Unconditional Controlled Statement-:

1. break-:

The 'break' is an unconditional controlled statement. When the compiler read the 'break' statement, it will come out the loop or switch.

'break' must be there inside the loop or switch. Only combination of 'if' and 'break' leads to the error.

Whenever the program result has been specified then we can use the 'break' statement and we can come out manually by using 'break'.

Syntax-

```
main()
{
  ----
  ----
  while(condition)
  {
    ----
    ----
    if(condition)
      break;
    ----
    ----
  }
  ----
  ----
}
```

T

T

// end of loop

// end of main()

Ex.

```
# include <stdio.h>
# include <conio.h>
 void main ()
{

int num, i=1, sum=0;
clrscr ();
while (i<=10)
{
printf("Enter the value of %d :",i);
scanf("%d", &num);

if (num<=0)

break;
sum += num;
 i++;
}

printf("count : %d sum : %d", i,
                          sum);

getch ();
}
```

o/p →

```
Enter the value of 1 : 56
Enter the value of 2: 85
Enter the value of 3! -87
count : 3 sum ! 141
```

2. continue -:

The 'continue' is exactly opposite to the 'break' statement. When the compiler read the 'continue' statement, it once again re-enter inside the loop.

'Continue' must be there inside the loop only.

If you want to skip specific situation in a condition, use the 'continue' statement. When we using continue statement iteration part has to present.

syntax -

```
main ()
{
------
------
while (condition)
{
if (condition)
continue;
------
------
}                    // end of loop
------
------
}                    // end of program
```

Ex. Write a program to findout biggest number without accepting negative values.

```
main()
{
int num, i = 1, big = 0;
clrscr();
while (i <= 10)
{
printf("Enter the number %d :", i);
scanf("%d", &num);

if (num < 0)
{
printf("Negative nos are not
                allowed \n");
getch();
continue;
}

if (num > big)
big = num;
i++;
}
printf("Biggest : %d", big);
getch();
}
```

Ex.

```
main()
{
int i = 1;
clrscr();
while (i <= 10)
{
if (i > 4)
break;
printf("%d\n", i);
i++;
}

printf("%d", i+10);
getch();
}
```

O/p → 1 2 3 4 15

Ex.

```
main()
{
int i = 0;
clrscr();
while (i < 100)
{
i = i+2;
if (i > 10 && i < 60)
continue;
printf("%d", i);
}
```

```
getch();
}
```

o/p →

2 4 6 8 ------- 38, 40, 60, 62, ---
--- 96, 98, 100.

Ex

```
main()
{
int i=1;
clrscr();
while(i<=20)
{
if(i>4 && i<=17)
continue;
printf("%d", i);
i++;
}
getch();
}
```

o/p → 1 2 3 4

3. exit(0) :-

When the compiler read the exit() statement, it will come out of the program. When exit() statement is utilizing use a parameter of 0, indication to the Operating System the program is successful. The exit() can be used any where in the program either in looping or decision making statement.

Mostly exit() will be used in the decision making statements.

Syntax -

```
main()                    → O.S.
{
- ---
- ---
if(condition)
{
- ---                     → Program Successful,
exit(0);                    (indication for O.S.)
}
- ---
- ---
- ---
}                         // end of the program
```

out of the program

Ex. Write a program to accept a no. and print its factorial.

```
main ()
{
int num;
long f = 1;
clrscr ();
printf (" Enter the number : ");
scanf (" %d ", & num );

if ( num == 0 || num == 1 )
{
printf (" Factorial is 1 ");
getch ();

exit (0);
}

while (num > 1)
f = f * num --;
printf (" Factorial is %ld ", f);
getch ();
}
```

O/p →

Enter the number: 5
Factorial is 120 .

* Fibonacci Series -

In this series first two values are 0 and 1. and remaing values can be find out by addition of last two values.

Fabonacci Series is given below -

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ---

* Perfect Number -:

Beginning with the number 1, keep adding power of 2 (doubling the numbers) untill you get a sum which is a prime number.

A perfect number is obtained by multiplying the this sum to last power of 2.

Ex.

Sum		Prime	Multiply	Perfect
1 + 2	= 3	Yes	3 * 2	6
1 + 2 + 4	= 7	Yes	7 * 4	28
1 + 2 + 4 + 8	= 15	No	- - - - -	- - -
1 + 2 + 4 + 8 + 16	= 31	Yes	31 * 16	496

Ex. Write a program to accept a number. Find out the given no. is in fibonacci series present or not.

```
main ( )
{
int t, t1 = 0, t2 = 1, num, ck = 0;
clrscr ( );
printf ("Enter the number : ");
scanf ("%d", &num);
if ( num == 0)
 ck = 1;
else
{
while ((t = t1 + t2) <= num)
{
printf ("%d", t);
 if ( t == num)
{
 ck = 1;
 break;
}
t1 = t2;
t2 = t;
}
}
if (ck == 1)
printf ("\n Given no. is present in
                fibonacii series");
else
printf ("\n Given no. is not present
            in fibonacci series");
getch ( );
}
```

o/p →

```
Enter the number : 27
1 2 3 5 8 13 21
```

Given no. is not present in fibonacii series

Ex. Write a program to print all the perfect nos.

```
main ( )
{
int numb, fact, frum;
clrscr ( );
```

```
numb = 1;
while (numb < 1000)
{
frsum = 0;
fact = 1;
while (numb < 10000)
{

frsum = 0;

while(fact <= numb/2)
{
 if (numb % fact == 0)

frsum = frsum + fact;
 fact ++;
}
if ( frsum == numb)
printf ("\n%4d is a Perfect no.\n",
                          numb);

numb ++;
}
getch ();
}
```

o/p →

6 is a Perfect no.
28 is a Perfect no.
496 is a Perfect no.

Ex: Write a program to compute the sum of series.

$1 + x + x^2 + x^3 + x^4 + \cdots + x^n$.

```
void main ()
{
 int x, count, term, sum, i = 1;
clrscr ();
printf ("\n Enter the value of x:");
scanf ("%d", &x);

printf("\n upto How many nos. sum
   is needed :");
scanf ("%d", &count);
 term = sum = 1;
while (i < count)
{
 term = term * x;
sum = sum + term;
  i++;
}
printf("\n The sum upto %d terms
      of series", count);
printf ("\n 1 + %d + %d**2 + ---
      --- is %d ", x, x, sum);
getch ();
}
```

o/p →

Enter the value of x : 5
upto how many nos sum is needed : 3

The sum upto 3 terms of series
1 + 5 + 5 **2 + ---- is 31.

