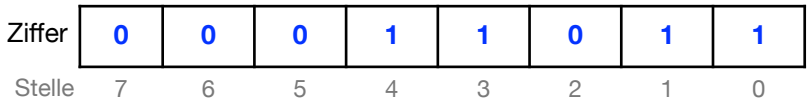
Bitweise Operatoren und Bitfelder in C

Prof. Dr. Malte Weiß

Dieser Artikel erläutert die bitweisen Operatoren in C.

Einführung

Wir wissen bereits, dass Ganzzahlen, mit denen wir in C arbeiten, im Rechner als Binärzahlen gespeichert sind, sich also als Folge von Nullen und Einsen darstellen lassen. Die Zahl $(15)_{10}$ lässt sich beispielsweise binär als $(11011)_2$ darstellen. In Form einer 8-Bit-Zahl vom Datentyp `char` würde die Zahl binär wie folgt gespeichert:



Gelegentlich möchte man einzelne Bits einer Zahl auslesen oder verändern, ohne die anderen Bits zu beeinflussen. Eine Anwendung dafür finden wir in der hardware-nahen Programmierung, wenn einzelnen Bits einer Variable Schalter und Teilwerte repräsentieren. Stellen wir uns z.B. einen Fön vor, deren Zustand wie folgt in einem einzige Byte gespeichert wird:

Im Beispiel steht Bit 7 (ganz links) dafür, ob die Heizung aktiviert ist oder nicht (1 = ja, 0 = nein). Das nächste Bit für die Aktivierung des Lüfters. Solche Schalter, die durch ein Bit gesetzt werden, nennt man auch **Flags**. Im obigen Beispiel folgen vier Bits für die Lüfterstufe. Für diese gibt es 2^4 also 16 unterschiedliche Zustände (Werte 0 bis 15). Die Heizstufe wird durch zwei Bits repräsentiert und besitzt damit 2^2 also 4 mögliche Zustände (Werte 0 bis 3).

Um einzelne Bits zu manipulieren, bietet C binäre Operatoren an, die im nächsten Kapitel besprochen werden. Hilfreich ist hierbei die Notation von Zahlen im Binärformat. Entsprechende Literale lassen sich in C durch ein vorangestelltes `0b` definieren. Im folgenden C-Code wird beispielsweise zweimal die gleiche Zahl definiert, einmal als Dezimal- und einmal als Binärliteral:

```
char ch1 = 15;           // dezimal 15
char ch2 = 0b00011011;   // entspricht dezimal 15
```

Bitweise Operatoren

C bietet die verschiedene binäre Operatoren an. Hier zunächst eine Übersicht:

Operator	Verwendung	Funktion
&	a & b	Anwendung von UND auf die Binärziffern von a und b
|	a | b	Anwendung von ODER auf die Binärziffern von a und b
^	a ^ b	Anwendung von XOR (Exklusives Oder) auf die Binärziffern von a und b
~	~a	Alle Ziffern von a umdrehen. Entspricht der Bildung des Einerkomplements.
<<	a << n	Alle Ziffern um n Stellen nach links schieben. Entspricht der Multiplikation mit 2^n.
>>	a >> n	Alle Ziffern um n Stellen nach rechts schieben. Entspricht der Division durch 2^n.

Binäres UND (&), ODER (|) und XOR (^)

Der binäre UND- und ODER-Operator entspricht der Anwendung der Booleschen Operatoren && und || auf Bitebene. Die Wahrheitstabellen sind dementsprechend identisch:

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

Werden diese Operatoren auf Zahlen angewendet, werden sie Bit für Bit umgesetzt. Wir demonstrieren dies an zwei 8-Bit-Zahlen a = $(51)_{10}$ = $(110011)_2$ und b = $(91)_{10}$ = $(1011011)_2$ zunächst für den **UND**-Operator.

a	0	0	1	1	0	0	1	1	= $(51)_{10}$
b	0	1	0	1	1	0	1	1	= $(91)_{10}$
a & b	0	0	0	1	0	0	1	1	**= $(19)_{10}$**
Stelle	7	6	5	4	3	2	1	0	

Als C-Code:

```
char ch1 = 0b0110011;       // entspricht dezimal 51
char ch2 = 0b1011011;       // entspricht dezimal 91
char result = ch1 & ch2;    // entspricht dezimal 19
```

Das gleiche Beispiel nochmal mit dem **ODER**-Operator:

a	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	$= (51)_{10}$
b	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	$= (91)_{10}$
a | *b*	**0**	**1**	**1**	**1**	**1**	**0**	**1**	**1**	$\mathbf{= (123)_{10}}$
Stelle	7	6	5	4	3	2	1	0	

