3.2 C-Datentypen/Ganze Zahlen und Bit-Bearbeitung

3.2.1 Übersicht

Ganzzahlige Datentypen sind in der Sprache C an vielen Stellen wichtig. Sie haben folgende Eigenschaften:

- Es gibt Ganzzahlen mit Implementations-abhängiger Länge wie `int`, `short` und `long`. Sie belegen auf kleinen System wenige Bits und auf großen Systemen viele Bits. Seit dem Standard C99 gibt es außerdem Ganzzahlen mit fester Länge wie `int16_t`, `int32_t` und `int64_t`.
- Wie für Gleitkommazahlen sind auch für Ganzzahlen die arithmetischen Operatoren `+`, `-`, `*` und `/` erlaubt. Bei der Division handelt es sich jedoch um eine reine Ganzzahldivision (79/10 gibt 7). Den Divisionrest bekommt man mit dem Modulo-Operator `%`.
- Mathematische Vergleiche sind auch für Ganzzahlen erlaubt. Wenn man links und rechts verschiedene Datentypen hat, ist Vorsicht geboten (siehe Kapitel Typkonversion).
- Die Ergebnisse der Vergleichsoperationen (`<`, `<=`, `>`, `>=`, `==` und `!=`) sind vom ganzzahligen Datentyp `int`.
- Die Ergebnisse vieler Funktionsaufrufe sind vom ganzzahligen Datentyp `int`. Bei einigen dieser Funktionen ist die zurückgegebene Ganzzahl nicht mathematisch zu verstehen, sondern hat eine andere Bedeutung (z. B. `EOF` bei `getchar()`).
- Für Speicher-Adressen gibt es in C besondere Datentypen. Sie lassen sich aber in Ganzzahlen umwandeln (umgekehrt auch).
- Der Index eines Arrays (s. u.) ist eine Ganzzahl.
- Mit dem `sizeof()`-Operator kann man die Größe einer Variablen oder eines Datentyps in Byte bestimmen. Der Wert ist eine Ganzzahl mit dem Datentyp `size_t`.
- Es gibt mehrere Operatoren für die Bit-Bearbeitung von Ganzzahlen. Einzelne Bits einer Ganzzahl können gezielt gesetzt, gelöscht, invertiert werden. Außerdem können alle Bits wie bei einem Schieberegister in der gleichen Richtung verschoben werden.

3.2.2 Ganzzahlige Datentypen mit fester Länge

Die elementaren Datentypen von C sind so angelegt, dass sie mit der Datentechnik mitwachsen. Während die ersten C-Compiler für PCs dem Datentyp `int` einen Speicherplatz von 16 Bit gaben, sind es bei heutigen PCs 32 oder 64 Bit.

In manchen Fällen — besonders bei der hardwarenahen Programmierung oder auch bei der Netzwerkprogrammierung — ist es hilfreich, darüberhinaus Datentypen mit fester Läenge zu haben. Im C99-Standard ist dies berücksichtigt. Mit der Einbindung der Datei `<stdint.h>` stehen folgende weitere Typen zur Verfügung: In den genannten Fällen (hardwarenahe Programmierung, Netzwerkprogrammierung) sind die oberen vier Typen (also die vorzeichenlosen Typen) erfahrungsgemäß meistens die bessere Wahl.

3.2.3 Platzhalter für Datentypen mit fester Länge

Für die Aus- und Eingabe von Variablen dieser neuen Datentypen kann man die bisherigen Platzhalter %i, %u usw. nicht ohne weiteres verwenden. Deshalb gibt es in C99 weitere Platzhalter, die in der Headerdatei `<inttypes.h>` definiert sind. Tabelle 2 zeigt die Platzhalter für die Ausgabe mit `printf`, in Tabelle 3 sind die Platzhalter für die Eingabe mit `scanf` zu finden.