Ex. Write a program to compute the sum of series

$1 + 1/x + 1/x^2 + 1/x^3 + \cdots + 1/x^n$

```
#include <stdio.h>
#include <conio.h>
  void main ()
{
 int x, count, term, i = 1;
 float sum, y;
 clrscr ();
printf ("\n Enter the value of x : ");
scanf ("%d", &x);

printf ("\n Upto how many nos
          sum is needed : ");
scanf ("%d", & count );

term = sum = 1;
while ( i< count )
{
 term = term * x;

  y = 1.0 / term ;
sum = sum + y;
i++;
}

printf ("\n The sum upto %d term
        of series ", count );
printf ("\n 1 + 1/%d + 1/(%d **2) + --
        ---- is %f \n", x, x, sum);

getch ();
}
```

o/p →

Enter the value of x : 4

Upto how many nos sum is needed : 3

The sum upto 3 term of series

1 + 1/4 + 1/(4 **2) + ---- is 1.312500

Ex. Write a program to accept two nos as a power and base. Find out the power value.

```
void main ()
{
int a, b, i = 1;
long result = 1;
clrscr ();
printf ("\n Enter the base value: ");
scanf ("%d", &a);
printf ("\n Enter the power value: ");
scanf ("%d", &b);
while (i <= b)
{
result = result * a;
i++;
}
printf "\n Result is %ld", result);
getch ();
}
```

o/p →

```
Enter the base value : 2
Enter the power value: 5
Result is 32
```

Ex. Write a program to accept a number. Find out the given number's factors. and also find out its a prime number or composit number.

```
void main ()
{
int num, count = 0, i = 1, a;
clrscr ();
printf ("\n Enter the number: ");
scanf ("%d", &num);
while (i <= num)
{
if (num % i == 0)
{
printf (" %d", i);
count ++;
}
i++;
printf (" are factors of %d", num);
if (count <= 2)
```

```
printf ("\n Prime number ");
else
printf ("\n Composite number");
getch ();
}
```

o/p →
Enter the number : 15
1 3 5 15 are factors of 15
composite number.

Ex. Write a program to accept a decimal no. and print its binary.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
void main()
{
 int n, p = 0;
long unsigned bia = 0;
clrscr();
printf("Enter any decimal number:");
scanf("%d", &n);
while (n > 0)
{
bia = bia + ((n % 2) * pow(10, p));
n = n/2;
p++;
}
printf("Binary no. of given no.: %lu",
                              bia);
getch();
}
```

o/p →
Enter any decimal number : 14
Binary no. of given no : 1110

Ex. Write a program to accept a number and also accept two digits. Replace the first digit with second digit and display the number.

```
#include <stdio.h>
#include <conio.h>
 void main()
{

long int n, r = 0, dummy;
int d1, d2;
clrscr();
printf("Enter any number:");
scanf("%li", &n);

printf("Enter first digit:");
scanf("%d", &d1);
```

```
printf("Enter second digit : ");
scanf("%d", &d2);

while (n>0)
{

r = r*10 + (n%10);
 n = n/10;
}
 n=0;
while (r>0)
{
 dummy = r%10;

if(dummy==d1)
dummy = d2;
n = n*10 + dummy;
r = r/10;
}
printf("%ld", n);
getch();
}
```

o/p →

```
Enter any number : 345373
Enter first digit : 3
Enter second digit: 1

145171
```

for -:

'for' is an entry controlled looping statements. The 'for' work flow will takes place in anti-clock wise direction.

Syntax -

```
for(initialization; condition; updation)
{
   statement 1;
   statement 2;
   ----------
   ----------
}
```

In 'for' loop everything is optional. i.e. we can write it like this -

```
for ( ; ; )          // valid
```

but

```
while ( )            // invalid
```

'for' is more flexible rather than 'while'.

In 'for', there can be any no. of initialization, any no. of condition and any no. of updation.

as like -

```
for(i=0,j=0; i<=5,j<=10; i++,j--)
```

no. of initialization and condition can be seperate with the comma (,) operator. Between the condition we can use relational and logical operators also.

Nested for -:

Nested for is also supported by 'c' language. i.e. there can be a 'for' loop inside a 'for' loop.

~~First~~

Ex. Write a program to print nos. from 10 to 1 with the help of for loop.

First method -:

```
main()
{
 int i;
for(i=10; i>=1; i--)
 {
  printf("%3d", i);
 }
```

```
getch();
 }
```

o/p →

10 9 8 7 6 5 4 3 2 1

Second Method -:

Ex.

```
main()
{
 int i;

for(i=10; i>=1; printf("%3d", i--));

getch();
 }
```

o/p →

10 9 8 7 6 5 4 3 2 1

in the above program compiler will do some replacement as -

```
for(i=10; i>=1; printf("%3d", i--))
 {

 }
```

Ex.

```
void main()
{
int a,b;
clrscr();
for(a=b=1; a; printf("\n%d %d",a,b))
a = b++ <=3;
printf("\n%d %d", a+10, b+10);
getch();
}
```

o/p →

```
1   2
1   3
1   4
0   5
10  15
```

Replacement —

```
for(a=b=1; a; printf("%d %d",a,b))
{
a = b++ <=3;
}
```

Ex.

```
void main()
{
int x=0, j, k;
clrscr();
for(k=0, j=5; k<5, j>0; k=k+2, j--)
{ x++;
printf("\n the value of k,j,x is
        %d %d %d", k, j, x);
}
getch();
}
```

o/p →

```
the value of k,j,x is  0  5  1
the value of k,j,x is  2  4  2
the value of k,j,x is  4  3  3
the value of k,j,x is  6  2  4
the value of k,j,x is  8  1  5
```

Ex.

```
void main()
{
int x=0, j, k;
clrscr();
```

```
for(k=0, j=5; k<5 || j>0; k=k+2, j--)
{
  x++;
printf("\n the value of k,j,x is
        %d %d %d", k, j, x);
}
getch();
}
```

o/p →

```
the value of k,j,x is  0 5 1
the value of k,j,x is  2 4 2
the value of k,j,x is  4 3 3
the value of k,j,x is  6 2 4
the value of k,j,x is  8 1 5
```

Ex.

```
void main()
{
  int x=0, j, k;
clrscr();
for(k=0, j=5; k<5 && j>0; k=k+2, j--)
{
  x++;
printf("\n the value of k,j,x is
        %d %d %d", k, j, x);
}
getch();
}
```

o/p →

```
the value of k,j,x is  0 5 1
the value of k,j,x is  2 4 2
the value of k,j,x is  4 3 3
```

Ex.

```
void main()
{
  int j=0;
clrscr();
for(;;)
{
printf("%d", j);
  j++;
if(j>5)
break;
}
getch();
}
```

o/p → 0 1 2 3 4 5

Ex.

```
void main()
{
  int a=1;
clrscr();

for(a++; a++<=2; a++)
```

```
for(a++ ; a++ <= 7 ; a++)

 a++;
printf("%d", a);

getch();
 }
```

o/p → 1 3

Replacement –

```
for (a++ ; a++ <= 2 ; a++)
{

for (a++ ; a++ <= 7 ; a++)
  {
    a++;
  }
}
printf("%d", a);
```

Write a program to print

```
0 0   0 1   0 2
1 0   1 1   1 2
2 0   2 1   2 2
```

```
void main()
{
 int i, j;
clrscr();

for( i = 0 ; i <= 2 ; i++)
{
 for (j = 0 ; j < 3 ; j++)

printf("%3d %d", i, j);
printf ("\n");
}

getch();
 }
```

Ex. Write a program to print

```
1   1   1   1
2   2   2   2
3   3   3   3
4   4   4   4
```

```
void main()
{
 int i, j;
clrscr();
for (i = 1 ; i <= 4 ; i++)
{
 for (j = 1 ; j <= 4 ; j++)
```

```
printf ("%3d", i);
printf ("\n");
}

getch ();
}
```

Ex. Write a program to print

```
1   1   1   1
2   2   2   2
3   3   3   3
4   4   4   4
```

don't use 'for' loop. Use only 'while' loop.

