

Der Irrsinn ist bei einzelnen etwas Seltenes, aber bei Vielen die Regel! (Nitzsche)

Informatik 2

Bit-Manipulation, **enum**, Makros u.a.



Irene Rothe

Zi. B 241

irene.rothe@h-brs.de

Instagram: irenerothesdesign



Hochschule
Bonn-Rhein-Sieg

Vorlesung_M_BitMani

Das Marie Kondo Prinzip

→ If it doesn't spark joy, get rid of it.

Wenn etwas keinen Spaß macht, ändere es!

If a function or line doesn't spark joy, get rid of it.

Quelle: <https://www.karim-geiger.de/blog/das-marie-kondo-software-design-principle>

Informatik: 2 Semester für Ingenieure

Informatik = Lösen von Problemen mit dem Rechner

✓ Zum Lösen von Problemen mit dem Rechner braucht man **Programmierfähigkeiten** (nur mit Übung möglich): Was ist Programmieren?

✓ Was ist ein Flussdiagramm?

→ **Programmiersprache C:**

- ✓ Elementare Datentypen
- ✓ Deklaration/Initialisierung
- ✓ Kontrollstrukturen: if/else, while, for
- ✓ Funktionen
- ✓ Felder (Strings)
- ✓ Zeiger
- ✓ struct
- ✓ Speicheranforderung: malloc

→ Listen

→ Bitmanipulation

✓ Wie löst der Rechner unsere Probleme? → mit **Dualdarstellung** von Zeichen und Zahlen und mit Hilfe von **Algorithmen**





✓ Ein Beispiel für ein Problem: **Kryptografie**

✓ Sind Rechner auch Menschen? → **Künstliche Intelligenz**

✓ Für alle Probleme gibt es viele Algorithmen. Welcher ist der Beste? → **Aufwand** von Algorithmen



Design der Folien

-  hinterlegt sind alle Übungsaufgaben. Sie sind teilweise sehr schwer, bitte absolut nicht entmutigen lassen! Wir können diese in Präsenz besprechen oder über Fragen im Forum.
-  hinterlegte Informationen und grüne Smileys sind wichtig und klausurrelevant.
- Alles hinter „**Achtung**“ unbedingt beachten!
-  verwende ich, wenn überraschende Probleme auftreten können. Wenn Sie schon programmiererfahrend sind, können das eventuell besonders große Überraschungen für Sie sein, wenn Sie eine andere Sprache als C kennen.
- „Tipp“ benutze ich, um Ihnen einen Weg zu zeigen, wie ich damit umgehen würde.
- „Bemerkung“ in Folien beziehen sich meist auf Sonderfälle, die nicht unbedingt klausurrelevant sind, aber für Sie beim Programmieren eine Bedeutung haben könnten
-  hinter diesem Symbol ist ein Link fürs Anhören bzw. Gucken weiterer Infos

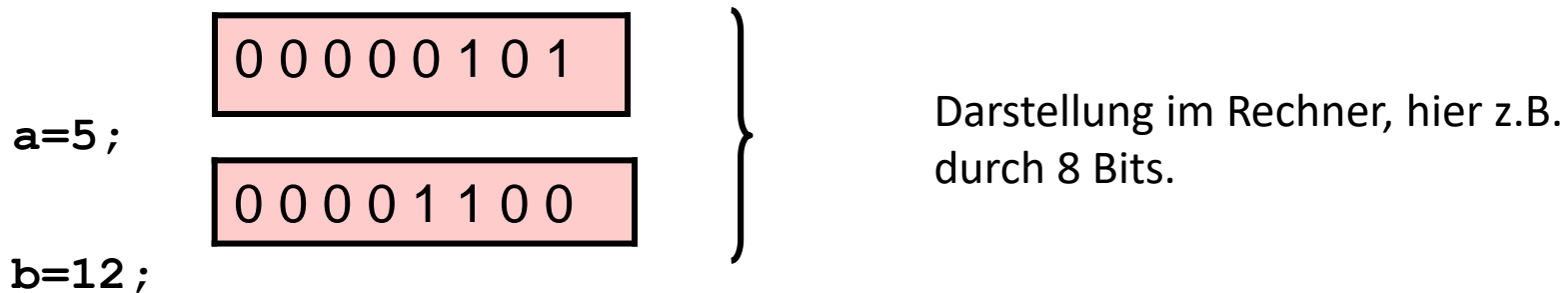
1. Bit-Operationen: logische Operationen, Shift-Operationen
2. Hexa-Darstellung in C
3. Prioritäten von Operatoren
4. Makros
5. enum Typ
6. globale Variablen
7. make-Dateien



