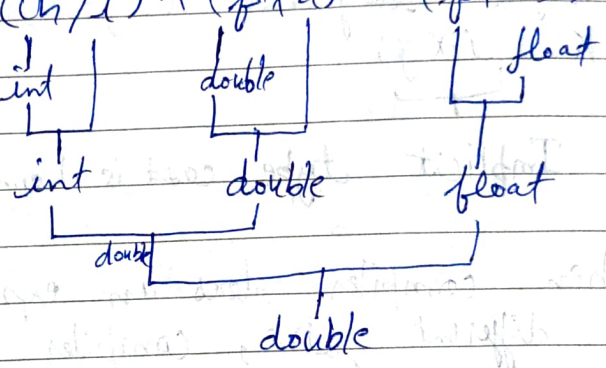
* Integral promotion:- In any exp., all char and short int values are elevated to int. This is called as integral promotion.

Type conversion Example:-

```
char ch;
int i;
float f;
double d;
result = (ch/i) + (f*d) - (f+i);
```



Example:-

```
unsigned int i = 2000000000, j=8;
unsigned long int k;
k = (unsigned long int)i*j; //Here j is implicitly
                            //type casted to 'j'.
```

- 'i' is explicitly type casted into long int.
- Here i and j are not physically changed is is converted or explicit type casted only for that expression

```
k = (unsigned long int)(i*j)
```

So type casting is not works here in grouping. Here the result is truncated this point only

Signed is treated as unsigned in type casting.

Session :- 11

→ Operators
  → Relational operators (>> = < <= == !=)
    Logical or Boolean operators (&& || !)

Relational operators :-

→ All the relational operators are binary operators
  → Required two operands to perform the task.
→ All the relational operators gives the result 0/1. Zero and One
→ All these operators will not modify the operands.

□	>	□	greater than
□	<	□	lesser than
□	>=	□	
□	<=	□	
□	==	□	
□	!=	□	

→ The operands can be variables/constants

Example :-

```
# include <stdio.h>
  Void main()
{
  int  i=10, j=20, k;
   k= i>j;   // 10>20 the result is FALSE so '0'
printf("i= %d  j= %d  k= %d \n", i, j, k);
}
```

O/P :- i= 10 j= 20 k= 0

- Arithmetic operator have more priority than the relational operators.

→
```
void main()
{
int i=10, j=20, k;
k= i>j-30;
printf("i=%d j=%d k=%d\n", i, j, k);
}
```

o/p:- i=10 j=20 k=1

→ i=10, j=20, k;
k= (i>j)-30; // (10>20)-30
// '0'-30 = -30

o/p k=-30

→ int i=10, j=30, k;
k= i==j;
Here == is higher priority
o/p K=0 // because i==j is false, so 0

→ Consider the same input
i==j=k
↳ for this statement, 'lvalue error' because i==j is constant. (i.e=0)

Examples:- Expressions:-

```
int i=10; j=20, k;
k = i>j;          // 0
k = i>j+20;       // 0
k = i>(j+20)      // 0
k = (i>j)+20      // 20
k = i==j;         // 0
k==i=10;          // lvalue req. error
```

k = i != j; // 1

→ Example:-

{
→ int i = -1, j = 1;
printf("%d\n", i > j); // 0 (-1 > 1) = 0
}

{
→ int i = -1; // signed i > unsigned j
unsigned int j = 1; // compiler will treat this signed as unsigned
printf("%d\n", i > j);
}

Note:- Type promotion :- In any expression, if one operand is signed and another operand is unsigned, signed is treated as unsigned.

signed i > unsigned j // compiler treated signed i as unsigned i

4u > i

because, in signed the binary value of -1 is

1 | 1111111 | 11111111 | 11111111 | 11111111 = i

sign bit Data bits

But compiler consider this as unsigned so all are data bits

- i.e value of i is 4u

⇒ 4u > 1 is checked

Logical operators :-

&& Logical AND
|| Logical OR
! Logical NOT

- &&, || are binary operators (2 operands)
- ! is unary operator
- All these logical operators will not modify the operands
  i.e we can supply a constant or variable.