```
void main ()
{
 int i=1, j;
clrscr ();
while (i<=4)
{
 j=1;

while (j<=4)
{
 printf("%3d", i);
  j++;
}
```

```
printf ("\n");

i++;
}
getch ();
}
```

Ex. Write a program to print

```
1
1 2
1 2 3
1 2 3 4
```

```
main ()
{
 int i, j;
clrscr ();
for (i=1; i<=4; j++)
{
 for (j=1; j<=i; j++)

printf("%d", j);
printf ("\n");
}

getch ();
}
```

Some program can be write with the help of 'while' as-

```
main()
{
int i=1, j;
clrscr();

while(i<=4)
{
 j=1;
while(j<=i)
{
printf("%d", j);
 j++;
}

printf("\n");
 i++;
}
getch();
}
```

Ex: Write a program to print

```
   1
  1 2
 1 2 3
1 2 3 4
```

```
#include<stdio.h>
#include<conio.h>

 void main()
{
 int i, j, n;
clrscr();

printf("Enter the N:");
scanf("%d", &n);

for(i=1; i<=n; i++)
{
  j=n;

while (j>i)
{
printf(" ");
 j--;
}

for (j=1; j<=i; j++)

printf("%d", j);
printf("\n");
 }
getch();
 }
```

Ex Write a program to print -

```
        1
      1 2 1
    1 2 3 2 1
  1 2 3 4 3 2 1
1 2 3 4 5 4 3 2 1
```

```
void main()
{
 int i, j, n;

clrscr();
printf("Enter the N: ");

scanf("%d", &n);

for(i=1; i<=n; i++)
 {
   j= n;

while(j>i)
{
 printf(" ");
   j--;
 }

for(j=1; j<=i; j++)
 printf("%d", j);

 j= i;
while (j>1)
{

printf("%d", j-1);
  j--;
}

printf("\n");
 }
 getch();
}
```

Ex. Write a program to print

```
    1
  2 2
3 3 3
4 4 4 4
```

```
void main()
{
 int i, j, n;
clrscr();
printf("Enter the N: ");
scanf("%d", &n);

for(i=1; i<=n; i++)
```

```
{
for(j=1; j<=i; j++)

printf("%d", i);
printf("\n");
}
getch();
}
```

Ex. Write a program to print -

```
    1
   2 2
  3 3 3
 4 4 4 4
```

```
void main()
{
 int i,j,n;
clrscr();

printf("Enter the N:");
scanf("%d", &n);

for(i=1; i<=n; i++)
{

j=n;
```

```
while(j>=i)
{
 printf(" ");
  j--;
}

for(j=1; j<=i; j++)

printf("%d", i);
printf("\n");
}

getch();
}
```

Ex. Write a program to print -

```
      1
    2 2 2
  3 3 3 3 3
4 4 4 4 4 4 4
```

```
 main()
{
 int i,j,n;
clrscr();
printf("Enter the N:");
scanf("%d", &n);
```

```
for (i = 1; i <= n; i++)
{
 j = n;
while ( j >= i)
{
 printf (" ");
  j--;
}
for (j = 1; j <= i ; j++)

printf ("%d", j);

 j = i ;

while (j > 1)
{

printf ("%d", j);

 j-- ;
 }

printf ("\n");
 }

getch ( );
 }
```

Ex. Write a program to print -

```
4 3 2 1
3 2 1
2 1
1
```

```
void main ( )
{

int i, j, n;
clrscr ( );
printf ("Enter the N: ");
scanf ("%d", &n);

for (i = n; i >= 1; i--)
{

for ( j = i ; j >= 1 ; j--)

printf ("%d", j);

printf ("\n");
}

getch ( );
 }
```

Ex.

```
void main ( )
{
 int i, j ;
clrscr ( );

for ( i = 1 ; i <= 4 ; i++ )
{

for ( j = 1 ; j <= 4 ; j++ )
{

printf ( "%d", j );
 if ( i == j )
break ;
}

printf ( "\n" );
}
getch ( );
 }
```

O/P →

```
1
1 2
1 2 3
1 2 3 4
```

Ex.

```
void main (void )
{
 int i, j ;
clrscr ( );

for ( i = 1 ; i < 6 ; i++ )
{

for ( j = 1 ; j <= 5 ; j++ )
{

if ( i == j )

continue ;
printf ( "\t %d", j );
 }

printf ( "\n" );
 }

getch ( );
 }
```

O/P →

```
2  3  4  5
1  3  4  5
1  2  4  5
1  2  3  5
1  2  3  4
```

Second logic (for spaces):-

Ex.
```
void main ()
{
 int i, j, r, s;
char ch = '1';

clrscr ();
printf ("Enter the row size:");
scanf ("%d", &r);

for (i=1; i<=r; i++)
{

for (s=1; s<=38-i*2; s++)
printf (" ");

for (j=1; j<=i; j++)
{

textcolor (9);
cprintf ("%4c", ch);
}

printf ("\n");
}
getch ();
}
```

Ex. Write a program to print

```
5  5  5  5  5
5  5  5  5  5
5  5  5  5  5
```

```
main ()
{
 int i;
for (i=1; i<=15; i++)
{
  printf ("5");
if (i%5==0)
printf ("\n");
}
getch ();
}
```

Ex. Write a program to print

```
1  1  1  1  1
1  0  0  0  1
1  0  0  0  1
1  0  0  0  1
1  1  1  1  1
```

```
main ()
{
 int i, j, n;
```

```
clrscr();
printf("Enter the value : ");
scanf("%d", &n);

for(i=1; i<=n; i++)
{

for( j=1; j<=n; j++)

if(i==1 || i==n || j==1 || j==n)

printf("1");
else
printf("0");
printf("\n");
}

getch();
}
```

Ex: Write a program to print

```
1 2 3
6 5 4
7 8 9
```

```
main()
{
int i,j,n,k=1;
clrscr();
printf("Enter the Number : ");
scanf("%d", &n);

for(i=1; i<=n; i++)
{
 for(j=1; j<=n; j++)
{

printf("%4d", k);

if(j!=n)
{

if(i%2!=0)

  k++;
else k
   k--;
}
}
  k = k+n;
printf("\n");
}
getch();
}
```

Ex. Write a program to print

```
1  0  0  0  1
0  1  0  1  0
0  0  1  0  0
0  1  0  1  0
1  0  0  0  1
```

```
main ()
{
int i, j, k, n;
clrscr ();
textmode (2);

printf ("Enter the number : ");
scanf ("%d", &n);

textcolor (4+128);

for ( i = 1, k = n; i <= n; i++, k--)
{
 for (j = 1; j <= n; j++)
{
if (i == j || k == j)

cprintf ("%-4d", 1);
else
printf ("%-4d", 0);
}
printf ("\n");
}

getch ();
}
```

Ex. Write a program to accept a number. Find out this no. is prime or not.

Third logic (for spaces) -:

Ex.
```
void main()
{
  int n, s;
clrscr();
printf("Enter the number :");
scanf("%d", &n);

s = 5 * n;

printf("%*c", s, 32);
getch();
}
```

Note :-

Here 32 is an ASCII code for space-bar.

Ex.
```
#include <stdio.h>
#include <conio.h>
void main()
{

int i, j, k, n, m;
clrscr();
printf("%-4d", j);
printf("Enter the number :");
scanf("%d", &n);
```