Hier ist ein Beispiel für die Anwendung der neuen Platzhalter (`src/platzhalter.c`):

Datentyp	Länge in Bit	Vorzeichen	Minimum	Maximum
`uint8_t`	8	nein	`0`	`UINT8_MAX`
`uint16_t`	16	nein	`0`	`UINT16_MAX`
`uint32_t`	32	nein	`0`	`UINT32_MAX`
`uint64_t`	64	nein	`0`	`UINT64_MAX`
`int8_t`	8	ja	`INT8_MIN`	`INT8_MAX`
`int16_t`	16	ja	`INT16_MIN`	`INT16_MAX`
`int32_t`	32	ja	`INT32_MIN`	`INT32_MAX`
`int64_t`	64	ja	`INT64_MIN`	`INT64_MAX`

Tabelle 1: Datentypen mit fester Länge

Datentyp	dezimal	oktal	hexadezimal
`uint8_t`	PRIu8	PRIo8	PRIx8
`uint16_t`	PRIu16	PRIo16	PRIx16
`uint32_t`	PRIu32	PRIo32	PRIx32
`uint64_t`	PRIu64	PRIo64	PRIx64
`int8_t`	PRId8, PRIi8	—	—
`int16_t`	PRId16, PRIi16	—	—
`int32_t`	PRId32, PRIi32	—	—
`int64_t`	PRId64, PRIi64	—	—

Tabelle 2: Platzhalter für die neuen Datentypen und `printf`

Datentyp	dez., okt., oder hex.	dezimal	oktal	hexadezimal
`uint8_t`	—	`SCNu8`	`SCNo8`	`SCNx8`
`uint16_t`	—	`SCNu16`	`SCNo16`	`SCNx16`
`uint32_t`	—	`SCNu32`	`SCNo32`	`SCNx32`
`uint64_t`	—	`SCNu64`	`SCNo64`	`SCNx64`
`int8_t`	`SCNi8`	`SCNd8`	—	—
`int16_t`	`SCNi16`	`SCNd16`	—	—
`int32_t`	`SCNi32`	`SCNd32`	—	—
`int64_t`	`SCNi64`	`SCNd64`	—	—

Tabelle 3: Platzhalter für die neuen Datentypen und `scanf`

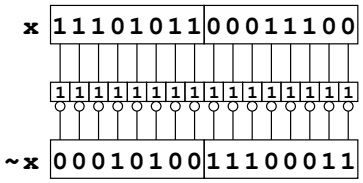

Abbildung 1: NICHT-Verknüpfung

```c
#include <stdio.h>
#include <stdint.h>    /* fuer int16_t */
#include <inttypes.h> /* fuer PRI*16 und SCN*16 */
int main(void)
{
    uint16_t x;
    printf("Zahl eingeben (dezimal):");
    scanf("%" SCNu16, &x);
    printf("Die Zahl lautet (oktal):%" PRIo16 "\n", x);
    return 0;
}
```

3.2.4 Bit-Operatoren

Es gibt zwei Gruppen von Bit-Operationen in C: `~`, `&`, `|` und `^` verknüpfen alle Bits einer Ganzzahl. Dabei sind aber alle Bits dieser Ganzzahl voneinander unabhängig. `>>` und `<<` sind Schiebe-Operatoren. Hier nun die Operatoren einzeln:

a) Der Operator `~` entspricht der NICHT-Verknüpfung. Das Bitmuster von `~x` ist das invertierte (negierte) Bitmuster von `x` (Abbildung 1).

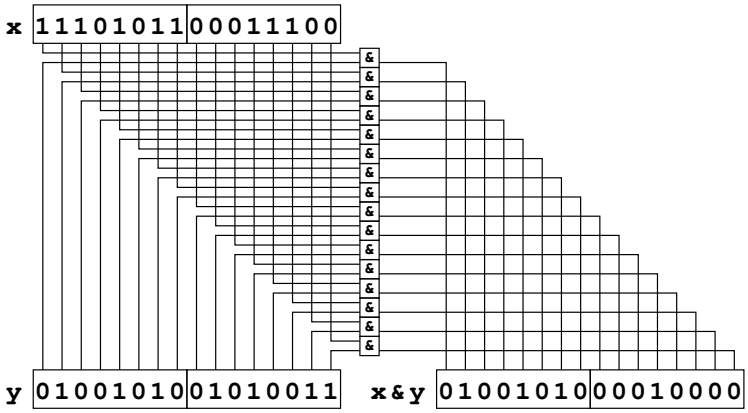

Abbildung 2: UND-Verknüpfung

b) Der Operator `&` entspricht der UND-Verknüpfung. Das Bitmuster von `x&y` entsteht, indem jedes Bit von `x` mit dem entsprechenden Bit `y` UND-verknüpft wird (Abbildung 2). Der Operator `|` entspricht der ODER-Verknüpfung. Das Bitmuster von `x|y` entsteht, indem jedes Bit von `x` mit dem entsprechenden Bit `y` ODER-verknüpft wird (Abbildung 3). Der Operator `^` entspricht der EXOR-Verknüpfung. Das Bitmuster von `x&y` entsteht, indem jedes Bit von `x` mit dem entsprechenden Bit `y` EXKLUSIV-ODER-verknüpft wird (Abbildung 4).