1. Bitdarstellung

Ein Bit ist die kleinste Speichereinheit auf einem Computer.

```
unsigned int a,b,c; //neuer Datentyp: ganze Zahlen  
                //ohne Vorzeichen
```



1. Bit-Operatoren

Logische Bit-Operatoren:

- $\&$ binäres UND (bitweise AND)
- $|$ binäres ODER (bitweise OR)
- \wedge Exklusives-ODER
- \sim binäres NICHT (Einer-Komplement)

Shift-Operatoren:

- \ll binäres Links-Shift (Verschiebung)
- \gg binäres Rechts-Shift

1. Logische Bit-Operatoren: Funktionsweise

UND	
$0 \& 0$	0
$0 \& 1$	0
$1 \& 0$	0
$1 \& 1$	1

ODER	
$0 \mid 0$	0
$0 \mid 1$	1
$1 \mid 0$	1
$1 \mid 1$	1

Exklusiv ODER	
$0 \wedge 0$	0
$0 \wedge 1$	1
$1 \wedge 0$	1
$1 \wedge 1$	0

NICHT	
~ 0	1
~ 1	0

1. Logische Bit-Operatoren: Beispiele

```
unsigned int a,b,c;
```

```
a=5;
```

0 0 0 0 0 1 0 1

```
b=12;
```

0 0 0 0 1 1 0 0

```
c=a&b;
```

0 0 0 0 0 1 0 0

```
c=a|b;
```

0 0 0 0 1 1 0 1

```
c=a^b;
```

0 0 0 0 1 0 0 1

```
c=~a;
```

1 1 1 1 1 0 1 0

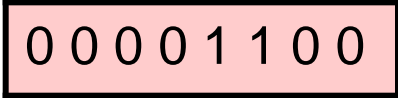
Bemerkung: & nicht verwechseln mit && und | nicht verwechseln mit ||. & und | wirken auf einzelne Bits. 1 && 2 ist nicht 0, aber 1 & 2 ist 0.



1. Shift-Operatoren: Beispiele

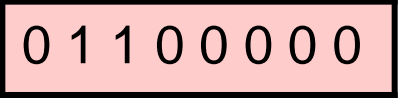
```
unsigned int a,b;
```

```
a=12;
```



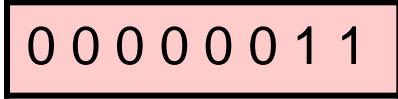
0 0 0 0 1 1 0 0

```
b=a<<3;
```



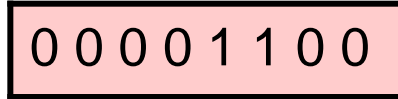
0 1 1 0 0 0 0 0

```
b=a>>2;
```

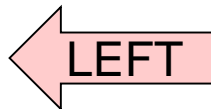


0 0 0 0 0 0 1 1

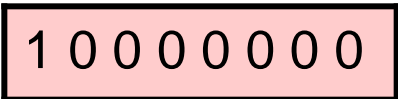
```
a=12
```



0 0 0 0 1 1 0 0



```
b=a<<5
```



1 0 0 0 0 0 0 0

1. Bitoperationen: Vorteile – zum Beispiel effiziente Multiplikation (Division) mit Zweierpotenzen

Verschiebung des Bitmusters um n Stellen nach links (bzw. rechts) entspricht einer Multiplikation (bzw. Division) mit 2^n :

```
unsigned int zahl = 9, product_2hoch3, quotient_2hoch2;
```

```
product_2hoch3 = zahl << 3; // 72 = 9 * (2*2*2)
```

```
quotient_2hoch2 = zahl >> 2; // 2 = 9 / (2*2)
```

UND: wird oft genutzt, um Bits abzufragen oder zu löschen

ODER: wird oft genutzt, um Bits zu setzen



1. Bitoperationen: Anwendung - Löschen von Bits mit &

Verwendung von Bitoperationen als Maske für maschinennahe Programmierung mit ganzen Zahlen:

Beispiel:

```
x = x & ~3; // löscht 0. und 1. Bit
```

3 nennt man oft Maske und definiert das dann wie folgt:

```
#typedef MASK 3
```

Erklärung:

