04/02/2022

* Bitwise Operator:

- AND → &
- OR → |
- NOT → ~
- XOR → ^

① AND

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

In AND operator i.e AND Bitwise operator both condition need to be true if one is false the resulting will be false.

Some ex:

①
```
~~int~~ bool a = true;
~~int~~ bool b = false;
cout << a&b;
```
output will be false as one of them is false.

②
```
int a = 2
int b = 3
```
first find bit equivalent for the integer.

a = 10 and b = 11

$$\begin{array}{r} 2 \rightarrow 10 \\ 3 \rightarrow 11 \\ \hline 10 = 2 \end{array}$$

output = 2 = 10 i.e bit equivalent

```
cout << a&b;
```

③
```
int main() {
  int a=5
  int b=6;
  cout << (a&b);
  return0.
}
```

output = 4.

Explain

Binary equivalent of 5 = 101

Binary equivalent of 6 = 110

$$\begin{array}{r} 5 = 101 \\ 6 = 110 \ \& \\ \hline 100 = 4 \end{array}$$

$\boxed{\text{Output} = 4}$

② OR operator

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

In OR if any of the condition is true than it will be true else false

ex: ①
```
int main()<
    bool a = true;
    bool b = false;
    cout<< (a|b);
```
output: True i.e 1
As any one of the condition is true

②
```
int main()<
    int a=5   |101
    int b=4   |100
    cout<< (a|b);
```
5→101
4→100
―――
101

output=5

③
```
int main ()<
    int a=5    |0101
    int b=~~10~~8|  1000
    cout<< (a|b);
```
5→0101
8→1000
―――
1101 =13

output=13

④
```
int main()<
    int a= -5
    int b= -4
```

*. How to find bitwise. AND(&) and OR(|) operator

ex :
```
int main()<
    int a= -4
    int b= -5
    cout<<( a&b);
```

output: a=-4 // ~~100~~ 100 1111........100 (2's comp)
b=-5 // 1111........1011

1000
1011
―――
1111....1000 =-8

```
int main()< 2's
    int a=-4 //1111....100
    int b=-5 //1111....1011
```

0100
1011
―――
1111=-1

output=-1

③ XOR Operator

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

In XOR operator if there are two same bit then value is 0 else, different value will be 1.

ex: ①
```
int main(){
    bool a= true;
    bool b= false;
    cout<< (a^b);
}
```
output = 1 i.e True
either one of the value is false

②
```
bool a = true;
bool b = true;
cout << (a^b);
```
output = 0 i.e False
~~either~~ both the value are True
same for both false

③
```
int main(){
    int a=14 = 1110
    int b = 10 = 1010
                 0100 = 4
```
output = 4.

④ NOT

a	not a= ~a
0	1
1	0

In NOT operator 0 becomes 1 and 1 becomes 0.

ex:
```
int main(){
    int a=5
    cout<< (~a);
    return 0;
}
```

output = - 6

5 =

0	0 - - - - - -	1	0	1

2's compl

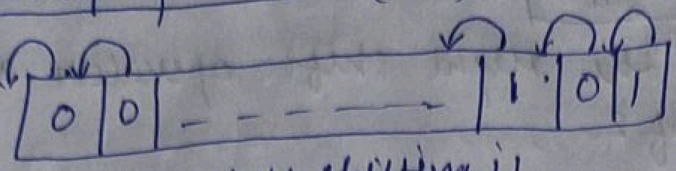
1	1	- - -	1	0	1	0

+1

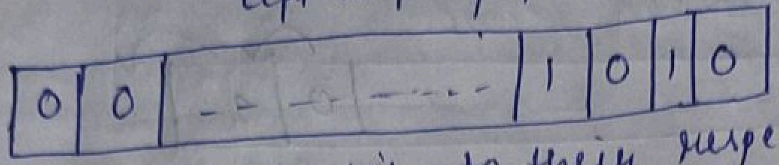
* Left and Right shift operator:

- Left shift operator: (<<)

ex:-



Left shifting it



= 10

Shifting each bit to their respective left side.

0.00 - - - - - - - 001 a=1

<< ↓×2

000 - - - - - 010 2 multiple of 2

<< ×2

000 - - - - - - - 0100 → 4

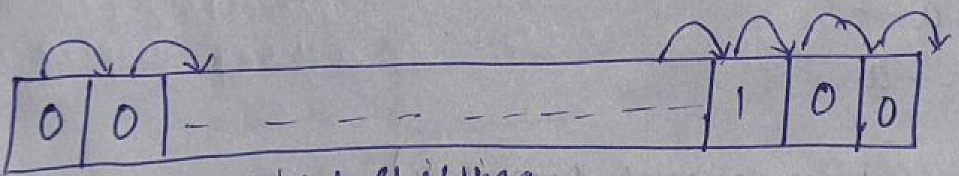
<< ×2

000 - - - - - 01000 → 8

The general formula to find the left shift operator is $\boxed{a \times 2^b}$

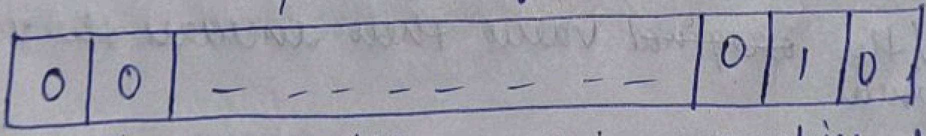
- Right shift operator (>>)

ex:



= 4

Right shifting.