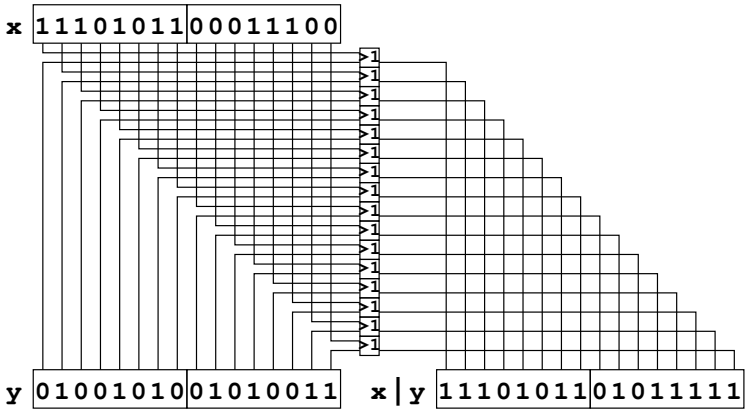

Abbildung 3: ODER-Verknüpfung

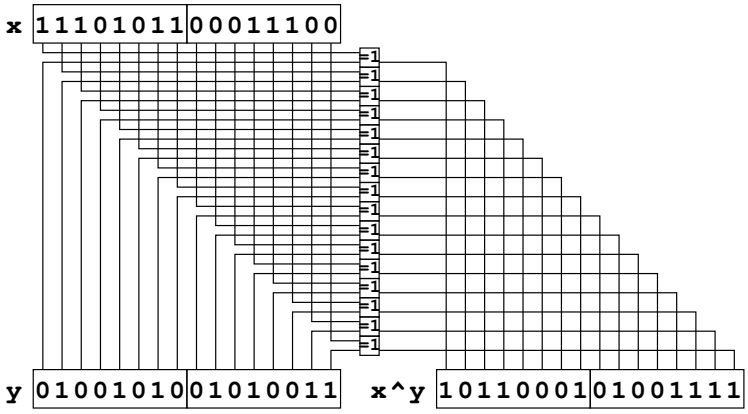

Abbildung 4: EXKLUSIV-ODER-Verknüpfung

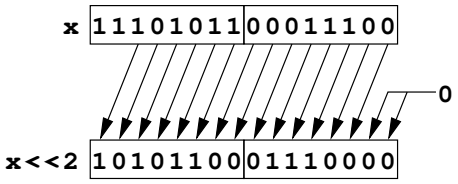

Abbildung 5: Verschiebung nach links

c) Mit dem Operator << wird das Schieben nach links veranlasst. Das Bitmuster von `x<<n` entsteht, indem jedes Bit von `x` um `n` Plätze nach links verschoben wird. Die `n` vordersten Bits gehen dabei verloren. Die `n` hintersten Bits werden mit Null-Bits aufgefüllt (Abbildung 5).

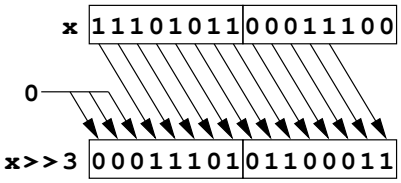

Abbildung 6: Verschiebung nach rechts

d) Genaus wird mit dem Operator >> das Schieben nach rechts veranlasst. Das Bitmuster von `x>>n` entsteht, indem jedes Bit von `x` um `n` Plätze nach rechts verschoben wird. Die `n` hintersten Bits gehen verloren, während die `n` vordersten Bits mit Null-Bits aufgefüllt werden (Abbildung 6).