Als C-Code:

```
char ch1 = 0b0110011;       // entspricht dezimal 51
char ch2 = 0b1011011;       // entspricht dezimal 91
char result = ch1 | ch2;    // entspricht dezimal 123
```

Zum Schluss noch ein Beispiel mit dem **XOR**-Operator:

a	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	$= (51)_{10}$
b	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	$= (91)_{10}$
a ^ *b*	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	$\mathbf{= (104)_{10}}$
Stelle	7	6	5	4	3	2	1	0	

Als C-Code:

```
char ch1 = 0b0110011;       // entspricht dezimal 51
char ch2 = 0b1011011;       // entspricht dezimal 91
char result = ch1 ^ ch2;    // entspricht dezimal 104
```

Invertieren (~)

Der Tilde-Operator dreht jedes Bit um. Das entspricht der Bildung des sog. Einer-Komplements:

x	~x
0	**1**
1	**0**

Wenden wir dies auf unsere Beispielzahl a = $(51)_{10}$ = $(110011)_2$ an, ergibt sich

a	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	= $(51)_{10}$
~a	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**= $(204)_{10}$**
Stelle	7	6	5	4	3	2	1	0	

Wir setzen das ganze in C um und geben die Variablen aus:

```
char ch = 0b110011;
char chInv = ~ch;
printf("%d\n", ch);      // Ausgabe: 51
printf("%d\n", chInv);   // Ausgabe: -52
```

Überraschenderweise wird für `chInv` nicht 204, sondern -52 ausgegeben. Das liegt daran, dass das erste Bit von vorzeichenhaften Variablen das Vorzeichen bestimmt. Ist das Bit 1, gilt die Zahl als negativ. Da negative Zahlen in C im Zweierkomplement[1] und nicht im Einerkomplement gebildet werden, liegt die negative Zahl eins daneben (-52 statt -51).

Die Verwendung eines vorzeichenlosen Datentyps, in diesem Fall unsigned char, behebt dieses Problem:

```
unsigned char ch = 0b110011;
unsigned char chInv = ~ch;
printf("%d\n", ch);      // Ausgabe: 51
printf("%d\n", chInv);   // Ausgabe: 204
```

Bit-Shifting (<< und >>)

Der Operator << schiebt alle Bits einer Zahl um n Stellen nach links, wobei am Ende jeweils als eine 0 eingefügt wird. Der Operator >> geht den umgekehrten Weg und schiebt alle Bits um n Stellen nach rechts, wobei eine führende 0 eingefügt wird.

Hier ein paar Beispiele für die Zahl a = $(51)_{10}$ = $(110011)_2$:

a	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	= $(51)_{10}$
a >> 1	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	= $(25)_{10}$
a >> 2	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	= $(12)_{10}$
a << 1	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	= $(102)_{10}$
a << 2	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	= $(204)_{10}$
Stelle	7	6	5	4	3	2	1	0	

Der dazugehörige C-Code sieht so aus:

```
unsigned char a = 0b110011;   // entspricht dezimal 51
printf("%d\n", a);            // Ausgabe: 51
printf("%d\n", a >> 1);       // Ausgabe: 25
```

---

[1] Beim Zweierkomplement wird zusätzlich zur Invertierung der Bits noch 1 auf die Zahl addiert.

```
printf("%d\n", a >> 2);       // Ausgabe: 12
printf("%d\n", a << 1);       // Ausgabe: 102
printf("%d\n", a << 2);       // Ausgabe: 204
```

Mathematisch gesehen entspricht ein Rechts-Shift um eine Stelle einer ganzzahligen Division durch 2, wie man auch an den Ausgaben des Codes sehen kann. Ein Links-Shift um eine Stelle entspricht einer Multiplikation mit 2. Bit-Shifting ist damit ein schneller Weg für binäre Division und Multiplikation, was in frühen Computerspielen oft als leistungssteigernde Maßnahme ausgenutzt wurde.

Das ist im Dezimalsystem übrigens genauso: Ein Verschieben der Zahl 1234 um eine Stelle nach rechts (123) entspricht einer ganzzahligen Division durch die Basis 10. Ein Verschieben um eine Stelle nach links (12340) entspricht einer Multiplikation mit der Basis 10.

Anwendung

Die oben genannten Operatoren lassen sich einsetzen, um gezielt Bits von Zahlen auszulesen und zu verändern. Im folgenden werden gängige Anwendungen beschrieben. Wir werden dazu auch auf das oben erwähnte Beispiel eingehen, das den Zustand eines Föns beschreibt.

Bedeutung	Heizung aktiviert?	Lüfter aktiviert?	Lüfterstufe (0-15)				Heizstufe (0-3)	
Stelle	7	6	5	4	3	2	1	0

Ein Bit prüfen

Um zu prüfen, ob ein Bit einer Zahl gesetzt ist, können wir das entsprechende Bit über eine UND-Maske prüfen. Ist das Ergebnis dann ungleich 0, ist das Bit gesetzt. Hier ein Beispiel, bei dem wir annehmen, dass es eine Variable `ch` vom Typ `char` gibt.

