Bitweise – oder binär

Aktualisiert am 14. 12. 2023:

Abgewandelt – nach der Quelle:
https://www.proggen.org/doku.php?id=c:article:binaryops

Die Begriffe „binär“, „unär“ und „bitweise“ können schon verwirren. Darum diese Abhandlung.

Binär - unär

Nach meiner Definition sind „binäre“ (**zweistellige; „zwei-operandige“, mit 2 Operanden in Beziehung stehende**) Operatoren bzw. Funktionen unter anderen diese hier: + - * / %. Hier kurz eine Erklärung: + ist die Addition, - ist die Subtraktion, * ist die Multiplikation, / ist die Division – und % ist der Restoperator oder „modulo“.

Funktionen sind zum Beispiel: add(x, y), subtract(a, b).

Binär heißt hier: zweistellig. Also ein Operator mit genau zwei Operanden. Ein Operator, der mit zwei Operanden in Beziehung steht. Die Addition dient hier als Beispiel: In der Mathematik sieht man Aufgaben wie: 2 + 8 = 10

Der Term, auf den es hier ankommt, ist der: 2 + 8

+ ist hier der Operator, 2 und 8 sind die zwei Operanden (genau zwei!).

Was bedeutet da zweistellig? Dem plus-Zeichen sind die zwei Zahlen 2 und 8 zugeordnet, oder: das Plus-Zeichen steht mit zwei Zahlen in Beziehung. Das sind zwei Zahlen, darum zweistellig. (Man sagt auch „Infix“-Notation dazu; aber das nur nebenbei.)

Mehr solche binäre (zweistellige) Operatoren gibt es in der Programmiersprache C.

In diesem Aufsatz gehe ich von den „Token“ der Programmiersprache C aus, damit wir über die gleichen Operationen sprechen.

Im Folgenden sind hier beispielhaft 18 Operatoren aufgeführt:

Aus dem arithmetischen Erahrungsbereich sind diese, die ich schon genannt habe +-*/%.

Die folgenden Operatoren haben mit Logik, Wahrheitswerten oder boolescher Algebra zu tun

< (kleiner als), > (größer als), == (logischer Vergleich), <= (kleiner oder gleich), >= (größer oder gleich).

Binär (zweistellig) ist sogar das einfache Gleichheitszeichen =. In C dient dieses als Zuweisungszeichen. (11)

Weitere binäre (zweistellige) Operatoren sind nach meiner Definition das logische Und (12), das logische Oder (13), das bitweise Und (14), das bitweise Oder (15) und das bitweise Xoder (das exklusive Oder; 16).

Noch mehr binäre Operatoren gibt es, nämlich: shift left und shift right. Zu deutsch: Linksverschiebung bzw. Rechtsverschiebung (In C: int val = 1 << 1;). Ein Codebeispiel in C mit shift left:

```
#include <stdio.h>


int main(void)
{
	int val = 1 << 1;
	printf("Val: %d\n", val);//2
	return 0;
}
```

Unär (einstellig, mit nur einem Operanden in Beziehung stehend) ist anders als binär: Unär sind u. a. das logische Nicht und das bitweise Nicht, das Komplement.

Eine Anweisung: int b = !a; Das ist unär, weil dem Operator (hier dem !) nur ein Operand zugeordnet ist. In C gibt es den Ausdruck: !1.

Bitweise

Bitweise ist nun etwas ganz anderes:

Die bitweisen-Operationen werden häufig als mystische Hacker-Operatoren angesehen. Das stimmt in gewisser Weise auch, aber dafür sind wir ja da, um hier Licht ins Dunkel zu bringen: Wissen zerstört Mystik. Wenn man damit umgehen kann, geht zwar die Mystik flöten, dafür finden sich eine ganze Reihe nützlicher Anwendungen. Als Beispiele sei hier die Kodierung von UTF-8 Zeichen genannt oder auch das Verpacken von Binärdaten in ASCII-Codes per Base64.

Die Grundlagen

Eigentlich gibt es da in der Programmiersprache C mehr als drei Stück, zum Beispiel „Und“, „Oder“ und „Exklusives Oder“. Verglichen werden einfach ausgedrückt immer Wahrheitswerte, also True und False; und sie geben einen entsprechenden Wert (also true oder false) zurück. Im Tutorial finden sich hierfür bisher die Operation „logisches Und“ (in C geschrieben „&&“) und „logisches Oder“ („||“) mit der Bedingungen entschieden werden. Ein „logisches exklusives Oder“ gibt es in C leider nicht.