→ All these logical operators will give us the result '0' or '1'.

a	b	AND a&&b	OR a||b
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

0 (zero), 1 (means non zero)

Ex:-
```
void main()
{
  int i=10, j=20, k;
  k= i && j;
printf("i=%d j=%d k=%d", i, j, k);
}
```

→ If the 1st operand is '0' the compiler will not check the second operand for Logical AND. It will decide the result zero.

i=10, j=20

K= i && (j=200)
= 0 && (exp)
⇒ K=0

But if i=10; j=20
K = i && (j=200);
= 10 && 200 exp
= 10 && 200 K=1

o/p i=10, j=20, K=1;

→
```
void main()
{
int i=10;
!i;
printf("i= %d\n", i);
}
```
Here the logical NOT will not change the operand

Consider:-

→
```
int i = 10;
i = !i;
printf("i= %d\n", i);
```

o/p = i=0

Complimend and NOT are different
↓ ↓
Bitwise (~) !

→ Logical AND has the higher priority than Logical OR

→ Priority
(i) NOT (ii) Logical AND
(iii) Logical OR

Session: 12

Operators :-

→ Logical (or) Boolean operators (&& || !)

→ Bitwise operators (& | ^ ~ << >>)

Consider,

```
m = i+j-k*10;
```

From a user's view,

```
m = [i+j] - [k*10]
    2       1
```

from a compiler's view

Depending upon the priority, the compiler will group it implicitly

```
m = i+j-(k*10);
m = i+j - exp.
m = exp - exp;
m = exp2
```

when a bigger exp. to the compiler, it will compress the expression to the smallest, depending upon the priority, until it becomes one operand. Then compiler start solving it.

→ Expressions:-

- `int i=10, j=20, k; //o/p k= 0`
  `k= i && (j=200)`
- `i=0, j=20, k=30, l; // o/p l= 1`
  `l=i && (j=200) || (k=300);`
- `l = i || (j=200) && (k=300) || 50;`
- `l = i && (j=200) || (k=300) && 50 || 0;`
  o/p i=10, j=200, k=30, l=1

Bitwise operators :-

& → Bitwise AND
| → Bitwise OR
^ → Bitwise XOR
~ → Bitwise NOT (complement)
>> → shift right
<< → shift left

p	q	p&q	p|q	p^q	~p
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

→ Except bitwise compliment, remaining operators are binary.
— All these operators will not modify the operands, so a constant or variable can be supplied as the operands.
— These bitwise operators are not applicable to real numbers. (float and double).
— These operands should be converted into binary to devide the result.

Example :-

```
→ void main()
  {
    int i=10, j=15, k;
    k=i&j;   // i & j are converted into binary
             bitwise & is used.
  printf("k=%d", k);
  }
```

Explanation :- i = 10 i.e 0000 1010 → 10
j = 15 i.e 0000 1111 → 15
& 0000 1010 →

result of bitwise AND 10 & 15 = 10
K = 10

→ If K |= i | j;

```
0000 0101  → 10
0000 1111  → 15
0000 1111
```

K = 15

→ K = i ^ j;

```
0000 1010  → 10
0000 1111  → 15
0000 0101  → 5
```

K = 10 ^ 15 = 5

→ K = ~i; // as the given data type is int (4 bytes = 32 bits), for its compliment entire 32 bits have to be considered

```
00000000 00000000 00000000 0000 1010 → 10
11111111 11111111 11111111 1111 0101 → 1's
```

2's compliment +1

After taking compliment
Here sign bit is 1 → so it is a -ve no.

→ To decide which -ve no is this again 2's compliment.

```
11111111 11111111 11111111 1111 0101
00000000 00000000 00000000 0000 1010
                                +1
                           1011
                           8421 =
                           -11
```

K = -11

→ Both ! and ~ are unary operators Here NOT operator inverts the zero/Non zero directly

Where as
Compliment (~) operator inverts the value of the operand in bitwise.