```
    m = n;
    for ( j = 1 ; j <= n ; j++ )
    {
    for ( j = n ; j > m ; j-- )
    {
    printf ( "%-4d", j);
    }
    for ( k = 1 ; k <= m ; k++ )
    {
    printf ( "%-4d", k );
    }
    m--;
    printf ( "\n");
    }
    getch ();
    }
```

O/p →

```
Enter the number : 3
    1   2   3
    3   1   2
    3   2   1
```

Ex: Write a program to print -

```
A B C D E F G F E D C B A
A B C D E F   F E D C B A
A B C D E       E D C B A
A B C D           D C B A
A B C               C B A
A B                   B A
A                       A
A B                   B A
A B C               C B A
A B C D           D C B A
A B C D E       E D C B A
A B C D E F   F E D C B A
A B C D E F G F E D C B A
```

```
#include <stdio.h>
#include <conio.h>
  void main ()
{
int i, j, p, x = 7, y = 7;
clrscr ();
for ( i = 1 ; i <= 13 ; i++)
{
```

```
p = 65;      /* for ASCII value
                of A */

for (j = 1; j <= 13; j++)
{
if (j > 1 && j >= x && j <= y)
                    /* for spaces */
printf ("%-3c", 32);
else
printf ("%-3c", p); /* for printing
                       letters */
if (j < 7)      /* for A to F */
p++;
else
p--;            /* for reverse order
                   F to A */
}               // inner

if (j > 1)
{
if (j < 7)
{           /* for first half increasing
               spaces */
x--;
y++;
}

else        /* for second half
               decreasing spaces */
{
x++;
y--;
}
}
printf ("\n");
}

getch ();
}
```

Ex.

```
# include <stdio.h>
# include <conio.h>
  void main ()
{
  int i, j, p, n, s;
clrscr ();

printf ("Enter the number :");
scanf ("%d", &n);

s = 5 * n;
printf ("%*c", s, 32);

printf ("%d\n", 0);

s = s-4;
```

```
for(j=2; j<=n+1; j++)
{

p=j-1;
printf("%*c", x, 32);

for(j=1; j<=2*j-1; j++)
{

if(p>=0)
printf("%-4d", p);
else
printf("%-4d", -(p));
p--;
}

printf("\n");
x=x-4;
}
getch();
}
```

o/p→

```
Enter the number: 4
                0
            1   0   1
        2   1   0   1   2
    3   2   1   0   1   2   3
4   3   2   1   0   1   2   3   4
```

Ex Write a program to accept two nos. Display all the prime nos in b/w these two nos.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
long int n1, n2, n, cnt=0, t, flag;

clrscr();
for(n=n1; n<=n2; n++)
{

for(flag=1, t=2; t<=sqrt(n); t++)
{

if(n%t==0)
{
flag=0;
break;
}
}

if(flag==1)

printf("\n%2ld PRIME=%5ld",
                ++cnt, n);
```

```
}
getch();
```

O/P →

```
Enter two numbers : 1  10
1 PRIME = 1
2 PRIME = 2
3 PRIME = 3
4 PRIME = 5
5 PRIME = 7
```

Ex. Write a program to accept a number. Find out the square-root of that no. (without using sqrt function).

```
#include <stdio.h>
#include <conio.h>
void main()
{

int n, j;
float j;

clrscr();
printf("Enter the number : ");
scanf("%d", &n);

for(j=1; j*j<n; j++);
if(j*j==n)
{

printf("Square-root of %d is %d\n",n,j);
printf("(%c %d = %c %d, 251, n, 241, j);
}

else
{

for(j=j-1; j*j<n; j=j+0.00001);

printf("Square-root of %d is %.3f\n", n,j);
printf("%c %d = %c %.3f", 251, n, 241, j);
}

getch();
}
```

O/P →

① Enter the number : 81
Square-root of 81 is 9
$\sqrt{81} = \pm 9$

② Enter the number : 56
Square-root of 56 is 7.483
$\sqrt{56} = \pm 7.483$

Ex. Write a program to print -

```
        1   1   1   1
    2   2   2   2
3   3   3   3
```

```
main()
{
int i, j, s, k;
clrscr();
printf("\n");

for (j=1; j<=3; j++)
{
for (s=1; s<32-3*j; s++)
printf(" ");

for (j=1; j<=i; j++)
printf("%6d", j);

for (k=4-j; k!=0; k--)
printf("%6d", j);

printf("\n");
}

getch();
}
```

Ex. Write a program to print -

```
4                       4
3 3                   3 3
2 2 2               2 2 2
1 1 1 1           1 1 1 1
0 0 0 0 0 0 0 0 0 0 0
  1 1 1 1 1 1 1
    2 2 2 2 2
      3 3 3
        4
```

```
main()
{
int i, j, n, k;
clrscr();
printf("Enter the number:");
scanf("%d", &n);

for (i=n; i>=1; i--)
{

for (k=n-1+1; k>=1; k--)
printf("%d", j);
for (j=1; j<=2*j-1; j++)
printf(" ");
for (k=n-j+1; k>=1; k--)
printf("%d", j);
```

```
printf("\n");
}
for(i=0; i<=n; i++)
{
for(j=1; j<=i; j++)
printf(" ");
for(k=2*(n-i)+1; k>=1; k--)
printf("%d", i);
printf("\n");
}
getch();
}
```

O/P → Enput the number: 3

```
  3           3
  2 2       2 2
  1 1 1   1 1 1
0 0 0 0 0 0 0
  1 1 1 1 1
    2 2 2
      3
```

Ex: Write a program to print —

```
1 1 1 1 1 1 1
1 2       2 1
1 1       1 1
1 0       0 1
1 1       1 1
1 2       2 1
1 1 1 1 1 1 1
```

```
main()
{
int i, j, n, k;
clrscr();
printf("Enter the row size:");
scanf("%d", &n);
for(k=-n; k<=n+2; k++)
printf("1");
printf("\n");
for(i=-n; i<=n; i++)
{
for(k=-n; k<=-n; k++)
printf("1");
for(j=-n; j<=n; j++)
{
```

```
if(abs(i) == abs(j))
printf("%d", abs(i));
else
printf(" ");
}
for(k=-n; k<=-n; k++)
printf("1");
printf("\n");
}
for(k=-n; k=n+2; k++)
printf("1");
getch();
}
```

O/p →

```
Enter the row size: 2
1 1 1 1 1 1 1
1 2 2 1
1 1 1 1
1 0 1
1 1 1 1
1 2 2 1
1 1 1 1 1 1 1
```

E: Write a program to print →

```
    3       3
  2           2
    1       1
      0
    1       1
  2           2
    3       3
```

```
main()
{
int i, j, n;
clrscr();
printf("Enter the row size:");
scanf("%d", &n);
for(i=-n; i<=n; i++)
{
 for(j=-n; j<=n; j++)
{
 if(abs(i) == abs(j))
printf("%d", abs(i));
else
printf(" ");
}
```

```
printf("\n");
}

getch();
}
```

o/p → Enter the row size: 3

```
  3                 3
    2             2
      1         1
        0
      1         1
    2             2
  3                 3
```

dowhile -:

"dowhile" is an exit control looping statement.