Diese logischen Operatoren treffen immer genau eine Entscheidung: wahr oder falsch. Die bitweisen Operatoren machen das gleiche, nur, dass sie zum Beispiel 32 bzw. 64 Entscheidungen gleichzeitig treffen,

denn man kann sie auf Integerzahlen anwenden. Um es einfacher und anschaulicher zu gestalten und damit wir hier nicht über den Papierrand schreiben müssen, treffen wir hier mal 8 Entscheidungen gleichzeitig: Schauen wir uns vier einfache 8 Bit-Integer an:

-7- -6- -5- -4- -3- -2- -1- -0-

0 0 0 0 0 0 0 0 **entspricht dem Wert 0 logisch false**

0 0 0 0 0 0 0 **1** **entspricht dem Wert 1 logisch true**

0 0 0 0 0 0 **1** 0 **entspricht dem Wert 2 logisch true**

0 0 0 0 0 **1** 0 **1** **entspricht dem Wert 5 logisch true**

Der Rückgabewert für (3 && 5) ist logisch true, denn 3 und 5 Werte sind logisch true. Aber beide Werte haben auch ein Bitmuster und mit diesem Bitmuster arbeiten die bitweisen Operatoren. In C werden sie „&“ (Und), „|“ (Oder) und „^“ (exklusives Oder) geschrieben. Das Besondere dieser bitweisen (auf alle einzelnen Bits bezogenen) Operatoren ist, dass sie den Wert nicht als ganzes betrachten, sondern eben bitweise arbeiten. Das erste Bit des linken Operanden wird immer mit dem ersten Bit des zweiten Operanden verarbeitet, usw.

Schauen wir uns nun die ersten drei Operatoren an, bevor wir uns Gedanken darüber machen, was man damit anfangen kann.

Die bitweise Und-Operation

Schauen wir uns das ganze mal mit einer „Und“-Operation an. Beim logischen Und („&&“) müssen beide Seiten „true“ ergeben, damit „true“ als Ergebnis herauskommt. Hier ergeben sich die beiden Seiten durch die jeweilige Bitrepräsentation:

-7- -6- -5- -4- -3- -2- -1- -0-

0 0 0 0 0 0 **1** **1** **entspricht dem Wert 3 logisch true**

&& && && && && && && && **bitweise entscheiden**

2									
&	0	0	0	0	0	**1**	0	**1**	**entspricht dem Wert 5 logisch true**
5									
ergibt	0	0	0	0	0	0	0	1	**entspricht dem Wert 1**

Mit dieser Anweisung schaffen wir also 8 „&&“ Anweisungen gleichzeitig auszuführen. Schauen wir uns das ganze nochmal mit 1 & 5 an:

	-7-	**-6-**	**-5-**	**-4-**	**-3-**	**-2-**	**-1-**	**-0-**	
	0	0	0	0	0	0	0	**1**	**entspricht dem Wert 1 logisch true**
1									
&	**&&**	**&&**	**&&**	**&&**	**&&**	**&&**	**&&**	**&&**	**bitweise entscheiden**
5									
	0	0	0	0	0	**1**	0	**1**	**entspricht dem Wert 5 logisch true**
ergibt	0	0	0	0	0	0	0	**1**	**entspricht dem Wert 1 logisch true**

Wir haben hier für den Wert 1 und für den Wert 5 das 0. Bit gesetzt. Das 0. Bit ist für jede ungerade Zahl gesetzt, so dass (wert & 1) immer 1 ist, wenn die Zahl ungerade ist. Dies entspricht der Operation (wert % 1), nur dass (wert & 1) eine sehr einfache Operation ist, die auch von alten Rechnern sehr schnell bearbeitet werden kann. (wert % 1) ist eine sehr aufwendige Operation, die besonders alte Computer deutlich ausbremsen kann. Die Zahlen werden Bit für Bit verglichen und wenn beide Bits gesetzt sind, so hat auch das Ergebnis an der entsprechenden Position sein Bit gesetzt.

Wenn wir uns das so anschauen, so eignet sich die Und-Operation dafür, um Gemeinsamkeiten in den Bitmustern herauszuarbeiten.