→ int i=10, j;
j = ! i
j = 0

j = ~ i
j = -11

Examples:-

→ int i=100, j=150, k;
- k = i & j; // k = 4
- k = i | j; // k = 246
- k = i ^ j; // k = 242
k = ~ i; // k = -101

/* By converting i & j into binary */

Session 13:- 30-6-20

→ Operators
→ Bitwise operators (& | ^ ~ << >>)
→ Swapping of 2 numbers using bitwise operators.
→ How to set a bit in a given no.
→ How to clear a bit in a given no.

→ Swapping of two no. using bitwise operator:-

{
a = a ^ b; // a=5
b = a ^ b; // b=10
a = a ^ b; //
}

a=10, b=15

0000 1010 → 10
0000 1111 → 15
a= 0000 0101 → 5 XOR → 5
b= 0000 1111 → 15
b= 0000 1010 → 10 XOR
a= 0000 0101 → 5 XOR
b= 0000 1111

before swap
a=10, b=15
After swap
a=15, b=10.

→ Shift operators:-

→ (<<) left shift
→ (>>) Right shift.

- These are binary operators
- These operators will not modify the operands
- The operand can be constant/variable.

Ex: Left shift

```
int i=5, j=2, k;
k=i<< j;
```

Here, it means i is left shifted by j times

```
    i  <<  j
    ↓      ↓
the operand   no. of times to
to be shifted be shifted
```

[1st operand is shifted by 2nd operand times]

Note:- In shift operator (both left and right) the 1st operand is converted into binary (2nd operand can not be converted into binary

Ex:- int i=5, j=2, k;
k= i << j;

i = 5

0 0 0 0 0 1 0 1

2 times shift so

This zero is out when shift

0 0 0 1 0 1 0 0 = 20

K = 20 Ans

Ex:- int i=5, j=1, k
k = i<<j;

0000 0101 → 5
0000 1010 → 10

Rule → Result = Number * 2^{shift}
Left shift.

* → Note:- In shift operators, the 2nd operand should not be negative number. The no. should be 0-31 while dealing with int datatype.
→ But if give -ve numbe no error will come but unexpected value comes.
→ and even if the 2nd operand exceeds the limit (eg. 0 to 31) the result is unspecified.

→ The 1st operand may be -ve or +ve no.

Ex:-
int i=-10, j=1, k;
k = i<<j;
k = -10<<1;
Now take binary of +10

00000000 00000000 00000000 00001010
Taking 2's compliment for -10
11111111 11111111 11111111 11110101
+1
11111111 11111111 11111111 11110110
This no will left shifted 1
is thrown out
11111111 11111111 11111111 11101100
Left shifted

To find what value it is again 2's compliment

0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 | 0 0 11
+ 1
0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 | 0 0 100

⇒ k = -20

→ Right shift operator (>>)

i = 10, j = 1, k
k = i >> j; // 10 is right shifted by 1 times
i = 10 0 0 0 0 | 0 | 0 ↷ thrown out (gone of of memory)
New zero added → 0 0 0 0 0 | 0 1
k = 5

Rule:- Right shift

Number / 2^shift

int i = -1, j = k;
k = i >> j;

→ -ve no. given so sign bit copy mechanism used.

Note:- Sign bit copy mechanism used here.

1 1 1 1 1 1 1 1 | 1 1 1 1 1 1 1 1 | 1 1 1 1 1 1 1 1 | 1 1 1 1 1 1 1 1 ↷ thrown out
1 1 1 1 1 1 1 1 | 1 1 1 1 1 1 1 1 | 1 1 1 1 1 1 1 1 | 1 1 1 1 1 1 1 1
Sign bit copied again after shift.
This is called sign bit copy mechanism

Note:- In right shift case, though the sign bit is shifted, it is copied to that shifted bit position.