= 2

Shifting each bit to their respective right side.

0 0 0 - - - - - - - 01000 → 8

>> /2

0 0 0 - - - - - - 0100 → 4

>> /2

0 0 0 - - - - - - 010 → 2

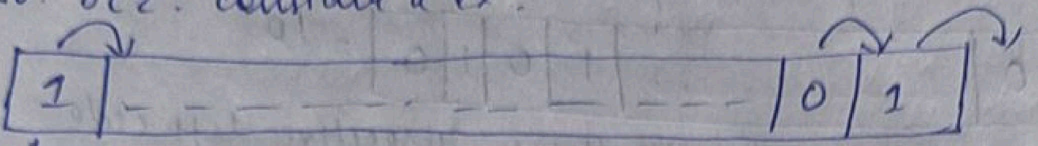
The general formula to find it is

$$\boxed{\frac{a}{2^b}}$$

Imp

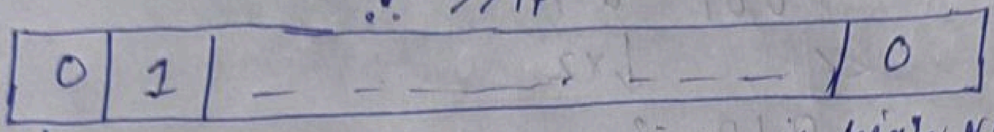
Q Can we say that the right shift operator are always divisor of 2.

→ No. bcz. consider a ex.



↓
MSB having MSB = 1 mean it is a -ve no.

∴ >>it



↓
MSB Now MSB = 0. which is high value +ve no. More the 1 is nearer to MSB more high no it is

as MSB = 0 i.e +ve No.

so ~~[illegible]~~ By dividing by 2 a -ve no doesn't convert into +ve high value no.

so always we can't say that they are exact divison of 2.

* Post and Pre Increment:

① a++ ③ a--

② ++a ④ --a

① a++ (a Post Incriament operator)

In this use/print the ~~original~~ value then increase the value by 1

```
ex:   int a=5
      cout << a++;   // 5
      cout << a;     // 6
```

a++ : will print the value the increase the value by 1 ~~Next~~ ∴ a++ = 5

a = 6

② ++a (Pre Increament operator)

In this first increase the value by 1 and then use it.

ex:
```
int a = 5;
cout<< ++a; //6
cout<< a;   //6
```

++a: will increase the value of a by 1 i.e a=6 then print the value i.e ++a=6

a: 6

③ a--(same as a++)

first use the value then decrease the value by1.

④ --a(same as ++a)

first ~~increase~~ decrease the value by 1 then use it.

* Examples:

①
```
int main(){
      int a = 6      → first use the value then increase.]
      int c = (a++) +1;
      cout<< c;         output = 6+1=7 //c
      cout<<a;                   7 //a
}
```

②
```
int main() {
   int a = 6  →    will decrease by 1 then print
   int c=(--a)+1;       c=5+1=6
   cout<<c <<a;         a=5
```

③
```
int main() {                    output = 49
   int a=5;                     Explain
   cout<<(++a)*(++a);
}
```

* Break and Continue Keyword ?

Break : break ~~statement~~ keyword is used to come out of the loop or stop the execution.

ex:

```
for( int i=0 ; i<n ; i++){
    cout<< ~~Bat~~ "Singh";<<endl;
    break;
}
cout<<" subrat"<<endl;
```

output n=5

i=0 i<5 Yes

Singh will be print

once reaching to break keyword execution stops & comes out of loop then print Subrat

Output

```
Singh
Subrat
```

Continue: continue keyword used to execute the ~~code~~ next iteration of the loop without doing the actual task. or going to next line.

ex:

```
for (int i=0;  i<5;  i++){
    continue;
    cout<<"subrat";
}
```

output

i=0 0<5

continue wont print subrat force to jump to next iteration

i.e i=1

same. i=2, i=3, i=4

so nothing display on screen.

continue is used when you want to skip any iteration.

* Variable Scoping:

Global variable: variable that can be used anywhere inside a program.

Local variable: variable whose scope belongs to the particular func. i.e can be used by that particular function

* The using of global variable is said to be bad practice because its value can be overwritten by any function.

ex:

```
int global = 6;
int main() {
    int a; // declaration
    int b = 6; // initialization
    b = 7; // updation
  X int b = 8; // Not possible
    if (true) {
        int b = 5;
        cout<<b;      ] → scope of this b is
                          inside if statement
                          can't be used outside.
    }
    cout << b;
    cout << global;
```

output

```
5
7
6
```

* switch case: alternative of if-else statement to increase the readability.

i.e

```
switch(expression) {
         value
    case 1: ________ break;
    case2: ________ break;
    case 3: ________ break;
     default: ________ ;
}
```

ex:

```
switch(value) {
   case 1: cout<<"Subrat"; break;
   case2: cout<< "supriya"; break;
   case 3: cout << "Singh"; break;
   default: cout <<"Not available";
}
```

we can't use continue statement with switch case because when particular case execute we come out of the switch case.

continue statements are basically used for loops without statement execution to the next iteration of loop.

As there is not possibility of ~~too~~ using continue in switch case.