Die bitweise Oder-Operation

Nun schauen wir uns das Gleiche mit Oder an.

```
      -7- -6- -5- -4- -3- -2- -1- -0-

      0   0   0   0   0   0   1   0   entspricht dem Wert 2 logisch true
2
|     ||  ||  ||  ||  ||  ||  ||  ||        bitweise entscheiden
5
      0   0   0   0   0   1   0   1   entspricht dem Wert 5 logisch true

ergibt 0   0   0   0   0   1   1   1   entspricht dem Wert 7 logisch true
```

Das sieht fast wie eine Addition aus - es ist aber keine! Das sehen wir deutlicher am folgenden Beispiel: (1 | 5)

```
      -7- -6- -5- -4- -3- -2- -1- -0-

      0   0   0   0   0   0   0   1   entspricht dem Wert 1 logisch true
1
|     ||  ||  ||  ||  ||  ||  ||  ||        bitweise entscheiden
5
      0   0   0   0   0   1   0   1   entspricht dem Wert 5 logisch true

ergibt 0   0   0   0   0   1   0   1   entspricht dem Wert 5 logisch true
```

Wie beim bitweisen Und-Operator wird Bit für Bit miteinander verarbeitet und wenn mindestens ein Bit gesetzt ist, bleibt beim bitweisen Oder das Bit auch im Ergebnis gesetzt.

Mit der „Oder“-Operation kann man ein Bitmuster über ein anderes Bitmuster legen.

Die bitweise Exklusiv-Oder-Operation

Das exklusive Oder wird häufig auch mit X-Oder, bzw. eher noch englisch XOR abgekürzt. Damit hier das Ergebnisbit auf true gesetzt wird, muss genau ein Bit true sein und das jeweils andere auf false stehen. Stehen beide Bits auf true oder beide Bits auf false, so ist das Ergebnisbit false. Einen „^^“-Operator gibt es nicht, ich trage ihn nur in die Tabelle ein, um zu verdeutlichen, dass es eine logische Operation zwischen den jeweiligen beiden Bits darstellt.

```
      -7- -6- -5- -4- -3- -2- -1- -0-

      0   0   0   0   0   0   1   0   entspricht dem Wert 2 logisch true
2
^     ^^  ^^  ^^  ^^  ^^  ^^  ^^  ^^        bitweise entscheiden
5
      0   0   0   0   0   1   0   1   entspricht dem Wert 5 logisch true

ergibt 0  0   0   0   0   1   1   1   entspricht dem Wert 7 logisch true
```

Das sieht soweit aus, wie bei der Oder-Operation, doch im zweiten Beispiel sehen wir den Unterschied:

```
      -7- -6- -5- -4- -3- -2- -1- -0-

      0   0   0   0   0   0   0   1   entspricht dem Wert 1 logisch true
1
^     ^^  ^^  ^^  ^^  ^^  ^^  ^^  ^^        bitweise entscheiden
5
      0   0   0   0   0   1   0   1   entspricht dem Wert 5 logisch true

ergibt 0  0   0   0   0   1   0   0   entspricht dem Wert 4 logisch true
```

Das 0. Bit ist hier in beiden Fällen 0.

Das exklusive Oder eignet sich um einzelne Bits nach einem Muster umzudrehen. Es eignet sich somit für eine sehr einfache, aber effektive Verschlüsslung von Daten.

Die bitweise Negation

Genau, wie man eine logische Aussage (wahr oder falsch) ins Gegenteil verkehren kann, kann man das auch mit einer binären Zahl, also bitweise: Man kippt einfach alle Bits um. Das entspricht einem exklusiven Oder mit einer Bitmaske in der alle Bits gesetzt sind. Da diese Bitmaske immer voller Einsen ist, ist sie als Parameter überflüssig. Die bitweise Negation ist also wie die logische Negation ein unärer Operator.

Zur Erinnerung: Die logische Negation nehmen wir z.B. in If-Abfragen:

```
if( !success )//Falls kein Erfolg
  printf( "Hier ist was schief gelaufen.\n" );
```

Damit drehen wir den Wahrheitswert von success um. Aus false wird true und umgekehrt. Falls wir nicht erfolgreich waren, gibt es eine Meldung. Macht man das logische Negieren mit einem Integer, so wird aus einer beliebigen Zahl 0 und aus 0 eine beliebige Zahl, die nicht 0 ist (in der Regel, ja in C, kommt 1 zurück).