Syntax -

```
initilization ;
do
{
  statement 1;
  statement 2;
  ---------
  ---------

  updation ;
}

while (condition);
```

First it will enter into the loop and execute the body atleast for one time and then it check the conditions. if the condition is satisfied then it will re-enter into the loop. This process of loop rotation will be taken place untill the condition is dis-satisfied.

In 'dowhile' the while will have the semicolon. (;)

Difference between 'while' and 'dowhile' -:

'while' first check the condition and then enter into the body.
'dowhile' first enter into the body and then checks the condition.

Note -:

1. Inside the body 'while' will not present without semicolon (;), its only normal 'while' and after the body 'while' will present with semicolon (;)

2. In 'dowhile' after the body is closed the next statement must be 'while' with condition including semicolon (;).

3. In 'dowhile' within the body there is no while with condition including semicolon (;)

Ex.

```
main ()
{
 int i= 10;
do
   printf("%d", --i);
while (--i) ;
```

```
getch();
}
```

o/p → 9 7 5 3 1

Ex.
```
main()
{
int i=1;
do
while (i++<1);
while (i++<=2);
printf("%d", i);
getch();
}
```

Replacement
```
do
{
while (i++<1)
{
}
}
while (i++<=2);
printf("%d", i);
```

o/p → 5

Ex.
```
main()
{
int i=1;
do
while (i++<=1);
while (i++<=3);
while (i++<=5);
printf("%d", i);
}
```

Replacement
```
do
{ while (i++<=1)
{}
}
while (i++<=3);
while (i++<=5)
{}
printf("%d", i);
}
```

o/p → 7

Number System –:

'C' Language support's four types of formats. A number can be represented in any one of these formats –

1. Binary format.
2. Octal format.
3. Hexa-decimal format.
4. Decimal format (like integer)

1. Binary format –:

Base of Binary is 2. and it contains 2 numbers that is. 0 and 1. So a binary data can be written with the by using 0's and 1's.
Ex. 01011O, ob101. etc.

2. Octal format –:

Base of octal is 8. and it contains 8 numbers that are 0, 1, 2, 3, 4, 5, 6, 7.

So an octal data can be written by using these eight numbers.

As per the 'C' Language if the number has to treate an octal, Define the number with 0 (zero).

Ex: 076, 045 etc.

3. Hexadecimal format -:

Base of Hexa-decimal is 16. and it contains 8 numbers. They are-

0, 1, 2, 3, 4, 5, 6, 7, 8, 9,

10, 11, 12, 13, 14, 15
↓ ↓ ↓ ↓ ↓ ↓
A B C D E F

as shown above hexadecimal format contains numbers from 0 to 9 and some alphabets like A, B, C, D, E, F. These alphabets are equivalent to 10, 11, 12, 13, 14 and 15 respectively.

As per the 'C' compiler if the number has to treat as hexadecimal, define the number with 0x (zero x).

Ex: 0x567
0x989
0x532 etc.

4. Decimal format -:

Decimal formats are integer numbers.

Ex: 456, 8083, 53 etc.

Base of decimal is 10 and it contains 10 numbers. They are-
0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

By using these numbers we can represent any decimal number.

Note -:

In Binary format, we can write a binary data by using 0b (zero b). but it is optional.

Ex: 0b101 or 101
both are valid.

Ex. Convert into Hexa and octal

i). 100
ii). 30
iii). 20

i).

8 | 100		R
8 | 12		4
8 | 1		4
0		1

16 | 100		R
16 | 6		4
0		6

So octal of 100 ⇒ 0 144
Hexa of 100 ⇒ 0x 64

ii)

8 | 30	R
8 | 3	6 ↑
0	3

16 | 30	R
16 | 1	14 (E) ↑
0	1

So octal of 30 ⇒ 0 36
Hexa of 30 ⇒ 0x 1E

iii).

8 | 20	R
8 | 2	4 ↑
0	2

16 | 20	R
16 | 1	4 ↑
0	1

So octal of 20 ⇒ 0 24
Hexa of 20 ⇒ 0x 14

Ex. Add (① ① ①)

```
  0 7 4 3
  0 1 2 4
+ 0 6 2 5
-------------
  0 1 7 1 4
```

8 | 12
① 4 ✓

8 | 9
① 1 ✓

8 | 15
① 7 ✓

Subtract (8 ↓)

```
  0 4 7 6
- 0 2 3 7
-------------
  0 2 3 7
```

8+6 = 14

Ex. Add.

(① ①)

```
  0 x 1 7 4
  0 x a 2 d
+ 0 x 9 3 b
-------------
  0x 1 4 d C
```

4+d+b
4+13+11 = 28

16 | 28
① 12 → C ✓

1+a+9 = 20

16 | 20
① 4 ✓

Ex. Add. (① ② ①)

```
  0 x 9 d b
  0 x 6 e 5
+ 0 x a b c
-------------
  0x 1 b 7 C
```

b+5+C = 11+5+12

= 16 | 28
① 12 → C ✓

1+d+e+b
1+13+14+11

= 16 | 39
② 7 ✓

2+9+6+a
= 27 ⇒ 16 | 27
① 11 → b ✓

Ex. Subtract (16, 16)

```
  0 x 7 3 2
- 0 x 2 a C
-------------
  0 x 4 8 6
```

16+2 = 18 - C
18 - 12 = 6
2 - a = (16+2) - a
= 18 - 10 = 8
5 - 2 = 3

Ex. Subtract

```
  0 7 9 4
- 0 1 2 4
```

it will show an error because octal format does-not contains 9.

Ex:

```
# include <stdio.h>
# include <conio.h>
 void main()
 {

  int a = 14;

 clrscr();

 printf("%d \n", a);
 printf("%o \n", a);
 printf("%#o", a);

 printf("\n %x", a);
 printf("\n %#x", a);
 printf("\n %#X", a);

 printf("\n %b", a);
 getch();
 }
```

O/P →

```
14
16
016
e
0xe
0XE
%b
```

Binary codes :-

0 — 0000
1 — 0001
2 — 0010
3 — 0011
4 — 0100
5 — 0101
6 — 0110
7 — 0111
8 — 1000
9 — 1001
10 → 1010
97 — 1100001

Binary codes in increasing bits :-

$2^1 - 1 = 1 = 1$
$2^2 - 1 = 3 = 11$

$2^3 - 1 = 7 = 111$
$2^4 - 1 = 15 = 1111$

$2^5 - 1 = 31 = 11111$
$2^6 - 1 = 63 = 111111$

$2^7 - 1 = 127 = 1111111$
$2^8 - 1 = 255 = 11111111$

Ex Add

$$\begin{array}{r} (1)(1)(1)(1)(1)(1) \\ (1)\,1\,1\,0\,1\,1 \\ \quad 1\,0\,1\,0\,1 \\ +\quad 1\,1\,0\,0\,1 \\ \hline 1\,0\,0\,1\,0\,0\,1 \end{array}$$

Ex. Subtract

$$\begin{array}{r} 1\,1\,0\,1\,1 \\ -\;1\,0\,1\,1\,0 \\ \hline 0\,0\,1\,0\,1 \end{array}$$

Ex. Calculate

$a = 10 + 010 + 0x10 + 0x10$;

First method -:

Convert all nos. into equivalent decimal nos and then ~~add~~. solve the expression.