3.2.5 Dual-Ein- und Ausgabe als Hilfsmittel

Im folgenden Programm kommen zwei kurze Funktionen zum Einsatz, mit denen man eine Variable vom Datentyp `uint16_t` einlesen und ausgeben kann.

```
1  #include <stdio.h>
2  #include <stdint.h>
3  void printdual(uint16_t wert)
4  {
5      int pos;
6      for(pos=0; pos<16; ++pos)
7      {
8          if(wert & (1<<15))
9              putchar('1');
10         else
11             putchar('0');
12         wert=wert<<1;
13     }
14 }
15 /***************************/
16 uint16_t scandual(void)
17 {
18     uint16_t erg=0;
19     int pos, ch;
20     pos=0;
21     ch=getchar();
22     while((ch=='0'||ch=='1')&&(pos<16))
23     {
24         erg=erg<<1;
25         ++pos;
26         if(pos<16)
27         {
28             if (ch=='1')
29                 ++erg;
30             ch=getchar();
31         }
```

```
32      }
33      if((ch!='\n')&&(ch!=EOF))
34          while(getchar()!='\n'){}
35      return erg;
36  }
37  /***************************/
38  int main(void)
39  {
40      uint16_t x;
41      printf("Zahl x eingeben: ");
42      x=scandual();
43      x=~x;
44      printf("Ergebnis fuer ~x:");
45      printdual(x);
46      putchar('\n');
47      return 0;
48  }
```

3.2.6 Setzen einzelner Bits

Mit dem Operator | kann man einzelne Bits setzen:

```
1   a=a|8; /* 00001000, setzt Bit Nr. 3 */
```

Die Zahl 8 ist hier eine sogenannte Bitmaske: Alle Bits, die in der Maske gleich eins sind, werden gesetzt (wie bei einer Druckmaske). Alle anderen Bits bleiben unverändert. Häufig wird die Operation so abgekürzt:

```
1   a|=8; /* 00001000, setzt Bit Nr. 3 */
```

Man kann auch mehrere Bits gleichzeitig setzen:

```
1   a|=9; /* 00001001 setzt Bits Nr. 3 und Nr. 0 */
```

Zur besseren Verdeutlichung, welche Bits gesetzt werden, kann man statt der Dezimalschreibweise auch die Hexadezimal- oder die Oktalschreibweise verwenden (seit C23 endlich auch die Dualschreibweise):

```
1   a|=36;   /* 0000000000100100 setzt Bits Nr. 5 und Nr. 2 */
2   a|=0x12;/* 0000000000100100 setzt Bits Nr. 5 und Nr. 2 */
3   a|=042; /* 0000000000100100 setzt Bits Nr. 5 und Nr. 2 */
4   a|=0b100100; /* setzt Bits Nr. 5 und Nr. 2 */
```

In der Hardware-nahen Programmierung ist es üblich, dass man die Bits einzeln nennt:

```
1   a|=(1<<5)|(1<<2);/*..0100100 setzt Bits Nr. 5 u. Nr. 2 */
```

Mit `1<<5` bekommt man die Zahl 1, jedoch um 5 Bit nach links verschoben. Das entspricht dem Bitmuster 00100000, also der Zahl 32. Mit `1<<2` bekommt man das Bitmuster 00000100, also die Zahl 4. Beides zusammen entspricht der Zahl 36.

Wenn die einzelnen Bits jetzt noch Namen haben, kann das Programm dadurch verständlicher werden:

```
1   #define TOR1_AUF 5
2   #define TOR2_ZU  2
3       a|=(1<<TOR1_AUF)|(1<<TOR2_ZU);
```

3.2.7 Löschen einzelner Bits

Mit dem Operator & kann man einzelne Bits löschen.

```
1    a=a&65527; /* 1111111111110111, loescht Bit Nr. 3 */
```

Alle Bits, die in der Maske gleich null sind, werden gelöscht. Alle anderen Bits bleiben unverändert. Auch hier kann man die Operation abgekürzt darstellen:

```
1    a&=65527; /* 1111111111110111, loescht Bit Nr. 3 */
```

Und man kann mehrere Bits gleichzeitig löschen:

```
1    a&=65526;  /* 1111111111110110, loescht Bits Nr. 3 und Nr. 0 */
```

Zur besseren Verdeutlichung, welche Bits geloescht werden, kann man die Hexadezimal- oder die Oktalschreibweise verwenden:

```
1    a&=0177767;/* 1111111111110110, loescht Bits Nr. 3 und Nr. 0 */
2    a&=0xFFF7; /* 1111111111110110, loescht Bits Nr. 3 und Nr. 0 */
3    a&=0b1111111111110110; /* seit C23 erlaubt */
```

