**Bitwise Operators in C Programming**

In arithmetic-logic unit (which is within the CPU), mathematical operations like: addition, subtraction, multiplication and division are done in bit-level. To perform bit-level operations in C programming, bitwise operators are used.

| Operators | Meaning of operators |
| --- | --- |
| & | [Bitwise AND](https://www.programiz.com/c-programming/bitwise-operators#and) |
| | | [Bitwise OR](https://www.programiz.com/c-programming/bitwise-operators#or) |
| ^ | [Bitwise XOR](https://www.programiz.com/c-programming/bitwise-operators#xor) |
| ~ | [Bitwise complement](https://www.programiz.com/c-programming/bitwise-operators#complement) |
| << | [Shift left](https://www.programiz.com/c-programming/bitwise-operators#left-shift) |
| >> | [Shift right](https://www.programiz.com/c-programming/bitwise-operators#right-shift) |

## Python Bitwise Operations Examples

## Bitwise AND operator &

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bit Operation of 12 and 25

00001100

& 00011001

\_\_\_\_\_\_\_\_

00001000 = 8 (In decimal)

#include <stdio.h>

int main()

{

    int a = 12, b = 25;

    printf("Output = %d", a&b); // and operation bt a and b

    return 0;

}

//Output = 8

Example #2: Bitwise OR

#include <stdio.h>

int main()

{

    int a = 12, b = 25;

    printf("Output = %d", a|b); //or operation bt a and b

    return 0;

}

//Output = 29

## Bitwise XOR (exclusive OR) operator ^

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise XOR Operation of 12 and 25

00001100

^ 00011001

\_\_\_\_\_\_\_\_

00010101 = 21 (In decimal)

#include <stdio.h>

int main()

{

    int a = 12, b = 25;

    printf("Output = %d", a^b);//xor operation between a and b

    return 0;

}

//Output = 21

## Bitwise complement operator

#include <stdio.h>

int main()

{

    printf("Output = %d\n",~5);  //n   -(n+1)

    printf("Output = %d\n",~-12);//-12    -(-12+1)   -(-11)   11

    return 0;

}

//Output = -6

Output = 11

## Shift Operators in C programming

There are two shift operators in C programming:

* Right shift operator
* Left shift operator.

### Right Shift Operator

Right shift operator shifts all bits towards right by certain number of specified bits. It is denoted by >>.

212 = 11010100 (In binary)

212>>2 = 00110101 (In binary) [Right shift by two bits]

212>>7 = 00000001 (In binary)

212>>8 = 00000000

212>>0 = 11010100 (No Shift)

## Left Shift Operator

Left shift operator shifts all bits towards left by a certain number of specified bits. The bit positions that have been vacated by the left shift operator are filled with 0. The symbol of the left shift operator is <<.

212 = 11010100 (In binary)

212<<1 = 110101000 (In binary) [Left shift by one bit]

212<<0 = 11010100 (Shift by 0)

212<<4 = 110101000000 (In binary) =3392(In decimal)

#include <stdio.h>

int main()

{

    int num=212, i;

    for (i=0; i<=2; ++i)

        printf("Right shift by %d: %d\n", i, num>>i);//right shifting

     printf("\n");

     for (i=0; i<=2; ++i)

        printf("Left shift by %d: %d\n", i, num<<i); //left shifting

     return 0;

}

//

Right shift by 0: 212

Right shift by 1: 106

Right shift by 2: 53

Left shift by 0: 212

Left shift by 1: 424

Left shift by 2: 848

memory layout of C program

A typical memory representation of a C program consists of the following sections.  
1. Text segment   
2. Initialized data segment   
3. Uninitialized data segment   
4. Stack   
5. Heap



A typical memory layout of a running process

**1. Text Segment:**   
A text segment, also known as a code segment or simply as text,

is one of the sections of a program in an object file or in memory,

which contains executable instructions.

As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it. 

Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

**2. Initialized Data Segment:**

Initialized data segment, usually called simply the Data Segment.

A data segment is a portion of the virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Note that, the data segment is not read-only, since the values of the variables can be altered at run time.

This segment can be further classified into the initialized read-only area and the initialized read-write area.

For instance, the global string defined by char s[] = “hello world” in C and a C statement like int debug=1 outside the main (i.e. global) would be stored in the initialized read-write area.

And a global C statement like const char\* string = “hello world” makes the string literal “hello world” to be stored in the initialized read-only area and the character pointer variable string in the initialized read-write area.

Ex: static int i = 10 will be stored in the data segment and global int i = 10 will also be stored in data segment

**3. Uninitialized Data Segment:**   
Uninitialized data segment often called the “**bss**” segment, named after an ancient assembler operator that stood for “**block started by symbol**.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing

uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

For instance, a variable declared static int i; would be contained in the BSS segment.   
For instance, a global variable declared int j; would be contained in the BSS segment.

**4. Stack:**

The stack area traditionally adjoined the heap area and grew in the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted.

Stack, where automatic variables are stored, along with information that is saved each time a function is called.

such as some of the machine registers, are saved on the stack.

 Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the

**5. Heap:**   
Heap is the segment where dynamic memory allocation usually takes place.

The heap area begins at the end of the BSS segment and grows to larger addresses from there. The Heap area is managed by malloc, realloc, and free,

it prevents overlap and allows both areas to grow freely as long as you have enough address space available.