Sudipta Acharjee, 2023, 11:4
ISSN (Online): 2348-4098
ISSN (Print): 2395-4752

Study of Bitwise Operations in C/C++

Assistant Professor Sudipta Acharjee
Dept. of Computer Engineering
GND DSEU Rohini Campus, Rohini, Delhi
sudipta.acharjee@dseu.ac.in

Abstract- Bitwise operators are characters that represent actions (bitwise operations) to be performed on single bits. They operate at the binary level and perform operations on bit patterns that involve the manipulation of individual bits.Bitwise operations in C/C++ are asked frequently in programming interviews as well as competitive programming. Therefore, it's essential to practice problems that use a variety of approaches and algorithms.In C/C++, bitwise operators perform operations on integer data at the individual bit-level. These operations include testing, setting, or shifting the actual bits.We use the bitwise operators in C language to perform operations on the available data at a bit level. Thus, performing a bitwise operation is also called bit-level programming. It is mainly used in numerical computations for a faster calculation because it consists of two digits – 1 or 0.In this article, we will take a look into the Bitwise Operators in C and C++ according to the use with the help of programming codes.In this article, we will learn about different types of operands and bitwise operators in C/C++, their functionality and examples of how to use them.

Keywords- Bitwise Operators, bit, binary.

I. INTRODUCTION

In the C programming language, operations can be performed on a bit level using bitwise operators. Bit wise operations are contrasted by byte-level operations which characterize the bitwise operators' logical counterparts, the AND, OR, NOT operators. Instead of performing on individual bits, byte-level operators perform on strings of eight bits (known as bytes) at a time.

The reason for this is that a byte is normally the smallest unit of addressable memory (i.e. data with a unique memory address)This applies to bitwise operators as well, which means that even though they operate on only one bit at a time they cannot accept anything smaller than a byte as their input. of these operators are also available in C++, and many C-family languages.

Bit wise operators
Bi C provides six operators for bit manipulation.

Symbol	Operator
&	bitwise AND
|	bitwise inclusive OR
^	bitwise XOR (exclusive OR)
<<	left shift
>>	right shift
~	bitwise NOT (one's complement) (unary)

Bitwise AND &

bit a	bit b	a & b (a AND b)

International Journal of Science, Engineering and Technology

An Open Access Journal

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

The bitwise AND operator is a single am persand: `&`. It is just a representation of AND which does its work on the bits of the operands rather than the truth value of the operands. Bitwise binary AND performs logical conjunction (shown in the table above) of the bits in each position of a number in its binary form. For instance, working with a byte (the char type):

```
  11001000
& 10111000
  --------
= 10001000
```

The most significant bit of the first number is 1 and that of the second number is also 1 so the most significant bit of the result is 1; in the second most significant bit, the bit of second number is zero, so we have the result as 0. [2]

Bitwise OR `|`

bit a	bit b	a | b (a OR b)
0	0	0
0	1	1
1	0	1
1	1	1

Similar to bitwise AND, bitwise OR performs logical disjunction at the bit level. Its result is a 1 if either of the bits is 1 and zero only when both bits are 0. Its symbol is `|` which can be called a pipe.

```
  11001000
| 10111000
  --------
= 11111000
```

[2]

Bitwise XOR `^`

bit a	bit b	a ^ b (a XOR b)
0	0	0
0	1	1
1	0	1
1	1	0

The bitwise XOR (exclusive or) performs an exclusive disjunction, which is equivalent to adding two bits and discarding the carry. The result is zero only when we have two zeroes or two ones.[3] XOR can be used to toggle the bits between 1 and 0. Thus `i = i ^ 1` when used in a loop toggles its values between 1 and 0.[4]

```
  11001000
^ 10111000
  --------
= 01110000
```

Shift operators

There are two bitwise shift operators. They are

- Right shift (>>)
- Left shift (<<)Right shift `>>`

The symbol of right shift operator is `>>`. For its operation, it requires two operands. It shifts each bit in its left operand to the right. The number following the operator decides the number of places the bits are shifted (i.e. the right operand). Thus by doing `ch>> 3` all the bits will be shifted to the right by three places and so on.However, do note that a shift operand value which is either a negative number or is greater than or equal to the total number of bits in this value results in undefined behaviour. For example, when shifting a 32 bit unsigned integer, a shift amount of 32 or higher would be undefined.

Example:

If the variable `ch` contains the bit pattern `11100101`, then `ch>> 1` will produce the result `01110010`, and `ch>> 2` will produce `00111001`.

