

# Datenbankzugriffssprachen



# Prüfungsleistung

- Zusammen mit Prüfungsleistung für Prof. Klein
- Dauer 60 Minuten
- Zwei Teile:
  - Theorie (max  $\frac{1}{4}$  des Gesamtumfangs)
  - Praktische Arbeit am Rechner (SQL)

# Wann sind Datenbanken „notwendig“ und wann nicht?

- Einfache Projekte erfordern oft keine Datenbank
- Zunehmende Komplexität macht Datenbankeinsatz hilfreich
- Diesen Einfluss zunehmender Komplexität soll diese Einführung aufzeigen

# Datenverwaltung mit Dateien

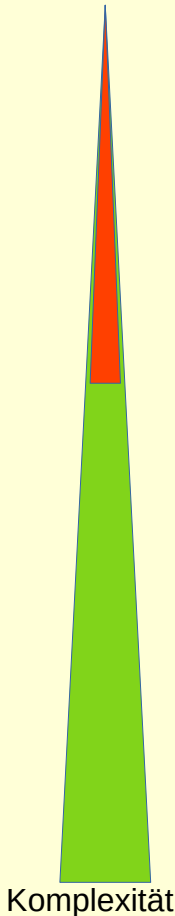
- Motivation: Geht Programmieren auch „schlank“?
  - Weil: Ressourcen sparen, wenn notwendig
  - Oft sind gerade kleine Systeme (IoT) nicht leistungsfähig genug für eine Datenbank

# Challenge1:

## Persistente Datenhaltung

```
main() {  
  
    persistent int x;  
  
    printf(„x = %d\\n“, x);  
  
    x = 23;  
  
}
```

Das ist C-Pseudocode. Im Grunde geht es darum, dass die Variable x nach dem zweiten Start des Programmes immer initialisiert ist.

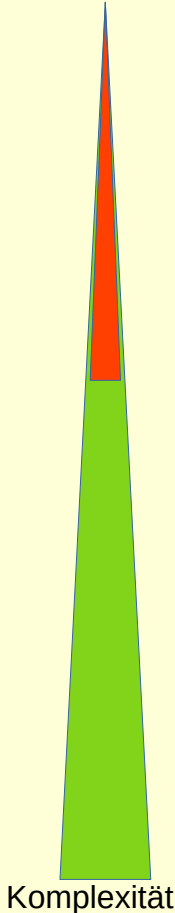


Komplexität

# Persistente Datenhaltung

```
main() {  
    int x;  
  
    x = getX();  
    printf("x = %d\n", x);  
    x = 23;                //aktives Ändern  
    saveX();  
}
```

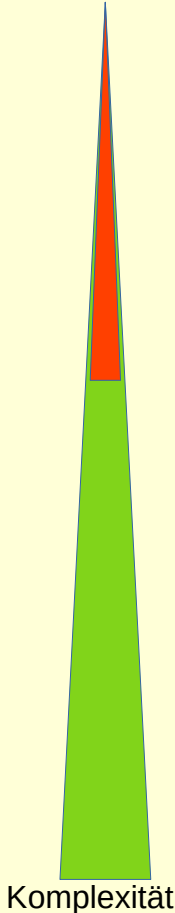
Das ist kein Pseudocode mehr, sondern „echtes“ C und zeigt eine Implementierungsmöglichkeit.



# Persistente Datenhaltung

```
void saveX(int x) {  
    FILE *f;  
  
    f = fopen("/some/path/x.dat", "w");  
    fprintf(f, "%d", x);  
    fclose(f);  
    return;  
}
```

So könnte es z.B. konkret umgesetzt werden.



# Persistente Datenhaltung: Problem Datenrepräsentation

*x.dat (ARM32)*

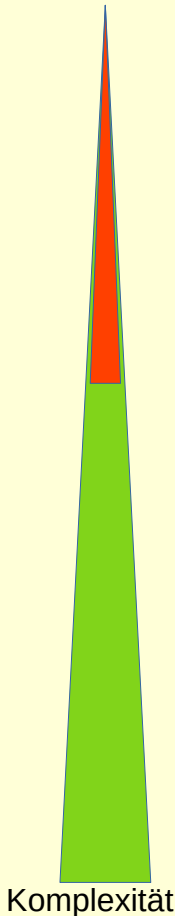
*23 00 00 00*      *32 Bit Little Endian*