Viel besser geeignet ist jedoch die invertierte Bitmaske:

```
1    a&= ~9;     /* 1111111111110110, loescht Bits Nr. 3 und Nr. 0 */
2    a&= ~0b1001; /* seit C23 erlaubt */
```

Und für die Hardware-nahe Programmierung kann man die Bits wieder einzeln nennen:

```
1    a&= ~((1<<3)|(1<<0));
```

Dasselbe mit Bit-Namen:

```
1  #define LAMPE_LINKS  3
2  #define LAMPE_RECHTS 0
3    a&= ~((1<<LAMPE_LINKS)|(1<<LAMPE_RECHTS));
```

3.2.8 Invertieren einzelner Bits

Mit dem Operator & (EXOR) kann man einzelne Bits invertieren (=negieren). Die Exklusiv-Oder-Verknüpfung kann als schaltbarer Inverter angesehen werden.

```
1    a=a^18; /* 0000000000100010, invertiert Bits Nr. 5 und Bit Nr. 1 */
```

Alle Bits, die in in der Maske gleich eins sind, werden invertiert. Alle anderen Bits bleiben unverändert. Auch bei dieser Verknüpfung sind weitere Schreibweisen möglich und oft sinnvoll:

```
1    a^=18;
2    a^=022;
3    a^=0x12;
4    a^=0b100010;
5    a^=(1<<5)|(1<<1);
```

3.2.9 Abfrage gesetzter Bits

Mit dem Operator `&` kann man abfragen, ob ein bestimmtes Bit eins ist:

```
1  if(a&16)
2      printf("Bit 4 ist gesetzt.\n");
```

Falls Bit 4 von `a` gesetzt ist, ist das Ergebnis 16 und der `if`-Zweig wird ausgeführt.

Man kann auch abfragen, ob mindestens eins von mehreren Bits gesetzt wurde:

```
1  if(a&24)
2      printf("Bit 4 oder Bit 3 ist gesetzt.\n");
```

Falls Bit 4 von `a` gesetzt ist, ist das Ergebnis 16 und der `if`-Zweig wird ausgeführt. Falls Bit 3 von `a` gesetzt ist, ist das Ergebnis 8 und der `if`-Zweig wird ausgeführt. Falls die Bits 3 und 4 von `a` beide gesetzt sind, ist das Ergebnis 24 und der `if`-Zweig wird ausgeführt.

Wie fragt man nun ab, ob alle Bits der Bitmaske gesetzt wurden?

```
1  if(a&24==24)
2      printf("Bit 4 und Bit 3 sind beide gesetzt.\n");
```

Falls alle Bits der Bitmaske auch in `a` gesetzt sind, ist das Ergebnis gleich der Bitmaske; der `if`-Zweig wird ausgeführt.

3.2.10 Abfrage gelöschter Bits

Mit den Operatoren `~` und `&` kann man abfragen, ob ein bestimmtes Bit null ist:

```
1  if(~a&16)
2      printf("Bit 4 ist geloescht.\n");
```

Falls Bit 4 von `a` gesetzt ist, ist Bit 4 von `~a` gesetzt. Das Ergebnis ist dann 16 und der `if`-Zweig wird ausgeführt.

Bei mehreren Bits funktioniert das entsprechend wie oben:

```
1  if(~a&24)
2      printf("Bit 4 oder Bit 3 ist geloescht.\n");
3  if(~a&24==24)
4      printf("Bit 4 und Bit 3 sind alle geloescht.\n");
```

3.2.11 Durchschalten einzelner Bits

Manchmal muss man ein Bit (oder mehrere) vom Eingang e an den Ausgang a übertragen. Falls alle Bits gemeint wären, wäre das kein Problem:

```
1  a=e;
```

Bei nur einem Bit kann man das mit einer Verzweigung machen; hier im Beispiel wieder mit Bit 4:

```
1  if(e&16)
2      a|=16;
3  else
4      a&=~16;
```

Eine andere Möglichkeit besteht darin, das Bit 4 dann zu kippen, wenn es zwischen a und e verschieden ist:

```
1  a^=(a^e)&16;
```

`a^e` ermittelt alle Bits, die bei a anders sind als bei e. Die Verknüpfung `&16` setzt alle Bits des Ergebnisses auf null – außer Bit 4. Und der Operator `^=` kippt das Bit 4 bei a, falls rechts vom Operator dort eine eins steht.