Here blank spaces are generated simultaneously on the left when the bits are shifted to the right. When performed on an unsigned type or a non-negative value in a signed type, the operation performed is a logical shift, causing the blanks to be filled by `0`s (zeros). When performed on a negative value in a

signed type, the result is technically implementation-defined (compiler dependent),[5] however most compilers will perform an arithmetic shift, causing the blank to be filled with the set sign bit of the left operand. Right shift can be used to divide a bit pattern by 2 as shown:

```
i=14;// Bit pattern 00001110
j=i>>1;// here we have the bit pattern shifted by 1 thus we get 00000111 = 7 which is 14/2
```

Right shift operator usage

Left shift `<<`

The symbol of left shift operator is `<<`. It shifts each bit in its left-hand operand to the left by the number of positions indicated by the right-hand operand. It works opposite to that of right shift operator. Thus by doing `ch<< 1` in the above example (`11100101`) we have `11001010`. Blank spaces generated are filled up by zeroes as above.

However, do note that a shift operand value which is either a negative number or is greater than or equal to the total number of bits in this value results in undefined behaviour. This is defined in the standard at ISO 9899:2011 6.5.7 Bit-wise shift operators. For example, when shifting a 32 bit unsigned integer, a shift amount of 32 or higher would be undefined.

Left shift can be used to multiply an integer by powers of 2 as in

```
inti=7;// Decimal 7 is Binary (2^2) + (2^1) + (2^0) = 0000 0111
intj=3;// Decimal 3 is Binary (2^1) + (2^0) = 0000 0011
k=(i<<j);// Left shift operation multiplies the value by 2 to the power of j in decimal
// Equivalent to adding j zeros to the binary representation of i
// 56 = 7 * 2^3
```

```
// 0011 1000 = 0000 0111 << 0000 0011
```

Example: a simple addition program

The following program adds two operands using AND, XOR and left shift (<<).

```
#include<stdio.h>

intmain(void)
{
unsignedintx=3,y=1,sum,carry;
sum=x^y;// x XOR y
carry=x&y;// x AND y
while(carry!=0)
{
carry=carry<<1;// left shift the carry
x=sum;// initialize x as sum
y=carry;// initialize y as carry
sum=x^y;// sum is calculated
carry=x&y;/* carry is calculated, the loop condition is
                evaluated and the process is repeated until
                carry is equal to 0.
               */
}
printf("%u\n",sum);// the program will print 4
return0;
}
```

Bitwise assignment operators

C provides a compound assignment operator for each binary arithmetic and bitwise operation. Each operator accepts a left operand and a right operand, performs the appropriate binary operation on both and stores the result in the left operand.[6]

The bitwise assignment operators are as follows.

Symbol	Operator
`&=`	bitwise AND assignment
`|=`	bitwise inclusive OR assignment
`^=`	bitwise exclusive OR assignment
`<<=`	left shift assignment
`>>=`	right shift assignment

Logical equivalents

Four of the bitwise operators have equivalent logical operators. They are equivalent in that they have the same truth tables. However, logical operators treat each operand as having only one value, either true or false, rather than treating each bit of an operand as an independent value. Logical operators consider zero false and any nonzero value true. Another difference is that logical operators perform short-circuit evaluation.

The table below matches equivalent operators and shows a and b as operands of the operators.

Bitwise	Logical
`a & b`	`a && b`

a | b	a || b
a ^ b	a != b
~a	!a

!= has the same truth table as ^ but unlike the true logical operators, by itself != is not strictly speaking a logical operator. This is because a logical operator must treat any nonzero value the same. To be used as a logical operator != requires that operands be normalized first. A logical not applied to both operands won't change the truth table that results but will ensure all nonzero values are converted to the same value before comparison. This works because ! on a zero always results in a one and ! on any nonzero value always results in a zero.
Example:

```
/* Equivalent bitwise and logical operator tests */
#include<stdio.h>
Void test Operator(char*name ,unsigned cha rwas, unsigned char expected);

Int main (void)
{
// -- Bitwise operators -- //

//Truth tables packed in bits
constunsignedcharoperand1=0x0A;//0000 1010
constunsignedcharoperand2=0x0C;//0000 1100
constunsignedcharexpectedAnd=0x08;//0000 1000
constunsignedcharexpectedOr=0x0E;//0000 1110
constunsignedcharexpectedXor=0x06;//0000 0110

constunsignedcharoperand3=0x01;//0000 0001
constunsignedcharexpectedNot=0xFE;//1111 1110

testOperator("Bitwise
AND",operand1&operand2,expectedAnd);
testOperator("Bitwise
OR",operand1|operand2,expectedOr);
testOperator("Bitwise
XOR",operand1^operand2,expectedXor);
testOperator("Bitwise
NOT",~operand3,expectedNot);
printf("\n");

// -- Logical operators -- //

constunsignedcharF=0x00;//Zero
constunsignedcharT=0x01;//Any nonzero value

// Truth tables packed in arrays

constunsignedcharoperandArray1[4]={T,F,T,F};
constunsignedcharoperandArray2[4]={T,T,F,F};
constunsignedcharexpectedArrayAnd[4]={T,F,F,F};
constunsignedcharexpectedArrayOr[4]={T,T,T,F};
constunsignedcharexpectedArrayXor[4]={F,T,T,F};

constunsignedcharoperandArray3[2]={F,T};
constunsignedcharexpectedArrayNot[2]={T,F};

inti;
for(i=0;i<4;i++)
{
testOperator("Logical
AND",operandArray1[i]&&operandArray2[i],expected
ArrayAnd[i]);
}
printf("\n");

for(i=0;i<4;i++)
{
Test Operator ("Logical  OR",oper and Array1 [i]|| oper and Array 2 [i],expected Array Or[i]);
}
printf("\n");

for(i=0;i<4;i++)
{
//Needs ! on operand's in case nonzero values are different
testOperator("Logical
XOR",!operandArray1[i]!=!operandArray2[i],expected
ArrayXor[i]);
}
printf("\n");

for(i=0;i<2;i++)
{
testOperator("Logical
NOT",!operandArray3[i],expectedArrayNot[i]);
}
printf("\n");

return0;
}

Voidtest Operator ( char*name, unsigned char was, unsigned char expected)
{
char*result=(was==expected)?"passed":"failed";
```

```
printf("%s %s, was: %X expected: %X \n",name ,result
,was ,expected);
}
```

