Acta et Commentationes, Sciences of Education, nr. 4 (18), 2019 ISSN 1857-0623
p. 62-72 E-ISSN 2587-3636

CZU: 004.02 DOI: 10.36120/2587-3636.v18i4.62-72

BITWISE OPERATORS IN C LANGUAGE
THE EXPLANATION, IN DETAIL, OF THEIR USE. LITTLE TRICKS

Marius NICOLI, ”Frații Buzești” National College, Craiova, România
*Coordinator of the National Junior Lottery; Team Leader of the national team at several international competitions; coordinator of the Scientific Commission at the Balkan Youth Olympiad (2018); teacher didactic grade 1.

Summary. The programming languages provide a number of operators, primarily for the usual mathematical operations. Usually they have associated instructions that the microprocessor can execute directly. The bitwise operators also correspond to instructions that the microprocessor has in its set, even if they do not necessarily correspond to widely known operations. Thus, using them to perform certain tasks makes the programs very fast, especially if they are used in their critical sequences. In this material we will present the syntax and the effect of these operators in the C / C ++ language and we will present relevant situations where certain problems can be solved elegantly and by code that is executed quickly, with the help of bitwise operators.

Keywords. Programming languages, bitwise operators, the solution of the problem, instructions.

OPERATORI PE BIȚI ÎN LIMBAJUL C.
EXPLICAREA ÎN DETALIU A MODULUI DE UTILIZARE. MICI TRUCURI

Rezumat. Limbajele de programare oferă o serie de operatori, în principal pentru operațiile matematice obișnuite. De obicei, au instrucțiuni asociate pe care microprocesorul le poate executa direct. Operatorii pe biți corespund, de asemenea, instrucțiunilor pe care microprocesorul le are în set, chiar dacă nu corespund neapărat cu operații cunoscute. Astfel, utilizarea lor pentru a îndeplini anumite sarcini face ca programele să fie foarte rapide. În acest articol este prezentată sintaxa și efectul acestor operatori în limbajul C / C ++ și elucidate situații relevante în care anumite probleme pot fi rezolvate în mod elegant și printr-un cod care este executat rapid, cu ajutorul operatorilor pe biți.

Cuvinte cheie. limbaje de programare, operatori pe biți, soluția problemei, instrucțiuni.

Introduction

They are operators that apply to data of whole types. In order to understand the effect of their application, we must master the internal representation of integers in the reserved memory area. To be simpler to write the examples, we will consider integer variables on 2 bytes (16 bits), which we will present being valid for the other types of whole data. The first rule is that positive values are stored in their reserved bits according to the writing of the value in base 2. Exemple:

```
short a;
a = 43;
```

The variable a will occupy 16 bits and following the assignment, their configuration will be the following:

Bit position	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bit value	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	1

According to the writing rule in base 10 we have $2^5 + 2^3 + 2^1 + 2^0 = 43$. A short variable type we know it may range between -2^{15} and $2^{15} - 1$ (there are a total of 2^{16} values, that is, for each configuration of 0 and 1 of length 16, one value in this range). Therefore, the maximum positive value is $2^{15} - 1$. But this number is equal to the value of the expression: $2^{14} + 2^{13} + \ldots + 2^1 + 2^0$. This is a known thing in mathematics, but we will see that there are more interpretations in informatics that justify this result. We deduce that for the maximum value, all bits, except the first one, have the value 1. So, the internal representation of 32767 is:

0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

We deduce automatically: the other positive values have as internal representations configurations that can have 0 values instead of the existing 1 values. The conclusion is as follows: all configurations that have the first bit 0 correspond to positive values. If our variable were of the unsigned short type, the value 1 for the first bit (the front bit, bit 15 or, more often said, the most significant bit) would have represented positive values between 2^{15} and $2^{16} - 1$. That's because at the positive values of the short 2^{15} is added (because it is the 1 the most significant bit).

For the `short` type (and the other integer types that are not written `unsigned`), the configurations that have the most significant bit equal to `1` are internal representations of negative numbers.

We will present below the rule after which the module is deduced when the configuration stores a negative value. If we thought hastily, we could say that the presence of 1 in front means a negative number and the conversion into the base 10 of the configuration with the other bits is the absolute value. But it is not quite so. Here is a simple justification, for example:

The number 1 stored in a short variable has the representation:

```
0000000000000001
```

Number -1, as in the above reasoning would be:

```
1000000000000001
```

The addition of the two values should be 0. But it is obvious that this is not the case.

```
0000000000000001 +
1000000000000001 =
1000000000000010
```

Here is the correct rule of finding the absolute value at the negative numbers (called the rule of the complement against 2): all the bits are complemented (the values 1 are replaced by 0 and the 0 are replaced by 1), then they are added to the value 1 (based on 2).

