Bitwise Operators & Loops

Date: .....
Page: .....

+ Bitwise operators are used to perform bit-level operations.

operator	description	operator	description
&	Bitwise AND	<<	Bitwise left shift
|	" OR	>>	" right "
^	" XOR	>>>	unsigned Right shift.
~	" Complement		

1) **Bitwise AND operator**

Return 1 if & only if both the operands are 1.

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

Eg:- Bitwise AND of integer 2 & 3

2 : 00000010
3 : 00000011 &
00000010 ⇒ 2

∴ 2 & 3 = 2

2) **Bitwise OR Operator**

Return 1 if atleast one of the operand is 1.

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

Eg:- Bitwise OR of 7 & 4

7 = 00000111
4 = 00000100 |
00000111 ⇒ 7

∴ 7 | 4 = 7

3) **Bitwise XOR operator**

Return 1 if & only if one of the operands is 1. However, if both the operands are 0 or 1 then the result is 0.

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

Date: .....
Page: .....

4) **Bitwise NOT (~) operator**

It changes binary digits 1 to 0 & 0 to 1

a	~a
0	1
1	0

~ = Tilda
0 = off (false)
1 = ON (True)

Q Find unique element (use XOR operator)

→ Example of Bitwise AND, OR, NOT & XOR operator.

→
```
int main () {
    cout << (2 & 3);
    return 0;
}
```
Output = 2

2 = 0 0 1 0
3 = 0 0 1 1 &
0 0 1 0 => 2

∴ 2 & 3 = 2

→
```
int main () {
    cout << (5 & 10);
    return 0;
}
```
Output = 0

5 = 0 1 0 1
10 = 1 0 1 0 &
0 0 0 0 => 0

∴ 5 & 10 = 0

→
```
int main () {
    cout << (5 | 10);
    return 0;
}
```
Output = 15

5 = 0 1 0 1
10 = 1 0 1 0 OR (|)
1 1 1 1 => 15

∴ 5 | 10 = 15

→
```
int main () {
    cout << (2 | 3);
}
```

2 = 0 0 1 0
3 = 0 0 1 1 OR (|)
0 0 1 1 => 3

Date: .....

Page: .....

→
```
int main () {
    int num = 1;
    cout << ~num << endl;
    cout << (~num) << endl;
    cout << ~(num) << endl;
}
```

outPut = -2
-2
-2

∴ That's why cout << (~num), cout << ~num & cout << ~(num) Gives outPut as -2.

→ 1 = 00000001

~1 = 11111110

Here, the most left bit is 1. That means it's a -ve number. To access negative number, we've to do 2's complement

1's complement = 0 → 1 & 1 → 0

11111110 = 00000001

2's complement = Add 1 in 1's complement

```
  00000001
+       1
  --------
  00000010 => 2
```

→
```
int main () {
    cout << (5^5);
    return 0;
}
```

outPut = 0

→ 5 = 0101

5 = 0101 XOR (^)

0000 ⇒ 0

∴ (5^5) = 0

→
```
int main () {
    cout << (5^10);
    return 0;
}
```

outPut = 15

→ 5 = 0101

10 = 1010 XOR (^)

1111 ⇒ 15.

∴ (5^10) = 15

→
```
int main () {
    cout << (4^7);
}
```

→ 4 = 0100

7 = 0111 XOR (^)

0011 ⇒ 3

Date: .....
Page: .....

→ int main() {
    cout << (5^-5);
    return 0;
}

Output : -2

5 = 0 0 0 0 0 1 0 1
1's = 1 1 1 1 1 0 1 0
2's = +1
1 1 1 1 1 0 1 1 ⇒ -5

∴ 2's complement of 5 is -5

→ 5 = 0 0 0 0 0 1 0 1
-5 = 1 1 1 1 1 0 1 1 XOR (^)
(1) 1 1 1 1 1 1 0

→ This MSB shows that XOR of 5 & -5 is a negative number

1's complement of 5^-5
= 0 0 0 0 0 0 0 1
2's complem of (5^-5) 1's complement
= 0 0 0 0 0 0 0 1
+1
0 0 0 0 0 0 1 0 ⇒ 2

That's why (5^-5) output is -2

→ **Bitwise Left & Right shift operators.**

1> **Left Shift operator :**

The leftshift operator shift all bits towards the left by a certain no. of specified bits.

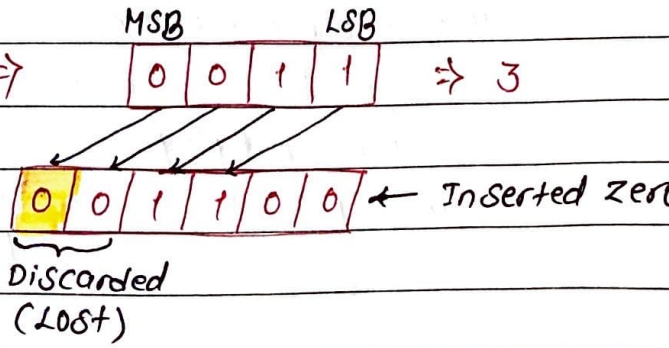
NOTE : Left shifting a number by certain positions is equivalent to multiplying the number by two raised to the power of the specified position. i.e.

leftshift x by n positions = $x \times 2^n$

Eg :- leftshift 3 by 2 positions (3 << 2) ⇒ $3 \times 2^2 = 12$

3 = 0 0 0 0 0 0 1 1
↓
0 0 0 0 1 1 0 0 = 12

∴ 3 << 2 = 12



Date: .....
Page: .....