Bei der **bitweisen** Negation wird aber jedes Bit gekippt. Wozu das unter anderem gut ist, sehen wir später bei der Anwendung mit den Flags.

	-7-	-6-	-5-	-4-	-3-	-2-	-1-	-0-	
~ 5	0	0	0	0	0	**1**	0	**1**	**entspricht dem Wert 5 logisch true**
ergibt	**1**	**1**	**1**	**1**	**1**	0	**1**	0	**entspricht dem Wert 250 logisch true**

Anwendungsbeispiele

ASCII-Zeichensatz

Schauen wir uns die ASCII-Tabelle an:

Code	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	**NUL**	**SOH**	**STX**	**ETX**	**EOT**	**ENQ**	**ACK**	**BEL**	**BS**	**HT**	**LF**	**VT**	**FF**	**CR**	**SO**	**SI**
1_	**DLE**	**DC1**	**DC2**	**DC3**	**DC4**	**NAK**	**SYN**	**ETB**	**CAN**	**EM**	**SUB**	**ESC**	**FS**	**GS**	**RS**	**US**
2_	SP	!	„	#	$	%	&	'	(	)	*	+	,	-	.	/
3_	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4_	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5_	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6_	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7_	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	**DEL**

Wir sehen zunächst die nicht druckbaren Zeichen von 0 (0x00) bis 31 (0x1F). Das Zeichen 48 (0x30) ist die '0'. Warum legt man die Ziffer Null auf das ASCII-Zeichen 48? Ganz einfach: Die Zahl 48 ist hexadezimal 0x30, also binär 0011 0000. Wenn man weiß, dass nun

nur Ziffern folgen, kann man den Wert der Ziffer durch eine Bitmaske auslesen. Uns interessieren nur die hinteren vier Bits, also ist die Bitmaske 00001111 (entspricht 15 oder 0x0F).

```
Wert = ASCIIZiffer & 0x0F;
```

Hier holen wir uns also nur die hinteren 4 Bits ab, die die Werte von 0-15 einnehmen können. Solange wir ASCII-Ziffern lesen, sind das die Werte von 0-9.

Schauen wir uns nun die Buchstaben an. Sie liegen direkt untereinander. Werfen wir einen Blick auf das große und das kleine 'A'. Wir haben die Werte 65 für A und 97 für a. Die beiden Zahlen ergeben auf den ersten Blick aber also keinen besonderen Sinn. Schauen wir uns nun aber das Bitmuster an:

	-7-	**-6-**	**-5-**	**-4-**	**-3-**	**-2-**	**-1-**	**-0-**	**dezimal**	**hex**
'A'	0	1	**0**	0	0	0	0	1	**65**	**0x41**
'a'	0	1	**1**	0	0	0	0	1	**97**	**0x61**

Wir erkennen, dass sich die beiden Buchstaben nur in einem Bit unterscheiden, sich also eigentlich sogar sehr ähnlich sind. Nehmen wir an, wir wissen, dass wir einen Buchstaben betrachten und wollen nun wissen, ob dieser Buchstabe groß oder klein ist. Das 5. Bit entspricht 2^5, also 32 oder hexadezimal 0x20 (2*16+0 == 32)

```
char c='c';

// Ich weiß, dass sich in der Variablen c ein Buchstabe befindet

if( c >= 'a' && c <= 'z' )
  printf( "Es handelt sich um einen kleinen Buchstaben\n" );
else
  printf( "Es handelt sich um einen großen Buchstaben\n" );
```

Diese Frage besteht aus der Frage, ob c größer oder gleich 'a' ist und ob c kleiner oder gleich 'z' ist - es handelt sich also um zwei Fragen. Wir wissen aber, dass in c ein Buchstabe enthalten ist, wir können nun also einfach das entsprechende Bit abfragen:

```
char c='c';
```

```
// Ich weiß, dass sich in der Variablen c ein Buchstabe befindet

if( c & 0x20 )
  printf( "Es handelt sich um einen kleinen Buchstaben\n" );
else
  printf( "Es handelt sich um einen großen Buchstaben\n" );
```

Nun haben wir nur noch eine Frage, die Entscheidung fällt also doppelt so schnell wie zuvor.