Here is an example:

Internal representation	1111111100010100
After complementary	0000000011101011
After adding 1	0000000011101100 (bits of 1 at the end become 0 and the first 0 becomes 1, the remaining values of the bits are kept).

Now, writing in base 10 the obtained number (0000000011101100) we have $2^7 + 2^6 + 2^5 + 2^3 + 2^2 = 236$.

So the initial sequence, that is 1111111100010100, represents the internal representation of the number -236.

Here's a sample:

-236 +	1111111100010100 +
236	0000000011101100
0	10000000000000000

It is observed that the last 16 bits reach all 0, and the front bit, 1 (given by transport), is outside the allocated memory area, so the result is 0.

Remarks:

- the value -1 will be represented by all bits 1 (if we add the value 1 the transport always propagates and the value 0 is obtained);
- the value with all bits 1 is at the same time the maximum value of the corresponding *unsigned* type;
- a configuration consisting only of bits 1 represents a power shape number of 2 - 1;
- the odd numbers are those that have the least significant bit equal to 1 (the only odd power of 2 that appears in sum when transforming from base 2 to base 10).

We can say that the following *rule for transforming a number from base* 10 *to base* 2 is valid (apart from the general one where we repeatedly divide to 2 and take the remaining values in reverse order): For the given number, we look for the highest power of 2 less than or equal to the given number, then subtract it from the number and resume for the remaining value. We repeat this until the value reached is 0. This approach is correct because we will not determine twice the same power of 2 (the sum of two powers of 2 is also a power of 2 and would mean that at a step we would not have found correctly the power of 2 less than or equal to the current value). Also, the fact that the powers of 2 that appear are distinct is consistent with the fact that we have only the digit 0 and the digit 1 in the base 2.

Exemple: We would like to write in base 2 the number 41. We look for the biggest power of 2 less than or equal to 41 and we find 32. We continue with 9 (41 - 32). We find 8 and we continue with 1 (9-8). Now we have 1 and so we reach 1-1 = 0, meaning that we can stop. So $41 = 32 + 8 + 1 = 2^5 + 2^3 + 2^0$. So the writing in base 2 of the number 41 is: 101001.

Bitwise operators are divided into two categories: *logical bitwise operators* and *shift operators*.

Logical bitwise operators

We emphasize what we said before, they apply to whole data.

They are:

~	*not* logical on bits, it is a unary operator
&	*and* logical on bits, binary
|	*or* logical on bits, binary
^	*exclusiv*e (*xor*) logical on bits, binary

We notice that operators `and/or` are written differently than logical ones that contain the double sign character. The operator `~` is unary, it is applied to an integer operand and the result is the one obtained by complementing the bits of the operand. Obviously, it has no effect on the operand, which can be any kind of expression.

Examples:

Code sequence	What is displayed	Explanation

`short a;`
`a = -1;`
`cout<<(~a);`
	`0`	The internal representation of `-1` is formed only of bits `1`. After complementing them, only bits `0` are obtained which represents the internal encoding of the value `0`.

`short a;`
`a = -236;`
`cout<<(~a);`
	`235`	The internal writing of `-236` is:
		`1111111100010100`
		Its negation:
		`0000000011101011` (`235`)

The `&` operator is binary, it is applied to two integer values and the result is what we get by applying "`and`" bit by bit between all pairs of bits in the same position. At "`and`" the result is `1` only if both "operands" are `1`.

Examples:

Code sequence	What is displayed	Explanation

`short a, b, c;`
`a = 101;`
`b = 87;`
`c = (a&b);`
`cout<<c;`
	`69`	`a: 0000000001100101`
		`b: 0000000001010111`
		`c: 0000000001000101`

`short a, b, c;`
`a = 501;`
`b = -236;`
`c = (a&b);`
`cout<<c;`
	`276`	`a: 0000000111110101`
		`b: 1111111100010100`
		`c: 0000000100010100`

If we imagine the writings in base 2 as being sets represented by a characteristic vector, we can say that after applying the operator & we obtain the intersection of the two sets.

The | operator is binary, it is applied to two integer values and the result is what we get by applying "or" bit by bit between all pairs of bits in the same position. In "or" the result is 1 only if there is 1 in at least one operand.
Examples:

Code sequence	What is displayed	Explanation

short a, b, c;
a = 101;
b = 87;
c = (a|b);
cout<<c;
	119	a: 0000000001100101
		b: 0000000001010111
		c: 0000000001110111

short a, b, c;
a = 501;
b = -236;
c = (a|b);
cout<<c;
	-11	a: 0000000111110101
		b: 1111111100010100
		c: 1111111111110101

If we imagine the writings in base 2 as being sets represented by a characteristic vector, we can say that following the application of the operator | the reunion of the two sets is obtained.