Der folgende Code prüft ob das dritte Bit von rechts gesetzt ist. Dafür wird die Variable über einen UND-Operator mit dem gesuchten Bit verknüpft.

```
unsigned char bitSet = (ch & 0b100) > 0;
```

Hier zwei Beispiele für konkrete Ausprägungen von `ch`:

ch	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	$= (51)_{10}$
0b100	0	0	0	0	0	**1**	0	0	
ch & 0b100	0	0	0	0	0	**0**	0	0	$\mathbf{= (0)_{10}}$
Stelle	7	6	5	4	3	2	1	0	

ch	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	$= (51)_{10}$
0b100	0	0	0	0	0	**1**	0	0	
ch & 0b100	0	0	0	0	0	**1**	0	0	$\mathbf{= (4)_{10}} \neq 0$ *(wahr)*
Stelle	7	6	5	4	3	2	1	0	

Der &-Operator kann also genutzt werden, um ein Bit zu **isolieren**, d.h. um alle Bits auf 0 zu setzen, die uns nicht interessieren.

Der obige Code lässt sich noch vereinfachen, da ein Wahrheitswert in C genau dann wahr ist, wenn er ungleich 0 ist:

```
unsigned char bitSet = ch & 0b100;
```

Wollen wir bei unserem Fön-Beispiel oben feststellen, ob die Heizung aktiviert ist, können wir dies demnach wie folgt feststellen:

```
unsigned char heaterActivated = hairDryerState & 0b1000000;
```

Einen Wert auslesen

Um einen Wert aus einer Variable auszulesen, der mehrere Bits umfasst, verwenden wir zunächst eine UND-Maske, die alle relevanten Bits isoliert. Anschließend schieben wir die Bits nach rechts, um den konkreten Wert zu erhalten.

Wollen wir beispielsweise die Lüfterstufe des Föns auslesen, geht das über die folgende Zeile:

```
unsigned char fanLevel = (ch & 0b111100) >> 2;
```

Die &-Verknüpftung setzt alle Bits auf 0, die nicht zur Lüfterstufe gehören. Der Bit-Shift nach rechts um 2 Stelle sorgt dafür, dass wir die konkrete Lüfterstufe von 0 bis 15 erhalten.

Ein Bit setzen

Über den binären |-Operator kann man gezielt ein Bit setzen, indem man einen Operanden benutzt, der nur das zu setzende Bit enthält. Durch die Oder-Operator bleiben alle bereits gesetzten Bits erhalten.

Der folgende Code demonstriert das:

```
unsigned char chWithBitSet = ch | 0b100;
```

`chWithBitSet` enthält nun den Wert von `ch`, wobei zusätzlich das dritte Bit von rechts gesetzt ist (sofern es nicht vorher schon der Fall war).

Um den Zustand zu setzen, der die Heizung des Beispiel-Föns als aktiv markiert, kann man also folgende Zeile nutzen.

```
unsigned char heaterActivated = hairDryerState | 0b1000000;
```

Ein Bit aufheben

Um ein Bit wieder auf 0 zusetzen, muss man den UND-Operator auf eine inverse Bitmaske anwenden. Möchte man beispielsweise das dritte Bit von rechts aus einer Zahl entfernen, würde das in C wie folgt aussehen.

```
unsigned char chWithBitSet = ch & ~0b100;
```

Der Code entspricht ausgehend von einem 8-Bit-unsigned-char dieser Codezeile:

```
unsigned char chWithBitSet = ch & 0b11111011;
```

Es bleiben also alle Bits erhalten, außer dem dritten von links, dass hier auf 0 gesetzt wird. Die Verwendung der Inversion über den Tilde-Operator (~) hat den Vorteil, dass man sich nicht über die binäre Länge der Zahl Gedanken machen muss.

Die Heizung des Föns kann man also wie folgt ausschalten:

```
unsigned char heaterActivated = hairDryerState & ~0b1000000;
```

Ein Bit umdrehen

Um ein Bit umzudrehen (von 0 auf 1 oder umgekehrt), verwendet man den XOR-Operator mit einer Bitmaske für dieses Bit. Die Operation sorgt dafür, dass alle Bits gleich bleiben, bei denen die Bitmaske 0 ist. Dort wo die Bitmaske 1 ist, wird das Bit umgedreht.