3 entspricht (000...011)₂

~3 entspricht (111...100)₂

Beispiel:

x=7

0 0 0 0 0 1 1 1

~3

1 1 1 1 1 1 0 0

← Maske zum Löschen von
0. und 1. Bit

Ergebnis: **x&~3**

0 0 0 0 0 1 0 0



1. Bitoperationen: Anwendung - Setzen von Bits mit |

Verwendung als Maske für maschinennahe Programmierung mit ganzen Zahlen:

Beispiel:

```
x = x | 2; // setzt 1. Bit
```

Erklärung:

2 entspricht (000...010)_2

Beispiel:

x=5

0 0 0 0 0 1 0 1

2

0 0 0 0 0 0 1 0

← Maske zum Setzen des 1. Bits

Ergebnis: **x | 2**

0 0 0 0 0 1 1 1



1. Bitoperationen: Anwendung - Abfragen von Bits mit &

Wie folgt können zum Beispiel die Belegung von Bits abgefragt werden:

```
#typedef MASK 2
if((variable & MASK) == 0){
    printf("1.Bit ist OFF");
}
else{
    printf("1.Bit ist ON");
}
```


1. Bitoperationen: Anwendung - Umschalten von Bits mit ^

Wie folgt können zum Beispiel Bit verändert werden:

```
//mit ^ können Bits invertiert werden
//durch zweifache Invertierung mit derselben Maske
#define Mask 2
Schalter = Schalter ^ Mask;//Schalter ^= Mask
//kann das ursprüngliche Bitmuster zurück erhalten
//werden, dh der Schalter steht wieder wie vorher
Schalter = (Schalter ^ Mask) ^ Mask;
```

1. Bitoperationen: Beispiel - Umwandlung Binärzahl in Dezimalzahl

```
unsigned int getbinary(){
1   int c;
2   unsigned int wert=0;
3   unsigned int bit=0;
4   c=getchar();//liest Buchstaben aus der Eingabe auf int

5   //falls Leerzeichen oder Tab...weg damit
6   while(c==' '||c=='\t'){
7       c=getchar();//...weiter zum nächsten Zeichen
8   }
9   //Schleife liest bitweise Binärzahlen ein und wandelt sie in Integer
9   while(c=='0' || c=='1'){
10      //getchar liest char ein, Minus 48 ('\0') macht char zu int
11      bit=c-'0';
12      //schon eingelesene Werte eins nach links schieben und mit
13      //ODER bit setzen (an wert anhaengen)
14      wert=(wert<<1)|bit; //hier mache ich durch Bitmanipulation aus den einzelnen Bits
                          // eine richtige int-Zahl; da wert ein integer ist werden die
                          // Bits als Dualzahl und nicht ASCII codiert
15      c=getchar();//naechstes Zeichen lesen
16  }
    return wert;
}
```



1. Bitoperationen: Beispiel - Microcontroller

```
//dies sind Microcontrollerbefehle, also nicht direkt für C relevant
//soll nur motierendes Beispiel sein
#include <avr/io.h>
#include "utils.h",
int main(){
    uint16_t cnt;
    //enable PB5 as output
    DDRB |= _BV(PB5);
    //Beep, Beep
    cnt = 0;
    while (1) {
        if ( cnt++ >= 1000 ) cnt = 0;
        if ( cnt < 800 ) PORTB ^= _BV(PB5);
        warten_ms(1);
    }
    return 0;
}
```



1. ✓ Bit-Operationen: logische Operationen, Shift-Operationen
2. Hexa-Darstellung in C
3. Prioritäten von Operatoren
4. Makros
5. enum Typ
6. globale Variablen
7. make-Dateien



2. Hexa-Darstellung unter C

Eine Hexa-Zahl beginnt in C mit **0x**.

Beispiel: 127 wird als Hexa-Zahl in C wie folgt dargestellt: **0x7F**

Beispiel (aus Vorlesung Mikrocontroller 3.Semester):

```
#define NOT !
if ( NOT (P1 & 0x10) ) { // Taste SW1 gedrueckt ?
    LED1 = ON; // rote LED an
}
else if ( NOT (P1 & 0x20) ) { // Taste SW2 gedrueckt ?
    LED2 = ON; // gelbe LED an
}
else if ( NOT (P1 & 0x40) ) { // Taste SW3 gedrueckt ?
    LED3 = ON; // gruene LED an
}
else {
    LED1 = LED2 = LED3 = OFF; // alle LEDs ausschalten
}
```