The operator ^ is binary, it is applied to two integer values and the result is what we obtain by applying "exclusive or" bit by bit between all pairs of bits in the same position. At "exclusive or" the result is *1* only if exactly *one* operand is 1.
Examples:

Code sequence	What is displayed	Explanation

short a, b, c;
a = 101;
b = 87;
c = (a^b);
cout<<c;
	50	a: 0000000001100101
		b: 0000000001010111
		c: 0000000000110010

short a, b, c;
a = 501;
b = -236;
c = (a^b);
cout<<c;
	-287	a: 0000000111110101
		b: 1111111100010100
		c: 1111111011100001

If we imagine the writings in base 2 as some sets represented by a characteristic vector, we can say that following the application of the operator ^ the symmetric difference of the two sets is obtained.

The shift operators

There are two shift operators, the left shift operator and the right shift operator. The operator << (moving left) is binary, both operands being integers. The result is obtained by translating to the left the bits of the left operand with as many positions as the value of the right operand. The values of the bits that were in front and are out of the memory area will be lost and the right side will be filled with 0.

Examples:

Code sequence	What is displayed	Explanation

short a, b;
a = 101;
b = (a << 3);
cout<<b;
	808	a: 0000000001100101
		b: 0000001100101000

short a, b;
a = -236;
b = (a << 2);
cout<<c;
	-944	a: 1111111100010100
		b: 1111110001010000

Here's a very important observation: the operation a << b is equivalent to $a * 2^b$. If we look at the first example, it is easy to prove what we said earlier. Moving to the left with 3 positions all the exponentials of the powers of 2 that made up the representation reach with 3 larger ones, that is, if at the new representation we give common factor 2^3 we get in parenthesis even the initial value.

$101 = 2^6 + 2^5 + 2^2 + 2^0$.

$808 = 101 << 3 = 2^9 + 2^8 + 2^5 + 2^3 = 2^3 * (2^6 + 2^5 + 2^2 + 2^0) = 2^3 * 101$.

The >> operator (moving to the right) is binary, both operands being integers. The result is obtained by translating to the right the bits of the left operand with as many positions as the value of the right operand. The values of the bits that were in the back and are out of the memory area will be lost and the left side will be filled with the sign bit. So, in the negative numbers it will be filled with 1 (on all the necessary positions), and in the positive numbers, analogous, with 0. Thus, the sign of the result will be the same as the sign of the left operand. With a similar reasoning to the one shown to the left shift operator, we deduce that the shift to the right is equivalent to the division to a power of 2. More precisely, the expression a >> b is equivalent to $a / 2^b$. Here is just the quotient of the division (bits of 1 can be lost on the right when the division is not complete).

Examples:

Code sequence	What is displayed	Explanation

`short a, b;`
`a = 101;`
`b = (a >> 1);`
`cout<<b;`
	`50`	`a: 0000000001100101`
		`b: 0000000000110010`

`short a, b;`
`a = -236;`
`b = (a >> 2);`
`cout<<c;`
	`-59`	`a: 1111111100010100`
		`b: 1111111111000101`

Bitwise operators (all except ~) can be combined with the assignment operator to obtain new shortcut operators. Thus, you can write form expressions:

variable op = expression

where op can be: `&`, `|`, `^`, <<, >>.

Exemple: `n = (n >> 1)` may be written as `n>>=1`

Pay attention to the use of bitwise operators in expressions with other operators as they have very low application priority and so it is recommended to use as many brackets as possible for grouping.

Using bitwise operators in expressions leads to increased running speed because for each of them the processor has its own instructions.

Here are some things we can do elegantly using bit operators.

`1<<k`	This expression is equivalent to 2^k. By using the operator we obtain this result directly compared to the classical calculation that requires repetition.
`1LL<<k`	Thus it must be written if we want to obtain powers of `2` greater than `32`. In the above expression constant `1` was of type int (so it is implicit in C / C ++) so we must put the suffix to force the representation of `1` on `64` bits.
`n>>1`	Equivalent to `n / 2`
`n&1`	This expression is equivalent to `n%2`. This means we can test the parity of a number this way. Here is the justification: the number `1` has only bits `0`, except the least significant bit which is `1`. Putting 1 as an operand next to `&` all bits of the result, minus the last one, will be `0`. The last bit is obtained as `&` between bit `1` (from number `1`) and the last bit of the other operand. So you get `1` only if that bit is `1`, too.
`n^n`	`0`
`n^n^n`	`n` (`xor` applied by an even number of times on the same value leads to `0`, and `xor` applied by an odd number of times on the same value leads to that value).
`n^0`	`n`

`(n >> k) & 1`	This is the expression we write to get the value of bit `k` of `n`. In practice, by shifting we ”take” that bit to the last position, and according to the previous example, it remains to apply `&` with the number `1`.