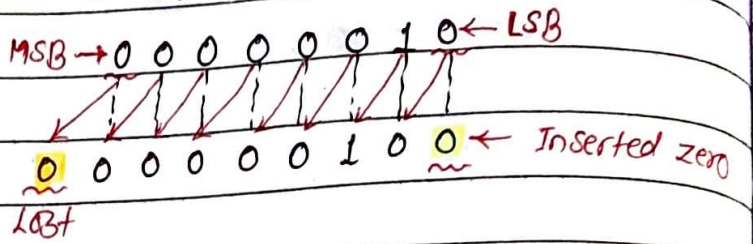
```
int main () {
    int a = 2;
    cout << (a << 1);
    return 0;
}
```

Output = 4

∴ 2 << 1 = 4.

⟶ a = 2 ⇒ 00000010

a << 1 = a is left shifted by 1 bit.



⇒ 00000100 = 4.

When shifting left, the most-significant (Left side) bit is lost & a '0' bit is inserted to the other end.

2) **Right shift operator :**

The rightshift operator shift all bits towards the right by a certain no. of specified bits.

NOTE: i) when bits are shifted right then leading positions are filled with zeros.

ii) Right shifting a number by certain position is equivalent to dividing the number by two raised to the power of the specified position. i.e.

Right shift x by n positions = $x/2^n$

Eg:- Right Shifting 14 by 1 position (14 >> 1) ⇒ $14/2^1 = 7$

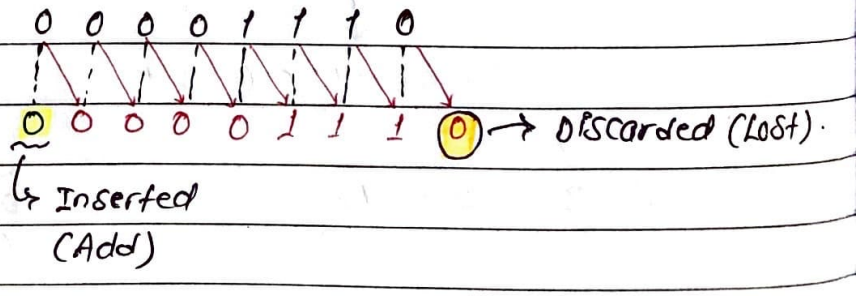
14 = 00001110

↓

00000111

∴ 14 >> 1 = 7

⇒

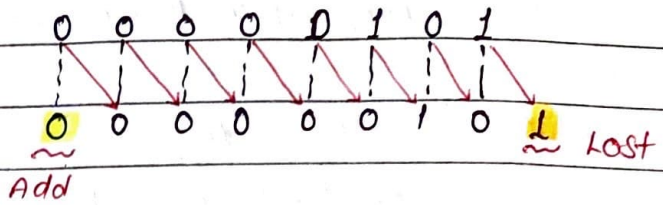


Date: .....
Page: .....

→
```
int main () {
    int n = 5;
    cout << (n >> 1);
    return 0;
}
```

n = 5 ⇒ 0 0 0 0 0 1 0 1

n >> 1 = n is right shifted by 1 bit



output = 2

∴ 5 >> 1 = 2

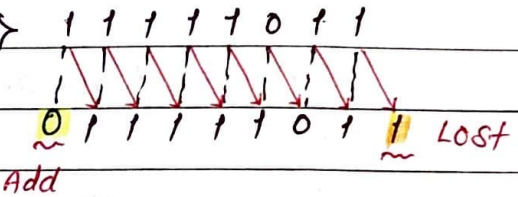
⇒ 0 0 0 0 0 0 1 0 = 2

When shifting right, the least-significant (Right-side) bit is lost & a '0' is inserted on other end.

→ Let see what happen when right shift a negative number.

```
int main () {
    int n = -5;
    cout << (n >> 1);
    return 0;
}
```

n = -5 ⇒ 1 1 1 1 1 0 1 1



output = -2

The msb shows that it is a positive number, but our output is a negative number.

→ In case of ^-ve' signed integer (hold +ve, -ve or 0) shifting a bit either left or right is handled by compiler.

But, in case of ^-ve unsigned integer (hold +ve, 0) shifting a bit right is not handled by compiler & it giving large number as output.

Eg:-
```
int n = -100;
cout << (n >> 1);
```
output = -50

(In case of signed integer) ↓ -ve

```
unsigned int n = -100;
cout << (n >> 1);
```
output = 2147483598

(Large number, in case of -ve unsigned integer)

Date: .....
Page: .....