Das folgende Beispiel zeigt, wie man z.B. den Zustand der Heizung (an oder aus) aus dem obigen Beispiel wechselt:

```
hairDryerState = hairDryerState ^ 0b1000000;
```

Der zweite Operand des XOR-Operators bewirkt für die **0**-Bits, dass das jeweilige Bit erhalten bleibt: Trifft es auf eine 1, bleibt der Wert 1 (**0** ^ 1). Trifft er auf eine 0, sind die Werte gleich, und die 0 bleibt erhalten (**0** ^ 0). Das Wechseln über das **1**-Bit (oben hervorgehoben) funktioniert wie folgt: Trifft es auf eine bestehende 1, wird sie auf 0 gesetzt (**1** ^ 1). Trifft sie auf eine 0, sind die Werte ungleich, und es wird eine 1 gesetzt (**1** ^ 0).

Einen Wert setzen

Um einen Wert zu setzen, der sich über mehrere Bits erstreckt, setzt man diese Bits zunächst auf 0 und setzt dann den neuen Wert mit einer ODER-Verknüpfung. Der neue Wert muss zuvor noch an die richtige Stelle geschoben werden.

Nehmen wir an, wir möchten die Lüfterstufe des Föns einstellen und diese ist in einer Variable `fanLevel` hinterlegt. Dann wird dieser Zustand des Föns wie folgt aktualisiert.

```
hairDryerState = (hairDryerState & ~0b111100) | (fanLevel << 2);
```

Die UND-Verknüpfung mit der invertierten Maske setzt die Bits 2 bis 5 auf 0. Anschließend wird der 4-Bit `fanLevel` an der richtigen Stelle durch die ODER-Verknüpfung eingesetzt.

Das folgende Bild veranschaulicht diesen Vorgang. Der zu ersetzende Teil für die Lüfterstufe ist gelb markiert, die neue Lüfterstufe grün:

hairDryerState	0	1	1	1	0	0	1	1
0b111100	0	0	1	1	1	1	0	0
~0b111100	1	1	0	0	0	0	1	1
hairDryerState & ~0b111100	0	1	0	0	0	0	0	1
fanLevel	0	0	0	0	1	0	1	0
fanLevel << 2	0	0	1	0	1	0	0	0
(hairDryerState & ~0b111100) | (fanLevel << 2)	0	1	1	0	1	0	1	1
Stelle	7	6	5	4	3	2	1	0

Alternative: Bitfelder

Die Manipulation einzelner Bits kann recht aufwendig sein und die Fehlersuche erweist sich oft als schwierig. Als Alternative für den Einsatz von bitweisen Operatoren können in C sogenannte Bitfelder verwendet werden. Das sind Strukturen, für deren Elemente genaue Bitgrößen angegeben werden können.

Bitfelder sind genauso definiert wie strukturierte Datentypen (struct), legen aber für jedes Element fest, wieviel Bits es belegt. Bitfelder erlauben damit dadurch, das Bit-Layout innerhalb eines Bytes zu definieren. Das bereits bekannte Beispiel

Bedeutung	Heizung aktiviert?	Lüfter aktiviert?	Lüfterstufe (0-15)				Heizstufe (0-3)	
Stelle	7	6	5	4	3	2	1	0

lässt sich wie folgt als Bitfeld definieren:

```
struct HairDryerState {
    unsigned char heaterActivated:1;  // Heizung aktiviert (ja/nein)
    unsigned char fanActivated:1;     // Lüfter aktiviert (ja/nein)
    unsigned char fanLevel:4;         // Lüfterstufe (0-15)
    unsigned char heaterLevel:2;      // Heizstufe (0-3)
};
```

Das Lesen und Schreiben von Werten erfolgt dann über die üblichen Mechanismen, um Variablen strukturierter Datentypen zu ändern. Der folgende Code zeigt, wie sich die Lüfterstufe eines Typs ändern lässt:

```
struct HairDryerState state;
state.fanLevel = newFanLevel;
```

Diese Zeile hat auf Bitebene die gleiche Auswirkung wie das oben bereits genannte Beispiel:

```
hairDryerState = (hairDryerState & ~0b111100) | (newFanLevel << 2);
```

Die Verwendung eines Bitfelds ist also deutlich komfortabler. Hier ist aber noch zu beachten, dass ein Bitfeld immer mindestens so groß ist wie der größte darin enthaltene Datentyp. Fügt man also ein long-Attribut in das obigen Bitfeld ein, ist das Bitfeld sizeof(long) groß.

Möchte man nun den Inhalt eines Bitfelds als 1-Byte-char auslesen, kann man dessen Adresse in einen unsigned-char-Zeiger casten und anschließend dereferenzieren:

```
unsigned char stateByte = *((unsigned char*) &state);
```

Das geht natürlich auch in die andere Richtung. Das Byte lässt sich also wie folgt wieder in ein HairDryerState umwandeln.

```
state = *((struct HairDryerState*) &stateByte);
```