The output of the above program will be

```
Bitwise AND passed, was: 8 expected: 8
Bitwise  OR passed, was: E expected: E
Bitwise XOR passed, was: 6 expected: 6
Bitwise NOT passed, was: FE expected: FE

Logical AND passed, was: 1 expected: 1
Logical AND passed, was: 0 expected: 0
Logical AND passed, was: 0 expected: 0
Logical AND passed, was: 0 expected: 0

Logical  OR passed, was: 1 expected: 1
Logical  OR passed, was: 1 expected: 1
Logical  OR passed, was: 1 expected: 1
Logical  OR passed, was: 0 expected: 0

Logical XOR passed, was: 0 expected: 0
Logical XOR passed, was: 1 expected: 1
Logical XOR passed, was: 1 expected: 1
Logical XOR passed, was: 0 expected: 0

Logical NOT passed, was: 1 expected: 1
Logical NOT passed, was: 0 expected: 0
```

```
//Example of Bitwise Operators
#include<stdio.h>
#include<conio.h>
int main()
{
int a=5,b=6,c;
clrscr();
/*Swapping using bitwise operator*/
printf("\nBefore Swapping A = %d  B = %d",a,b);
a=a^b;   /*bitwise Exclusive OR*/
b=a^b;
a=a^b;
printf("\nAfter Swapping A = %d  B =  %d",a,b);
//Bitwise leftshift
a=a<<1;
printf("\nBitwise Left of A variable = %d ",a);
a=a>>1;
printf("\nBitwise Right of A variable = %d ",a);
getch();
return 0;
}
```

Output:

```
Before Swapping A = 5 B = 6
After Swapping A = 6 B = 5
Bitwise Left of A variable = 12
Bitwise Right of A variable = 6
```

```
/*This Program will compute the binary pattern of a
number*/
#include<stdio.h>
#include<conio.h>
#include<dos.h>
int main()
{
int n,i=15,j,p=1;/*Declaring Input Variables*/
clrscr(); /*Clear the screen*/
printf("\nEnter the value of N :: ");
scanf("%d",&n);
printf("\nBinary pattern of %d = ",n);
while(i>=0)
 /*While Loop for 16 bits integer scanning*/
{
j=p<<i;
delay(100); /*Propagation delay of 100 miliseconds*/
                /*Bitwise left shift ith times*/
(j&n)?printf("1"):printf("0"); /*Conditional operators
checking*/
i--;                                /*the binary value of j
with n*/
}/*Ternary Operator*/
getch();
return 0;
}
```

Output:

```
Enter the value of N :: 5
Binary pattern of 5 = 0000000000000101
```

REFERENCES

1. R Kernighan; Dennis M. Ritchie (March 1988). The C Programming Language (2nd ed.). Englewood Cliffs, NJ: Prentice Hall. ISBN 0-13-110362-8. Archived from the original on 2019-07-06. Retrieved 2019-09-07. Regarded by many to be.
2. Jump up to:a b "Tutorials - Bitwise Operators and Bit Manipulations in C and C++". Cprogramming .com.
3. "Exclusive-OR Gate Tutorial". Basic Electronics Tutorials.
4. "C++ Notes: Bitwise Operators". fredosaurus.com.
5. "ISO/IEC 9899:2011 - Information technology -- Programming languages -- C". www.iso.org.

6. "Compound assignment operators". IBM. International Business Machines. Retrieved 29 January 2022.

External links

- Bitwise Operators
- Demystifying bitwise operations, a gentle C tutorial