$a = 10 + (1*8^1 + 0*8^0) + (1*16^1 + 0*16^0) + (1*16^1 + 0*16^0)$

$a = 10 + 8 + 16 + 16$

$a = 50$.

Second method -:

Convert all nos into its equivalent binary nos. and then add.

$a = 10 + 010 + 0x10 + 0x10$

$10 \rightarrow$ ①① 1010

$010 \rightarrow$ 001000

$0x10 \rightarrow$ 000100000

$0x10 \rightarrow +$ 000100000

00110010

Now convert it into its equivent decimal.

$a = (1*2^5) + (1*2^4) + (0*2^3) + (0*2^2) + (1*2^1) + (0*2^0)$

$a = 32 + 16 + 0 + 0 + 2 + 0$

$a = 50$.

Note -!

When a program is implementing if a variable is declared then first of all it allocates some bytes of memory ; for suppose.

char ch ; (1 byte = 8 bits).

Since character allocates 1 byte of memory -

1 byte = 8 bits

1 1 1 1 1 1 1 1 → $2^8 = 255$

Physical memory is created once you created a variable like above.

Note -:

Internally in the system everything is accepted as in the binary formats and then implementation will be takes place.

It can be represented either in 8 bit format or 16 bit format.

Better always represent by 16 bit format.

Ex. 0000 0000 0000 0000 → 0

0111 1111 1111 1111 → 32767

1000 0000 0000 0000 → -32768

1111 1111 1111 1111 → -1

Reverse Combination -:

1. If the given number is positive then the result will be negative by increment 1 value for the positive number.

Ex. 9 = -10

9 → 0000 0000 0000 1001

-10 → 1111 1111 1111 0110

Ex. 4 = -5

4 → 0000 0000 0000 0100

-5 → 1111 1111 1111 1011

2. If the given number is negative then the result will be positive by decrementing 1 value for the negative number.

Ex. -100 = 99

-100 → 1111 1111 1001 1100

99 → 0000 0000 0110 0011

Ex. -10 = 9

-10 = 1111 1111 1111 0110

9 = 0000 0000 0000 1001

Ex. Add 32767
+ 1

32767 → 0111 1111 1111 1111
+
1 → 0000 0000 0000 0001
1000 0000 0000 0000
= -32768

So 32767
+ 1
-32768

Ex. Subtract -32768
- 1

-32768 → 1000 0000 0000 0000
-
1 → 0000 0000 0000 0001
0111 1111 1111 1111
= 32767

So -32768
- 1
32767

Ex. Add -32768
+ -1

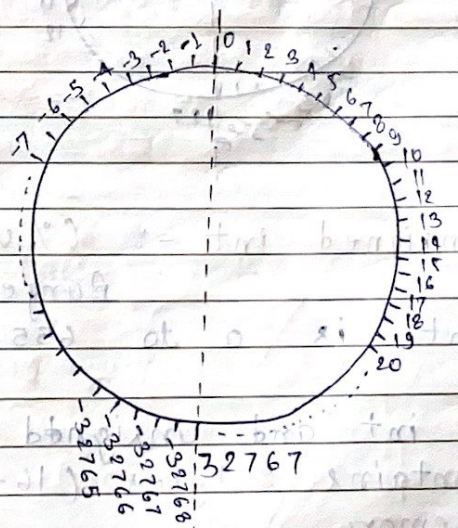
-32768 → 1000 0000 0000 0000
+
-1 → 1111 1111 1111 1111
1 0111 1111 1111 1111

ignore — only 16 bits will be taken.

= 32767

So -32768
+ -1
32767

Note :-
for int the ranges will be like - (%d)

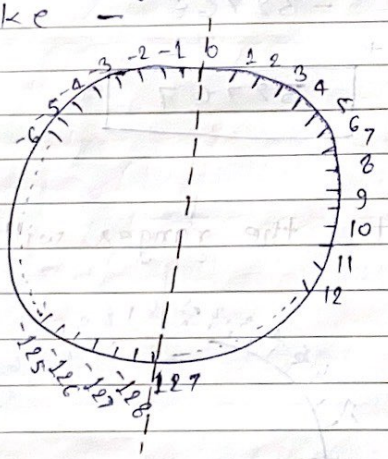


Char -: (%c)

Range of character is -128 to 127.

0 →	0000 0000
127 →	0111 1111
-128 →	1000 0000
-1 →	1111 1111

Character contains 1-byte (8-bits) of memory. its ranges will be like -



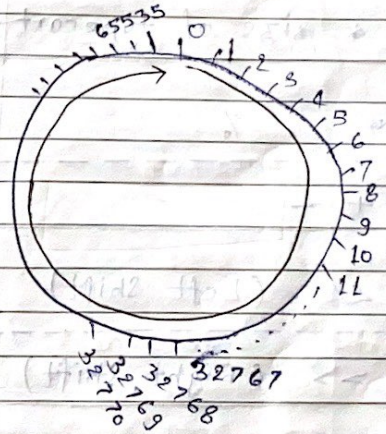
Unsigned int -: (%u)

Range of unsigned int is 0 to 65535.

int and unsigned int both contains 2 bytes (16-bits) of memory.

%d	Binary Code	%u
-1	1111 1111 1111 1111	65535
0	0000 0000 0000 0000	0
-32768	1000 0000 0000 0000	32768
32767	0111 1111 1111 1111	32767

Range of unsigned int will be like -



Ex.

```
void main ( )
{
printf("%d %u %o %x", -1, -1, -1, -1);
getch ( );
}
```

O/P → -1 65535 177777 ffff

Category	Operator Name	Associative
Heighest Category	( ) [ ] → .	L to R
Unary Category	+ − ! ~ (Bitwise) ++ −− & * sizeof typecast	R to L
Arithmatic Category	* / % + −	L to R
Bitwise Category	<< (Left Shift) >> (Right Shift)	L to R
Relational Category	< <= > >= == !=	L to R
Bitwise Category	& (Bitwise AND) ^ (Bitwise Exclusive OR)	L to R
Bitwise Category	| (Bitwise OR)	L to R.

Category	Operator Name	Associative
Logical Category	&& (Logical AND) || (Logical OR)	L to R
Ternary Category	? :	R to L.
Assignment Category	= += /= %= += -= <<= >>= &= ^= |=	R to L
Comma Category	,	L to R
—	++ (post increment) -- (post decrement	—

1's complement -:

Ex: main ()
{
printf ("%d", ~1);
}

o/p → -2

Ex: main()
{
printf("%d", ~-32768);
}

o/p → 32767

Ex: main()
{
printf("%d", ~~-32768);
}

o/p → -32768

Ex: main()
{
printf("%d", ~32768);
}

o/p → -32769

Left shift (<<) :-

Ex: 5 << 1

5 → 0000 0000 0000 0101

Now shift 1 bit to the left side -

0 0000 0000 0000 1010 ← in

out

this is equivalent to 10.