*x.dat (AMD64)*

*23 00 00 00 00 00 00 00*      *64 Bit Little Endian*

*x.dat (MIPS32)*

*00 00 00 23*      *32 Bit Big Endian*





# Persistente Datenhaltung: Problem Shared Data

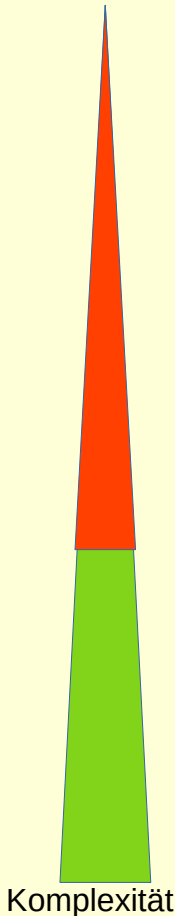
```
getX();  
x=23;  
saveX();
```

Prozess 1

```
getX()  
x=42;  
saveX();
```

Prozess 2

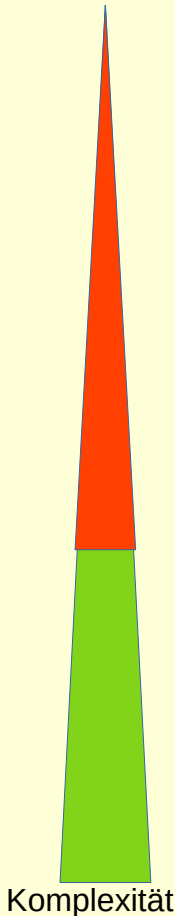
Prozess 1 und Prozess 2 laufen parallel und greifen auf eine gemeinsame Variable x (gleicher Speicherbereich) zu. Das erfordert Synchronisation.



Komplexität

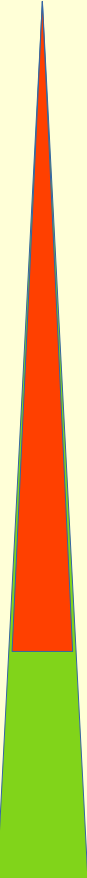
# Zwischenfazit

- Wir sehen: Schon einfachste Anforderungen bedingen schnell Mehraufwand
- Das betraf hier im Beispiel:
  - Datenrepräsentation (Little vs Big Endian)
  - Shared Data



# Weitere Problemfelder: Performance

- Zugriffe auf Datenträger dauern lange
- Mitunter ist aber eine hohe Commit-Frequenz notwendig, um Änderungen sichtbar werden zu lassen
- Lösung: Im Prinzip Caches – aber das ergibt Probleme, das Cache und Datenträger keine unterschiedlichen Informationen haben sollen

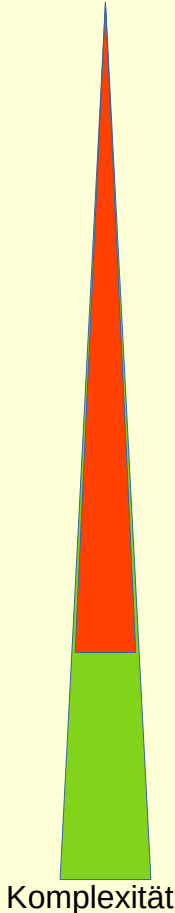


# Performance & Caches

Bsp:

```
x = GetX // Datenträger: x=23 Cache: x=23  
x = 42   // Datenträger: x=23 Cache: x=42 → Erfordert Abgleich  
writeX() // Datenträger: x=42 Cache: x=42
```

So etwas wird in komplexeren Fällen schnell schwer manuell zu verwalten

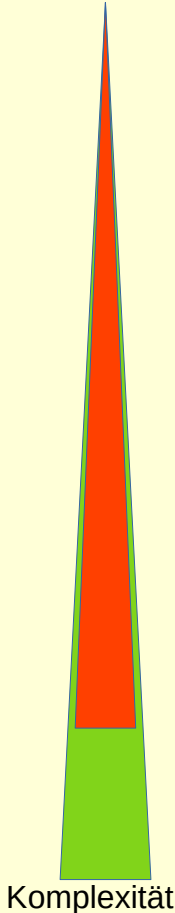


# Komplexität: Skill Level „Very Easy“

Mo: Spaghetti  
Di: Döner  
Mi: Rührei  
Do: Spaghetti  
Fr: Seitan Nuggets  
Sa: Döner  
So: Knäckebrot

Mo: Spaghetti  
Di: Spinat  
Mi: Rührei  
Do: Spaghetti  
Fr: Seitan Nuggets  
Sa: Möhren  
So: Knäckebrot