`n = n + (1<<k)`
or
`n += (1<<k)`
	This instruction adds to `2k` the value `n`. If the bit at position `k` of `n` is `0`, this expression has the effect of setting that value to `1` in `n` of that bit. Note that if the bit in position `k` is already `1`, it becomes `0` and more significant bits will be affected as transport to the assembly operation in base `2` appears.

`n = (n | (1<<k))`
or
`n |= (1<<k)`
	This is the expression that sets to `1` the bit value at position `k` in `n` regardless of its previous value, the rest of the bits remaining unchanged. If we imagine the binary representation of `n` as a characteristic vector associated with a set, by this operation we add (if it does not already exist), to the set the element `k`.
	Here, more detailed, we prove what happens:
	`k = 4`
	`n: XXXXXXXXXXXXXXXX`
	`1: 0000000000000001`
	`1<<k 0000000000010000`
	At ”`or`” with bit `1`, at that position the obtained bit will be `1`, and the others remain unchanged since `0` is a neutral element in this operation.

`n = (n & ~(1<<k))`
or
`n &= ~(1<<k)`
	These are ways to set to `0` a certain bit of a variable, regardless of its previous value and without affecting the values of the other bits. If we knew for sure that the bit `k` is `1`, to make it `0` we could use the expression `n - = (1 << k)`
	Here, more detailed,it is what happens:
	`k = 4`
	`n: XXXXXXXXXXXXXXXX`
	`1: 0000000000000001`
	`1<<k 0000000000010000`
	`~(1<<k) 1111111111101111`
	After the operation `&` with bit `0`, `0` is obtained, and otherwise no changes occur as `1` is a neutral element in ”`and`”.

`n & -n`	The value of this expression is always a power of `2`. It represents the least significant bit of `n`. If we analyze the complement rule for `2` in order to find the absolute value for a negative number, we notice that that is the only bit that remains `1` at both `n` and `- n` (the remaining positions have a maximum operand `1`).

`for (;n;n-=(n&-n))`
`{`
`  cnt++;`
`}`
	This code sequence counts in cnt how many bits of `1` it has `n` written in base `2`. It is observed that at every step it is eliminated from its least significant bit. It is interesting that the number of repetitive steps is equal to this value. In the following example we will also present the raw variant that checks the value of each bit separately.

	We emphasize that the current code sequence is also useful in implementing the data structure called binary indexed trees.

`for (i=15;i>=0;i--)`
`  if ((n>>i)&1)`
`  cnt++;`
	This is the sequence equivalent to the previous one but the number of steps is always equal to the total number of bits of the internal representation of `n`.

Exercises and solved problems

Problem	Solution
1. For a given variable of type `int`, write an expression that gives the value obtained with its last `k` bits (and the value `k` is given).	The operator `&` is applied between the value of `n` and the one with `1` on the last `k` positions and in the rest `0`. We have shown above that such values are powers of `2` of which is subtracted `1`. We obtain this: `n&((1LL<<k)-1).`
2. For a given variable of type `int`, write an expression that gives the value obtained with its bits from positions `6,7,8`.	We reduce this problem to the previous one by moving the three bits so that they reach the last three positions. Then we apply the above code for `k = 3`. `(n>>6)&7.` We chose to use directly `7(1112)` being equivalent to `(1LL<<k)-1)` for `k = 3.`
3. Write an expression that sets to `1` the last `2` bits of the value of an `int` variable, regardless of their previous value. The rest of the bits must keep their value.	`n |= 3` I made "`or`" with the number that has `1` on the last two positions, in the rest `0`. This is exactly `3`.
4. Write an expression that sets to `0` the last `2` bits of the value of an `int` - type variable, regardless of their previous value. The rest of the bits must keep their value.	`n &= (-1 << 2)` Here we use "`and`" the other operand having a value that has `0` on the last two positions and subtract `1`. We obtained it using the fact that `-1` contains only bits of `1` and the shift to the right introduces `0` at the end.
5. Given a variable of type `short`, write a sequence of instructions for the interchange of its two bytes (the sequence of the most insignificant `8` bits must arrive, in the same order on the most significant `8` bits, and vice versa).	`n = ((n&255)<<8)+((n>>8)&255)` Number `255` represents the value with the last `8` bits `1`. The first term obtains the value with the last `8` bits and by moving it `8` positions to the left, it adds practically `8` to `0` at the end, preparing it to add the value with the first `8` bits (obtained in the the second term by an "`and`" with `255` after going to the last 8 positions, by moving with `8` positions to the right).

