- 4 Hardwarenahe Programmierung**
- 5 Algorithmen**
- 6 Objektorientierte Programmierung**
 - 6.0** Dynamische Speicherverwaltung
 - 6.1** Konzepte und Ziele
 - 6.2** Beispiel: Zahlen und Buchstaben
 - 6.3** Unions
 - 6.4** Virtuelle Methoden
 - 6.5** Beispiel: Graphische Benutzeroberfläche (GUI)
 - 6.6** Ausblick: C++
- 7 Datenstrukturen**