$[K = -1]$

Here generally
int i=0, j=1, k;
i = i-1;
i = -1

unsigned int i=0, j=1, k;
i = i-1; i = 0-i;
$\Rightarrow [i = 4G]$
printf("%d", i);
0 $\longleftrightarrow$ 4G
after 0, it again goes to 4G, so the value of i = 4G
$\Rightarrow$ all 32 bits are '1'
All bits are data bits so

after shift
0 | 1111111 | 1111111 | 1111111 | 1111111 |
New zero added

Here the value is 2G

The op is
i = -1, j = 1 K = 2147483391
↓
-1 because %d is given to take the o/p

→ But in signed data type sign bit is copied again in case of -ve no. given.

Session 14

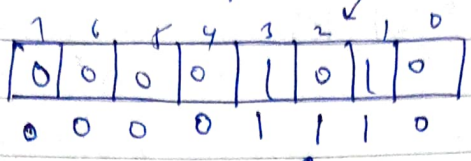
How to set a number using bitwise operator.

We need to take 2 i/ps from the user
① Number ② position

1 means set
0 means clear

Set = `num = num | 1 << pos;`

num = 10 pos = 2



Ans = 4

<< left shift have higher priority

→ num = num | 1 << pos;
num = num | Result
num = Result

Note:- Position should not be -ve number.

→ Now, num = +10, pos = 2
10 binary → 0000 1010
As per expression num = num | 1 << pos
num = 10 | 1 << 2
take 1 << 2 as it has higher priority
0000 0100
Now num = 10 | result
0000 1010
0000 0100 OR
0000 1110 14 Ans.

→ If Num = 15, pos = 2
Nothing change already set

0000 1111
0000 0100 OR
0000 1111 15 ✓

→ How to clear a bit using bitwise operator.

Clear:- `num = num & ~(1 << pos)`

num = num & ~(1 << Pos)

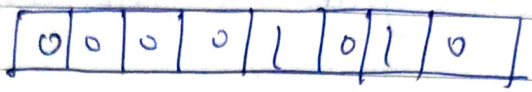
① ↓

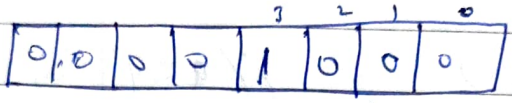
num = num & ~result

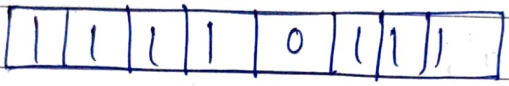
num = num & result

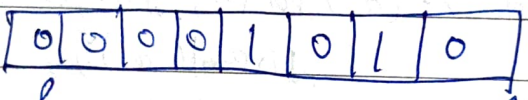
num = result.

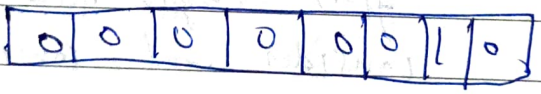
Consider:- num = 10 , pos = 3









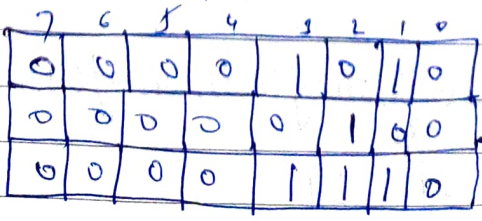


→ Complimenting the bits (Toggle)

This formula toggle 1 bit

Toggle - `num = num ^ 1 << pos;`

num = 10 , pos = 2;



Set → num = num | 1<<pos;

Clear → num = num & ~(1<<pos);

Compliment → num = num ^ 1<<pos;

→ Sizeof() operator:-

→ It is a unary operator sizeof()
⇒ The operand can be constant/variable/datatype
→ This operator gives the size in the form of bytes.

Ex:- void main() // a=97 in ASCII
{
char ch = 'a';
printf("%d %d %d %d\n", sizeof(char), sizeof(ch), sizeof('a'));
}

o/p:- 1 1 4 → The sizeof() character constant gives 4 bytes because compiler consider int constant.

→ Size of 'a' constant is 4 bytes the compiler will treat 97 as an integer.