→ When we shift a number ~~eg~~ either left or right by a negative number, then it throws warning & gives a garbage value, Remember, it will not show error.

Eg:- int n = 10;
cout << (n << -1);
output : 1346981272 (Garbage value)

→ Unary operators :

operator	operation	Equivalent To
++ (increment)	num++	num = num+1
-- (decrement)	num--	num = num-1

→ Pre & Post Increment operators.

Pre - Increment : ++a;
↳ first value change & then value use.

Post - Increment : a++;
↳ first old value use & then value change.

→ Pre & Post Decrement operators.

Pre - Decrement : --a;
↳ first value change & then new value use.

Post - Decrement : a--;
↳ first old value use & then value change.

Eg:- int a = 10; output : 11, 11.
int b = ++a;
cout << a;
cout << b;

Date: .....
Page: .....

```
int a = 10;
int b = ++a; (a = a+1)
    b = a = a+1
    b = a = 10+1
    b = a = 11;
```

Here, we've Pre-Increment before the operand (i.e. ++a). So the value of 'a' is first increment from 10 to 11 & then assigned to b. Hence, b becomes 11.

Eg:-
```
int a = 10;
int b = a++;
cout << a << endl;
cout << b << endl;
```

OutPut = 11
10

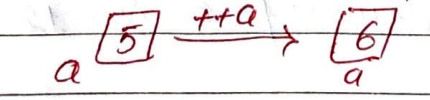
```
int a = 10;
int b = a++ (a = a+1)
    b = a = a+1
    b = 10 = 10+1
    b = 10 & a = 10+1 = 11
```

Here, we've Post-increment after the operand (a++), so the value of 'a' is first assigned to b. so b becomes 10 & 'a' is later incremented from 10 to 11.

Eg:-
```
int main () {
    int a = 5;
    cout << (++a);
}
```

OutPut = 6

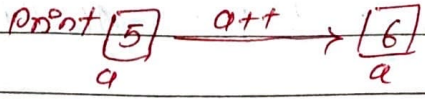
Here, In this type of case flow is most imp. Here, we're using Pre-Increment operater, so first value of 'a' is increase by 1 & then new value of 'a' is Print.




```
int main () {
    int a = 5;
    cout << (a++);
}
```

OutPut = 5

Here, in this example, we are using Post-increment operater, That means, first use the old value & then update the value. So first old value of a is Print (i.e. 5) & then value of 'a' is update to 6.



Date: .....
Page: .....

Eg:-

```
int main () {
    int a = 10;
    cout << (--a) * 10;
}
```

Output : 90

(--a) * 10 ;
first decrement the value of a. & then multiply by 10 (i.e. 9 * 10 = 90)

```
int main () {
    int a = 10;
    cout << (a--) * 10;
}
```

Output : 100

(a--) * 10 ;
first use the value of a & multiply by 10. Then decrement the value of a (i.e. 10 x 10 = 100)

→

```
int main () {
    int a = 21;
    cout << (++a) ;   // 22
    cout << (a++) ;   // 22
    cout << a ;       // 23
}
```

a → ~~21~~ ~~22~~ 23

first increment, then use
first use old value, then increment
Just print the new value of a.

→

```
int main () {
    int a = 10;
    cout << (++a) * 10;   // 110
    cout << (a++) * 10;   // 110
    cout << a ;           // 12
}
```

i) (++a) * 10 ; ⇒ first increament & then use. (i.e. a = 11)
11 x 10 = 110 ; now value of a is (11).

ii) (a++) * 10 ; ⇒ first use old value of a, then increment.
11 x 10 = 110 ; now value of a is (12) (Post-increment)

iii) a ; = 12 ; ⇒ At this time value of a is 12.

Date: .....

Page: .....

```
int main () {
    int a = 10;
    cout << ((++a) * (a++));
}
```

output = 121

++a = 11 (first increment)

a++ = 11 (first use old value)

(++a) * (a++)

= 11 * 11

= 121.

```
int main () {
    int a = 10;
    cout << ((a++) * (++a));
}
```

output = 120

a++ = 10 (first use old value & increment the value of a by 1.) i.e. a = 11

++a = 11+1 = 12 (first increment then use the value).

∴ (a++) = 10 & (++a) = 12.

(a++) * (++a) = 120.

```
int main () {
    int a = 10;
    int b = --a;
    cout << a; // 9
    cout << b; // 9
}
```

a = 10;

b = --a; first decrement & the use the value

b = a-1 = 10-1 = 9.

∴ a = 9 & b = 9

```
int main () {
    int a = 10;
    int b = a--;
    cout << a; // 9
    cout << b; // 10
}
```

a = 10;

b = a--; first use then change.

∴ b use the old value of a.

b = 10. After this process a is decrement by 1.

∴ a = a-1 ⇒ 10-1 = 9.

a = 9 & b = 10.