3. Prioritäten von Operationen

Operator	Verarbeitung
() Funktionsaufruf	
() Klammerung	
. Komponentenzugriff	
[] Feldelementzugriff	L->R (von links nach rechts)
-> Komponentenzugriff	L->R
Cast-Operator	L->R
sizeof	L<-R
* Pointer, & Adresse	L<-R
++ --	L<-R
+ - Vorzeichen	L<-R
! Negation	L<-R
~ Bitkomplement	L<-R
* / % Multiplikation, Division, Modulo	L->R
+ - Addition, Subtraktion	L->R
>> << Shift	L->R
< > <= >= Vergleiche	L->R
== !=	L->R
& Bitoperation	L->R
^ Bitoperation	L->R
Bitoperation	L->R
&&	L->R
	L->R



4. Makro: **define** – sehr praktisch

Makros werden wie folgt definiert: **#define name ersatztext**

Der Präprozessor ersetzt beim Compilieren alle **name**-Variablen durch **ersatztext**. Dieses Makro kann man benutzen für Konstanten und Befehlsfolgen.

Beispiele:

```
#define pi 3.14159
#define AND &&
#define NOT !
#define FOREVER while(1)
#define MAX(x,y) ((x)>(y)?(x):(y))
```

Vorteile:

- Gute Lesbarkeit
- Änderungsfreundlichkeit



1. ✓ Bit-Operationen: logische Operationen, Shift-Operationen
2. ✓ Hexa-Darstellung in C
3. ✓ Prioritäten von Operatoren
4. ✓ Makros
5. Aufzähltyp enum
6. globale Variablen
7. make-Dateien



5. Aufzähltyp **enum**

Der Aufzähltyp **enum** kann gut genutzt werden beim Erzeugen einer Aliasliste:
→ statt vieler Konstanten per **#define** wird die **enum** Anweisung verwendet, um einen ganzen Satz von Konstanten zu definieren.

Beispiel 1:

```
enum noten_genauer {sehrgut=1, gut=2, befr=3, genuegend=4, unbefr=5};
```

Beispiel 2:

```
enum bool {false = 0, true = 1};
```

Bemerkung: Wenn eine Variable mit **enum** definiert wird, behandelt sie der Compiler als Integerzahl. Sie kann dann jeden Wert annehmen, nicht nur die in **enum** festgelegten.

6. Globale/lokale Variablen

Variablen können **außerhalb** (global) und **innerhalb** (lokal) von Funktionen definiert werden.

Globale Variablen können überall verwendet werden.

Lokale Variablen können nur in derselben Funktion verwendet werden.

Bemerkung von Irene Rothe: In der Softwareentwicklung gelten globale Variablen als hässlich und unnötig. Sie verursachen nur Ärger!



7. Projekterstellung und make-Files

In Dev-Cpp ein Projekt anlegen -> siehe make File

SS2021: Überspringen?



Sonstige Bemerkungen für die Zukunft 1

- **defines** für Konstanten sind praktisch, da sie so nur an **einer Stelle** anzupassen sind
- Lokale Variablen sind immer besser. Nur wenn Zeit und Platz wirklich sehr wichtig sind (z.B. bei Mikrocontrollern), können **globale Variablen** bei Funktionsaufrufen vernünftig sein (siehe Programmierung von Mikroprozessoren).
Bemerkung: lokal geht vor global bei gleich Benamsung!
- Testausgaben mit **printf** sind sehr nützlich.
- **Gotos** können bei häufigen Abstürzen und daraus nötigen Neustarts von Programmen benutzt werden, sonst nicht!!!
- Schlimmste Fehlerquellen:
 - Zugriff auf **falschen Speicher**
 - **Logikfehler** (falsche Bedingungen testen)
 - Nichtberücksichtigung **aller** Möglichkeiten (z.B. bei **if/else**)
 - Ungeplante Gleichzeitigkeit (**Überlastung**)



Sonstige Bemerkungen für die Zukunft 2

How to consume CPU time?

Wie erhalte ich eine Verzögerung?

```
void delay(int ticks) {  
    int i;  
    for (i=0;i<ticks;i++) ;  
}
```

Scanf ist keine tolle Funktion

- `%i` und `%d` tun unterschiedliche Dinge bei `int`,
`%i` nimmt auch hexadezimal an, wenn beginnend mit `0x` und
octal, wenn beginnen mit `0`
Beispiel: `033` mit `%i` ist $27(3*8+3)$, mit `%d` `33`
Bemerkung: `%i` und `%d` gleich bei `printf`
- `fflush` arbeitet nicht für alle BS und Compiler
- Super beschrieben hier: http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/004_c_ein_ausgabe_001.htm

Haben Sie noch Fragen?

→ Bitte schicken Sie Fragen an irene.rothe@h-brs.de