Flags

Als Flags bezeichnet man Ja/Nein-Entscheidungen. Eine „Flagge“ ist also entweder gesetzt (oben) oder gelöscht (weg vom Fahnenmast). Flags findet man häufig zum Beispiel bei Betriebssystemen, wo man z.B. bei einem Fenster viele Ja/Nein-Fragen beantworten muss. Hat das Fenster einen Rahmen oder darf man es verschieben oder darf man es vergrößern oder darf man es schließen oder darf man auf Vollbild schalten. Diesen etwas holprigen Satzbau mit den vielen „oder“ wähle ich bewusst, denn auch hier hilft das „oder“.

Wir könnten nun eine CreateWindow-Funktion erstellen, die pro Frage einen Parameter hat. Das bedeutet, dass der Programmierer sehr viele Fragen beantworten muss. Außerdem wäre die CreateWindow-Funktion sofort veraltet, wenn mit einer neuen Betriebssystemversion auch nur eine zusätzliche Option für Fenster entstehen würde, zum Beispiel die Möglichkeit Fenster in die Taskleiste auszublenden.

Ein Integer ist 32 Bit breit, hier können also 32 Ja/Nein-Informationen untergebracht werden. Der Übersicht halber verwende ich jetzt nur sechs. Die Flags definieren wir hier mal mit Hilfe des Präprozessors:

```
#define WINDOWFLAG_CLOSEBUTTON    0x01
#define WINDOWFLAG_BORDERLESS     0x02
#define WINDOWFLAG_RESIZEABLE     0x04
#define WINDOWFLAG_FULLSCREEN     0x08
#define WINDOWFLAG_ALWAYSONTOP    0x10
#define WINDOWFLAG_STATICPOSITION 0x20
```

Die Flags entsprechen immer 2 hoch x. Also 2 hoch 0 für das erste (=1, entspricht 0x01), 2 hoch 1 für das zweite (=2 == 0x02) und so weiter.

Schauen wir uns nun eine fiktive CreateWindow-Funktion an:

```
void CreateWindow( int flags )
{
  if( flags & WINDOWFLAG_CLOSEBUTTON )
    printf( "Wir brauchen einen Close-Button\n" );

  if( flags & WINDOWFLAG_BORDERLESS )
    printf( "Normale Fenster haben einen Rahmen - dieses nicht\n" );

  if( flags & WINDOWFLAG_RESIZEABLE )
    printf( "Das Fenster darf vergrößert werden" );

  // und so weiter
}
```

Wir benutzen hier den Wert des Flags als Maske und maskieren damit das eine Bit aus, das uns an dieser Stelle interessiert. Wenn es gesetzt ist, kommt eine Zahl heraus (die dem Wert des Bits entspricht). Ist das Bit nicht gesetzt, so kommt 0 heraus, was dem Wahrheitswert false entspricht.

Mit den Flags können wir nun ein gewünschtes Verhalten des Fenster zusammensetzen:

```
CreateWindow( WINDOWFLAG_CLOSEBUTTON | WINDOWFLAG_RESIZEABLE );
```

Hiermit haben wir ein Fenster definiert, was einen Button zum Schließen besitzt und in der Größe verändert werden kann. Da wir WINDOWFLAG_BORDERLESS nicht verwenden, wird es entsprechend einen Border besitzen. Wir haben uns auch nicht explizit für eine statische Position ausgesprochen, also darf der Anwender es verschieben usw. Wir können also mit den Flags genau festlegen, wie das Fenster im Vergleich zum absoluten Standard-Fenster verändert werden soll.

Flags löschen

Wir wissen nun, dass wir Flags mit der Oder-Operation setzen können und sie mit der Und-Operation abfragen können. Da muss man sie doch mit der Exklusiv-Oder Operation löschen können! Vorsicht: XOR löscht Bits nicht, sondern dreht sie um!

```
int flags = WINDOWFLAG_FULLSCREEN | WINDOWFLAG_CLOSEBUTTON;
```

```
flags = flags ^ WINDOWFLAG_FULLSCREEN;     //          FullScreen-Flag
entfernen
flags = flags ^ WINDOWFLAG_RESIZEABLE;     // FALSCH: Resizeable-Flag
entfernen
flags = flags ^ WINDOWFLAG_CLOSEBUTTON;    //          Close-Button-Flag
entfernen
```

Da XOR die Bits kippt, wird das Resizeable-Flag nicht gelöscht, denn es war ursprünglich ja gelöscht. Also wird es gekippt und ist damit nun gesetzt.