Ex:- void main()
{
int i = 10;
printf("%d %d %d\n", sizeof(int), sizeof(i), sizeof(10));
}

o/p:- 4 4 4

→ Example:- void main()
{
float f=23.5;
printf("%d %d %d\n", sizeof(f), sizeof(4), sizeof(23.5));
}
o/p:- 4 4 8 // 23.5 compiler treats as double.

void main()
{
printf("%d\n", sizeof("abcd"));
}
o/p: 5 // string (" ")

Expressions should not be passed in a sizeof() operator.
If written, the expression is not going to be solved.
{ int i=10, j=20, k;
printf("k=%d\n", k);
sizeof(k=i+j);
printf("k=%d\n", k);
}
o/p:- k=0
k=0

Session 15:-
→ Conditional operators (? :) / Ternary
→ How to check in a given number given bit position set or or clear.
→ Introduction to IEEE 754 real number representation

→ Ternary / Conditional Operator :-

- It required 3 operands to perform the task
- The symbol for conditional operator is (__ ? __ : __)
- Syntax :- exp1 ? exp2 : exp3
- Compiler will solved the first expression
- Depends upon the first expression result may be the 2nd operand or 3rd operand
- If exp1 is non zero then result becomes exp2, if exp1 is zero the result becomes exp3.

Example:-

```
void main ()
{
int i=10, j;
j= i ?100 :200;
printf("i=%d j=%d\n", i, j);
}
```

// o/p:-
i=10 j=100
j=100 because the 1st operand is non zero.

→

```
void main ()
{ int i=0, j;
j= i ?100 :200;
printf("%d\n", j);
}
```

o/p:- j= 200
Here j=200 because 1st operand is zero

Here 1st operand is variable
We can take variable, expression, constant etc.

Ex:- K= i?j:100 :200;
printf("%d", K);

// i=10, j=20
i<i?j: False so
o/p:- K=200.

- Write a program to check whether the given no. is positive or negative.

```
int num;
≡
num>=0 ?printf("+ve \n"):printf("-ve");
```

- Odd / even program :-

```
int num;
≡
① num%2==0?printf("Even\n"):printf("odd\n");
② num%2?printf("odd\n"):printf("Even");
```

- Write a program to find the biggest of two no. :-

```
- int a,b;
  ≡
  a>b?printf("a is greater\n"):printf("b is greatest");
```

Note:- The conditional operator can be nested.

- Find the biggest of 3 numbers :-

```
int a,b,big;
big=a>b?(a>c?a:c):(b>c?b:c);
```

→ If the 3rd operand is not grouped the compiler thrown 'l value required error.

(i>=10) ? j=20 : (j=30) ← correct

- If not grouped compiler consider j as 3rd operand.

- How to check in a given number given bit is set or clear?

```
r = num & 1<<pos;
```

result

result.

```
r = num >> pos & 1;
```

result & 1

result

num	pos	r = num & 1<<pos (zero & non zero)	r = num >> pos & 1; (zero and 1)
15	0	1	1
15	1	2	1
15	2	4	1
15	3	8	1
15	4	0	

```
int num, pos;
Sprintf - -
sleep - —                    → result 0/1
num >> pos & 1 ? printf("set\n") : printf("clear");
//num & 1<<pos ? printf("set\n") : printf("clear");
                y result zero/non zero
```

o/p:- Enter the number
15
Enter the pos
3
Set.

Session :- 16

→ Operators
→ IEEE754 real number representation
→ Inc and Dec operator.

→ Size of float and double is not decided by the OS, it depends on the standard IEEE754.

- Size of the float is 4 bytes.
- Size of the double is 8 bytes

→ unsigned/signed float/double can't be declared. By default both are signed

→ There's no direct way to operate on the float by the mc/mp.
So a floating point unit [FPU] is used.
This is a co-processor.
[There are no assembly instructions to deal with the floating points].

- Few processors have inbuilt FPU
- Most of the FPU's internally follows the standard IEEE754.