~~80~~ 5 << 1 = 10

Ex:

5 << 2	0000 0000 0000 0101 0000 0000 0001 0100	20
5 << 3	0000 0000 0000 0101 0000 0000 0010 1000	~~30~~ 40
5 << 4	0000 0000 0000 0101 0000 0000 0101 0000	80
5 << 5	0000 0000 0000 0101 0000 0000 1010 0000	160
11 << 2	0000 0000 0000 1011 0000 0000 0010 1100	44

We can also calculate it as -

5 << 1 = 5 * 2^1 = 10

5 << 2 = 5 * 2^2 = 20

5 << 3 = 5 * 2^3 = 40

5 << 4 = 5 * 2^4 = 80

5 << 5 = 5 * 2^5 = 160

11 << 2 = 11 * 2^2 = 44

Right Shift (>>) :-

20 >> 1	0000 0000 0001 0100 0000 0000 0000 1010	10
20 >> 2	0000 0000 0000 0100 0000 0000 0000 0101	5
20 >> 3	0000 0000 0001 0100 0000 0000 0000 0010	2
20 >> 4	0000 0000 0000 0100 0000 0000 0000 0001	1
20 >> 5	0000 0000 0001 0100 0000 0000 0000 0000	0
7 >> 2	0000 0000 0000 0111 0000 0000 0000 0001	1

We can also calculate it as :-

20 >> 1 = $20/2^1$ = 10
20 >> 2 = $20/2^2$ = 5
20 >> 3 = $20/2^3$ = 2
20 >> 4 = $20/2^4$ = 1
20 >> 5 = $20/2^5$ = 0
7 >> 2 = $7/2^2$ = 1

Ex:

```
main()
{
printf("%d", -10 >>1);
getch();
}
```

o/p → -5

Explanation :-

Since -10 is reverse of 9 so -

9 = 0000 0000 0000 1001

-10 = 1111 1111 1111 0110

Now Right shift it by 1. then -

1111 1111 1111 1011

it is -5 because its reverse is 0000 0000 0000 0100 = 4.

Hence output = -5

Note-1.

Any number 16 times left shift leads to 0 (zero)

Ex. 7984 << 16 → 0.

Note-2 :-

Any odd number 15 times left shift leads to -32768.

Ex. 97 << 15 → -32768

Note -3 :-

Any even number 15 times left shift leads to 0 (zero).

Ex. 84 << 15 → 0

Note -4 :-

For any negative number right shift of 15 times leads to -1.

Ex. -9845 >> 15 → -1

Truth Table :-

1. Bitwise AND (&)

A	B	R
0	0	0
0	1	0
1	0	0
1	1	1

2. Bitwise OR (|)

A	B	R
0	0	0
0	1	1
1	0	1
1	1	1

3. Bitwise Exclusive OR (^)

A	B	R
0	0	0
0	1	1
1	0	1
1	1	0

Ex. x = 8, y = 10

Findout x & y, x | y and x ^ y.

x → 0000 0000 0000 1000

y → 0000 0000 0000 1010

x & y → 0000 0000 0000 1000 = 8

x | y → 0000 0000 0000 1010 = 10

x ^ y → 0000 0000 0000 0010 = 2

Utility of Bitwise Operator :-

1. The bitwise operators can be used inorder to check wheather a particular bit is ON or OFF.

2. It can be used to change the file attributes in our operating system files.

Ques. How to check whether a particular bit is on or off.

Ans. To check the bit is ON or OFF the bitwise of AND operator is suitable.

Ex: Suppose a byte that has a value 10101101.
We wish to check whether bit number 3 is ON or off. if it is 1 then ON otherwise OFF.

Since, we want to check the bit number 3, The second ~~bit~~ operand for AND operation, we choose in binary 0000 L000, which is equal to 8 in decimal.

Explanation :-

ANDing Operation :

```
1 0 L 0 1 1 0 1    original bit pattern.
0 0 0 0 L 0 0 0    AND mask
0 0 0 0 L 0 0 0    Resulting bit pattern.
```

The resulting value we get in this case is 8 .i.e. the value of the second operand. The result turned out to be 8.
Since the third bit of operand was ON

Ques. Which operator is suitable turning of a particular bit in an number?

Ans: Bitwise AND Operator (&) and 1's complement Operator (~).

Ex: To unset the 4th bit of byte data or to turn off a particular bit in a number.

Explanation :-
Let us assume character byte data = ob 00010111;

Now (byte_data) & (~(1<<4));

1 can be represented in binary as = 0000 0001

<< is left shift operator, it shifts the bit 1 by 4 places towards left.

So (1<<4) becomes ob0001 0000

~~ob 0000~~

And ~ is the 1's complement operator in "C" language

So

~(1<<4) means 1's complement of ob 0001 0000

i.e. ob 1110 1111.

Replacing value of byte_data and ~(1<<4) in (byte_data) & (~(1<<4));

we get (ob 0001 0111) & (ob 1110 1111);

Now Perform ANDing

```
  0001 0111
& 1110 1111
  ---------
  0000 0111
```

Thus the 4th bit is unset.

Ques. if x = 5, y = 12 then what will be the value of x and y after this expression—

x ^= y ^= x ^= y;

Sol. x ^= y ^= x ^= y;

x ^= y ^= x = x ^ y;

Now

x → 0000 0000 0000 0101

y → 0000 0000 0000 1100

x^y → 0000 0000 0000 1001

x^y → 9

∴ x ^= y ^= x = 9; ∴ x = 9

x ^= y ^= 9

x ^= y = y ^ 9

Now

y → 0000 0000 0000 1100

9 → 0000 0000 0000 1001

y^9 → 0000 0000 0000 0101

y^9 → 5

∴ x ^= y = 5 ∴ y = 5

∴ x = 9

y = 5 Ans.

Ex. Write a program to accept two numbers and findout its multiplication with the help of shifting operators.

```
void main()
{
int a, b, result;
clrscr();
printf("Enter the numbers to be
        multiplied : ");

scanf("%d %d", &a, &b);
result = 0;
while(b!=0)
{
if(b&01)
result = result + a;
a <<= 1;
b >>= 1;
}

printf("\n Result : %d", result);
getch();
}
```

O/p →

Enter the number to be multiplied
: 6 9
Result : 54

Ex.

```
main()
{
signed int bit = 512, i = 5;

clrscr();
for(; i; i--)
{

printf("%d \n", bit >> (i-(i-1)));
}

getch();
}
```

O/p → 256
256
256
256
256

Ex.

```
void main()
{
int i = 32, j = 0x20, k, l, m;
clrscr();
k = i | j;
l = i & j;
m = k ^ l;
printf("%d %d %d %d %d", i, j, k, l, m);
}
```

o/p → 32
32
32
32
0

Ex. main()
{
unsigned int res;
clrscr();
res = (64 >> (2+1-2)) & (~(1<<2));
printf("%d", res);
getch();
}

o/p → 32

Explanation -:
res = (64 >> (2+1-2)) & (~(1<<2));

res = (64 >> 1) & (~(1<<2));
res = 32 & (~(4));
= 32 & -5;
32 → 0000 0000 0010 0000
-5 → 1111 1111 1111 1011
32 & -5 → 0000 0000 0010 0000
→ 32

Ex. void main()
{
unsigned int a = 0xffff;
clrscr();
~a;
printf("%x", a);
getch();
}

o/p → ffff
because ~0 is non-updatable.