Wir müssen also erst wissen, ob wir zum Löschen überhaupt das jeweilige Bit kippen dürfen:

```
int flags = WINDOWFLAG_FULLSCREEN | WINDOWFLAG_CLOSEBUTTON;

if( flags & WINDOWFLAG_FULLSCREEN )
  flags = flags ^ WINDOWFLAG_FULLSCREEN;     //          FullScreen-Flag
entfernen
```

Das ist allerdings aufwendig (schließlich muss man erst die if-Abfrage verarbeiten) und man muss das für jedes Bit einzeln machen. Zum Glück können wir das Gegenteil der Oder-Anweisung bestimmen: Es ist das Und mit negierten Parametern. Unverständlich? Also mal ein Beispiel aus der Logik: Ein T-Shirt ist cool, wenn es schwarz ist ODER bedruckt ist. Weiße T-Shirts definieren wir damit also als uncool.

```
CoolesTShirt = Schwarz || Bedruckt;
```

Kein cooles T-Shirt ist also alles, was NICHT schwarz UND NICHT bedruckt ist. Wir können also auch schreiben, dass ein T-Shirt cool ist, wenn es nicht nicht schwarz ist und nicht bedruckt. Und hier gehen uns in der deutschen Sprache langsam die Möglichkeiten aus, nicht aber in C, denn da können wir nämlich Klammern setzen:

```
CoolesTShirt = !(!Schwarz && !Bedruckt);
```

Ein T-Shirt ist cool, wenn es NICHT (UNschwarz UND UNbedruckt) ist. Die Klammern sind jetzt absichtlich gesetzt.

Logik ist komisch, ich weiß, aber das gibt sich mit der Zeit.

SCHWARZ und BEDRUCKT könnten jetzt auch Flags sein, die ein Kleidungsstück beschreiben

```
(CreateClothes( FLAG_SCHWARZ | FLAG_BEDRUCKT )).
```

Das Gegenteil von Oder ist also „NICHT UND“. Und das können wir nun auch verwenden, um solche Flags zu löschen. Da wir nun mit Flags arbeiten, nehmen wir statt der logischen Negation die Operator für **bitweise** Negation: **~**.

```
flags = flags & ~WINDOWFLAG_CLOSEBUTTON;
```

Was passiert: wir generieren mit !WINDOWFLAG_CLOSEBUTTON eine Bitmaske, in der alle Bits 1 sind, außer das Bit, was für WINDOWFLAG_CLOSEBUTTON steht. Diese Bitmaske lässt alle Bits von flags genauso durch, wie sie sind - nur das Bit für WINDOWFLAG_CLOSEBUTTON wird auf Null gesetzt - egal, wie es vorher war.

Und auch hier können wir die Bitmaske wieder zusammensetzen:

WINDOWFLAG_CLOSEBUTTON | WINDOWFLAG_RESIZEABLE | WINDOWFLAG_FULLSCREEN.

```
flags = flags & ~( WINDOWFLAG_CLOSEBUTTON | WINDOWFLAG_RESIZEABLE |
WINDOWFLAG_FULLSCREEN );
```

oder noch kürzer:

```
flags &= ~( WINDOWFLAG_CLOSEBUTTON |
WINDOWFLAG_RESIZEABLE | WINDOWFLAG_FULLSCREEN );
```

Du bist am Zug!

Vermutlich raucht dem einen oder anderen jetzt der Kopf und man denkt sich, dass man nichts verstanden hat. Sollte das gerade der Fall sein - das ist normal. Flags werden daher heute seltener unterrichtet, weil es als kompliziert gilt. Weil man sich Zeit dafür nehmen muss, sich damit vertraut zu machen. Nimm Dir die Zeit Testprogramme zu schreiben. Schreibe die fiktive CreateWindow-Funktion von oben fertig. Dann lege Dir eine Flags-Variable an und verändere sie: Füge

Bits hinzu und lösche Bits. Mit der CreateWindow-Funktion erhältst Du eine Ausgabe, welche Bits gesetzt sind.

Schreibe eine Funktion, die Dir die sechs Flags als Bitmuster ausgibt, also z.B. „011001“ für

WINDOWFLAG_CLOSEBUTTON, WINDOWFLAG_FULLSCREEN und WINDOWFLAG_ALWAYSONTOP.

Bitweise oder binaer sehrgut.docx