6. Write a sequence of assignments that interchanges the values of two variables `a` and `b`, using the `xor` bitwise operator and without additional variables.	`a=a^b;`
	`b=a^b;`
	`a=a^b;`

Problem: We give n and n non-negative values and we ask to say for each how many bits of 1 it has in the internal representation. The given values are 32 bits and the number of given values is up to 10^6.

Solution:

The first solution is to apply the raw algorithm that tests each of the 32 bits of each number in turn. Thus, the number of steps taken is of the order n * 32 (32 = the number of bits on which the values are represented).

Another solution is to use the optimized alternative to obtain the number of bits 1 of a value (the one by which we always eliminate the least significant bit of 1 and count, that is n - = (n & -n)). However, if all given numbers have large number of bits 1, the execution time tends towards the one from the previous version.

There is also a third, much faster solution, which we will present below. It is based on the precalculation, for certain numbers, of the number of bits and the use of this information. Calculating how many bits of 1 has each value on 32 bits is not practical, because they are of the order of 4 billion values and both the time to realize this and the memory needed to store all the values would be too large. We will precalculate the number of bits 1 for each possible number on 16 bits and then see how we use this information.

On 16 bits there are 65536 possible values, that is the memory used is not big and we will see in the count that neither the time used. If we note B [i] = the number of bits 1 of the writing in the base 2 of i, we have:

B[0] = 0;

B[1] = 1;

B[i] = B[i/2] if i is even;

B[i] = 1+B[i/2] if i is odd.

This is because the value i / 2 contains exactly the same bits as i, except the last bit of i (i / 2 is equivalent to i >> 1 and it is possible to lose a bit of 1, in which case the number i would be odd and then the value 1 will be added to the result).

Therefore, the precalculation is as follows:

```
b[0] = 0;
b[1] = 1;
for (i=2;i<=(1<<16)-1;i++)
    b[i] = b[i/2] + i%2; /// or b[i] = b[i>>1] + (i&1);
```

If all given numbers are on 16 bits, we can use precalculation and solve the problem as follows:

```
cin>>n;
for (i=1;i<=n;i++) {
    cin>>a;
    cout<<b[a];
}
```

But we can do a little trick and solve the problem also for 32-bit numbers:

```
cin>>n;
for (i=1;i<=n;i++) {
    cin>>a;
    cout<<b[a & 0xFFFF] + b[(a>>16) & 0xFFFF];
}
```

What is the meaning ?

The value `0xFFFF` represents a `16`-bit number, all of `1`, that is (`216-1`), I could also write `(1 << 16) -1`, as we have used so far in the article. This is because `F` (`15`) is written as `1111`. Therefore, writing `a & 0xFFFF` gives us the value that is obtained by taking into account only the last `16` bits of `a`, so accessing `B` at this position we obtain as many as `1` among the last `16` bits of `a`. It remains to add the number of bits with the value of `1` of the first `16` bits of `a`. We notice that if we move to the right with `16` positions, the bits that interest us remain the last `16` so we can use on the new values formula from the other `16` - bit group.

We can say that we now answer directly for each given number, using an expression, without additional iterations.

Conclusions

A number of common operations (such as those with powers of `2`) can be performed very quickly using bitwise operators, often reducing the use of a repetitive code sequence when evaluating a single expression. The usual operations with sets can also be done elegantly by using bitwise operators, obtaining besides good execution time a reduction in the memory used, too. For example, we can represent a set that can have values between `0` and `31` through a single int type date, this having the characteristic vector meaning for the set. Thus, the intersection operation of two sets can be performed by applying the bitwise operator `&` to the variables that encode the set and the meeting operation by applying the operator `|`.

References

1. King K.N. C Programming: A Modern Approach, 2th Edition. W. W. Norton & Company, New-York, 2008. 832 p.
2. Kelley A., Pohl I. A Book on C: Programming in C, 4th edition. Addison-Wesley Professional, USA, 1998. 752 p.
3. Kochan S.G. Programming in C, 4th Edition. Addison-Wesley Professional, USA, 2014. 600 p.