Ein Speiseplan in zwei Varianten



# Komplexität: Skill Level „Still Easy“

## **Studierende**

300001: Maier

300002: Johnson

300003: Lin Pen

300004: Gurion

## **Vorlesung**

1: Reich werden in 10 Minuten

2: Programmieren in C

## **Teilnahme**

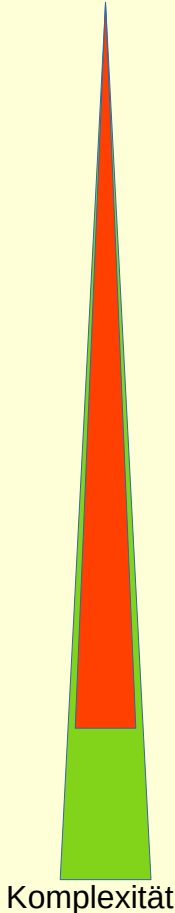
300002: 1

300003: 1

300004: 1

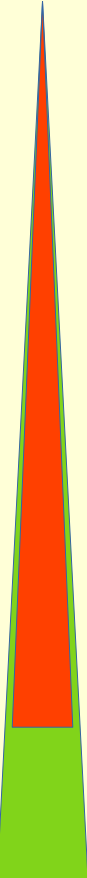
300001: 2

Universitäts - Mini-Modell: Vorlesungen, Studierende und Teilnahme



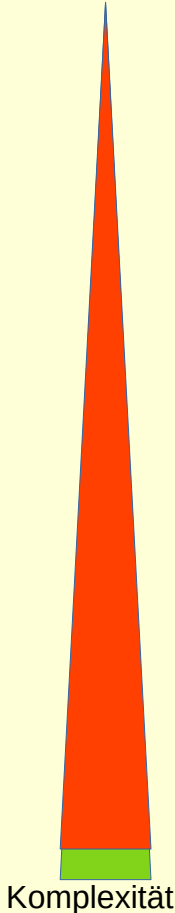
# Komplexität beherrschen?

- Antwort:
  - Alles noch (prinzipiell) „lösbar“
  - Aber: Der Aufwand steigt schon schnell an
- Ein Aktualisieren des Speiseplanes nur unter Verwendung von Dateien geht noch, beim Unimodell wird es schon schwerer, da man da sicherstellen muss, dass nur existierende Matrikelnummern in der Teilnahmeliste erscheinen.



# Weiteres Problem: Redundanz & Konsistenz

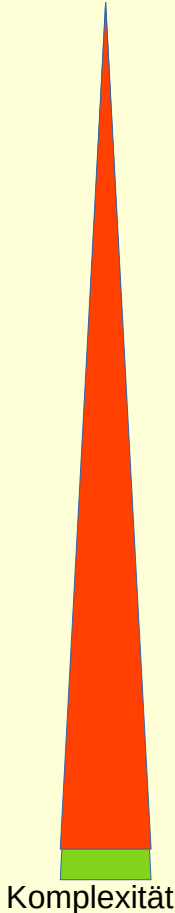
- Ich möchte von meine Daten „Sicherungskopien“ haben
- Diese müssen konsistent sein
- Diese müssen gefunden werden





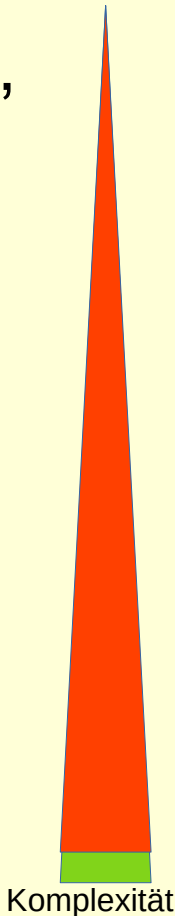
# Atomarität

- Datenaktionen müssen atomar sein
- Bsp: Reisebüro
  - Eine Buchung für eine Pauschalreise benötigt eine Flug- und eine Hotelreservierung
  - Beides muss klappen, sonst ist die Buchung wertlos



# Schnittstellen

- Definierte Schnittstellen werden spätestens dann wichtig, wenn ich Daten außerhalb meines Administrationsbereiches erreichen will.



# Fazit

- Wir hatten nur wenige einfache „Problemchen“:
  - Persistente Datenhaltung
  - Performancesicherung
  - einfache Komplexität
  - Atomarität
  - Schnittstellen
- Je komplizierter es wird, desto mehr ergibt sich der Wunsch nach Systemroutinen, die diese Verwaltung übernehmen →  
**Datenbanken können dies leisten.**

# Fazit(2)

- Datenbanken sind hilfreich, aber **kein Wundermittel** zur schnellen Softwareentwicklung
- Es gilt immer abzuwägen: Was muss ich erreichen und was **brauche** (nicht will!) ich dazu wirklich?
- Eine Datenbank ist also kein “muss“

# Fazit: Ein Beispiel contra Datenbank

- Load2Net – Lastmess-System
- Lasterfassung in C
- Visualisierung in Perl/PHP
- Naheliegende Kopplung mit MySQL **viel zu langsam**, daher „manuell“ über TCP/IP - Kommunikation gelöst



# Fazit: Ein Beispiel pro Datenbank

- Load2Net – Lastmess-System
- Nutzerverwaltung:
  - Namen, Passworte, Rechte
  - Sehr relevant für das Lastmesssystem
- Administrierbar via WebGUI
- Umsetzung mit Datenbank (MySQL) sehr elegant