— Eg:- void main()
{
float f = 23.5; ← Here double value is given
printf("%f %e %g\n", f, f, f);
}
// 23.5 double converted into // float by compiler internally.

→ %e prints the data in standard decimal exponent format.

O/P:- 23.750000 2.375000e+01 23.75

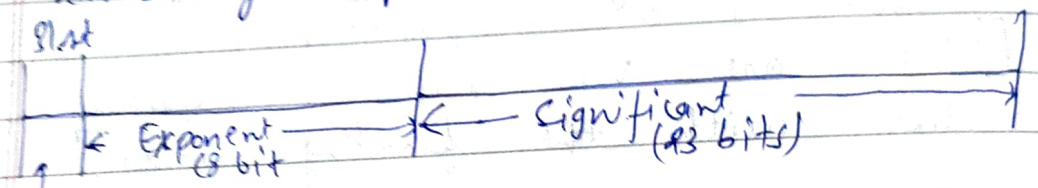
∴ When %f is used, it prints 6 digits after standard exponent format.
Standard exponent format → digit.xe ±y

- Double is treated as long float [%lf]

Representation in IEEE754 Standard :-

		Sign	exponent	significant (mantissa)
float	32 bits	1	8	23
double	64 bits	1	11	52

Here when float is declared for
float f = 1 and float f = -1
only the sign bit is changed (varies)
remaining exponent & significant are same.



→ Step to get IEEE754 format :- (Real number)

① Convert the float into normal binary format.

e.g.: f = 23.5

Ist 23 convert and then .5 convert

10111.10

2	7	
2	3	-1
	1	-1

11 = 7

Ex: 7.7 binary

.7 = 111

Now 0.7 X 2 = 1.4
0.4 X 2 = 0.8
0.8 X 2 = 1.6
0.6 X 2 = 1.2
0.2 X 2 = 0.4
0.4 X 2 = 0.8 → Here repeating from 0.8

$7.7 \rightarrow 111.\overline{0110}$ (bar is given over the nos because they are repeated)

(2) Convert the binary into standard binary exponent format.

$$\boxed{\text{Bit} \cdot x^{e \pm y}}$$ where $x \rightarrow$ significant, $y \rightarrow$ exponent

Take float of 23.5;

$\Rightarrow 10111.10 \rightarrow 1.011110e^{+4}$ (+4) exponent

$\rightarrow$ Significand / mätissa

(3) Add or substract exponand part with standard bias number (IEEE number)

For float 7F for double 3FF

(7F → 127) (3FF → 1023)

↓ Hexa binary.

Here, addition or substraction depends on the exponential sign

Now, $1.011110e^{+4}$

Bias +4 → 7F + 4 = 83 (Hex)

7F → 127 | 0
4
131
indecimal 0

(4) Exponent part 8 bits:

so 83H → 1000 0011
8 3

(5) Significant bits are 23 Here $1.011110e^{+4}$
(Fill the significant part from left to Right)

(6) Sign bit → 0 (bec. the no. +ve)

0 | 1000 0011 | 01100 00000000 00000000
(1) (8) (23)
Sign exponent significant

→ Example:- Consider
float = -7.7;

① Converting -7.7 into binary
→ 7.7_{10} = 111.1011̄0

② Standard binary format
111.1011̄0 ⇒ 1.1110110 e^{+2}

③ 7F +2 ⇒ 81 (Hexa)

④ 81 into binary
1000 0001 (8bits
8 1

⑤ Significant part (23 part)
1110 1100 1100 1100 1100 1101

⑥ Sign bit (1) given no -ve
float f = -7.7.

1	10000001	11101100110011001100110
Sign	exponent	significant

→ Conversion of double into IEEE754 format

① double d = 23.5;
① Convert the no. into its binary
⇒ 10111.10

② Convert the binary format into standard binary exponent format
1.011110 e^{+4}

③ Bias +exponent
⇒ 3FF +4 = 403
(Hexa)

④ Convert the exponent part into 11 bits
100 0000 0011
4 0 3

⑤ Significant part (52 bits)