Ex. main()
{
printf("%x %x %x", 10, 010, 0x10);
getch();
}

o/p → a 8 10

Ex. main()
{
unsigned int a, b, c, d, e;
clrscr();
a = b = c = d = e = 32;
a << 2;
b >> 2;

```
c ^= 2;
d |= 2;
e &= 2;

printf("%x %x %x %x %x", a, b,
                         c, d, e);
getch();
}
```

o/p → 20 20 22 22 0

Ex.
```
main()
{
signed int bit = 512, mbit;
clrscr();
mbit = ~bit;
bit = bit & ~bit;
printf("%d %d", bit, mbit);
getch();
}
```

o/p → 0 -513

Ex
```
main()
{
char c = 48;
int i, mask = 01;
clrscr();
for(i=1; i<=5; i++)
{
 printf("%c", c | mask);
mask = mask << 1;
}
getch();
}
```

o/p → 1 2 4 8 0

Explanation :-

if i = 1, mask = 01

c | mask = 48 → 0011 0000

mask → 0000 0001

OR → 0011 0001

→ 49

Ascii code of 49 is '1'

for i = 2, mask = 010

c | mask = 48 → 0011 0000

mask → 0000 0010

OR → 0011 0010

→ 50

Ascii code of 50 is '2'

for i = 3, mask = 0100

c | mask = 48 | 0100

48 → 0011 0000
mask → 0000 0100
OP → 0011 0100
→ 52
Ascii code of 52 is '4'.

for i = 4, mask = 01000

c/mask ⇒ 48 → 0000
mask →

48 → 0011 0000
mask → 0000 1000
OR → 0011 1000
→ 56
Ascii code of 56 is '8'

for i = 5, mask = 01 0000

c/mask = 48 → 00 11 0000
mask → 00 01 0000
OP → 00 11 0000
→ 48
Ascii code of 48 is '0'.

Hence output of the above program will be

1 2 4 8 0 .

Ex. main ( )
{
int x;
clrscr ( );
x = 011 | 0x10;
printf ("%d", x);
getch ( );
}

o/p → 25

Explanation -:
x = 011 | 0x10;
Here 011 represents octal number which is 11
and 0x10 represents Hexa-decimal number which is 10

Now 011 → 0001001
0x10 → 0001 0000
OR → 0001 1011
Hence x = 011 | 0x10
= 0001 1001

its equivalent ~~binary~~ decimal is -
x = (1*2^4) + (1*2^3) + (0*2^2) + (1*2^0) + 0
x = 16 + 8 + 0 + 1
x = 25.

Ex:

```
main()
{
 int x = 0x1234 , y = 0x5555;
clrscr();
printf("0x%04.4x\n", x|y);

printf("0x%04.4x\n", x&y);

printf("0x%04.4x\n", x^y);

printf("0x%04.4x\n", ~x);
getch();
}
```

O/P → 0x5775
0x1014
0x4761
0xedcb

Explanation :-

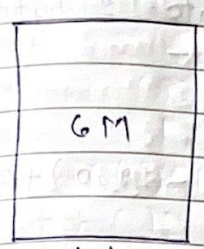
x = 0001 0010 0011 0100
y = 0101 0101 0101 0101

x|y → 0101 0111 0111 0101
5 7 7 5

x&y → 0001 0000 0001 0100
1 0 1 4

x^y → 0100 0111 0110 0001
4 7 6 1

Ex. Write a program to implement the date storing in two bytes.



database

first method :-
dd/mm/yy.

```
int dd;   } 2 bytes
int mm;   } 2 bytes   ⇒ 6 bytes
int yy;   } 2 bytes
```

so 6/6 = 1 M.

Second method :-

```
char dd;  } 1 bytes
char yy;  } 1 bytes   ⇒ 4 bytes
int  yy;  } 2 bytes
```

so 6/4 = 1.5 M

Third Method :-
int dd ⇒ 2 bytes

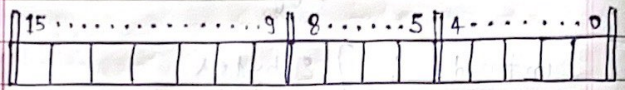
so 6/2 = 3 M.

Note -:

DOS and windows will store the date into 2 bytes by using the formula -

date = 512 * (year - 1980) + 32 * month + day.

ff-date



years since 1980 | month | Day

Ex: For suppose 09/03/1990 then the conversion is as -

date = 512 * (1990 - 1980) + (32 * 3) + 9

= 5120 + 96 + 9

date = 5225

The binary equivalent of 5225 is

0001 0100 0111 1001

This binary value is passed in the date field in the directory by using left shift and right shift of bits. We make the shift operation then we see that year, month and day exists as a bunch of bits in a sequential memory locations, so we have to seperate each of the process like year, month and day. which will be as -

0001 0100 0011 0100 1

year | month | day

Extracting year -:

To extract an year the date has to right shift by 9 times and then add 1980.

date = 0001 0100 0011 0100 1 >> 9

= 000

= 0000 0000 0000 1010

= 10

Now add this with 1980

so year = 10 + 1980

year = 1990

Extracting month -:

In order to extract a month left shift the date by 7 times and then right shift by 12 times.

date = 0001010 0011 01001 <<7

= 0011 0100 1000 0000 >>12

= 0000 0000 0000 0011

= 3

so month = 3

Extracting date -:

Inorder to extract the date first left shift the date by 11 times and then right shift by 11 times.

date = 0001010 0011 01001

left shift by 11

0001010 0011 01001 << 11

= 0100 1000 0000 0000 >>11

= 0000 0000 0000 1001

= 9

∴ day = 9.

Ex. date = 1/09/2011

date = 512 * (2011 - 1980) + (32 * 9) + 1

= 512 * 31 + 288 + 1

= 16161

The binary equivalent of 16161 is-

0011111 1001 00001

Extracting year -:

date = 0011111 1001 00001 >>9

= 0000 0000 0001 1111

= 31

∴ year = 31

Extracting month -:

date = 0011111 1001 00001 <<7

= 1001 0000 1000 0000 >>12

= 0000 0000 0000 1001

= 9

∴ month = 9.

Extracting day.

date = 0011111 1001 00001 <<11

= 0000 1000 0000 0000 >>11

= 0000 0000 0000 0001

= 1 ∴ day = 1.

Ex:

```
void main ()
{
unsigned int
d = 9, m = 3, y = 1990, year,
month, day, date;
date = (y - 1980) * 512 + (m * 32) + d;
printf ("\n Date = %u", date);
year = 1980 + (date >> 9);
month = ((date << 7) >> 12);
day = ((date << 11) >> 11);
printf ("\n Year = %u", year);
printf ("\n Month = %u", month);
printf ("\n Day = %u", day);
getch ();
}
```

o/p →

```
Date = 5225
Year = 1990
Month = 3
Day = 9
```

Ex:

```
void main ()
{
clrscr ();
printf ("\n %d %d", 32 << 1, 32 >> 0);
printf ("\n %d %d", 32 << -1, 32 >> -0);
printf ("\n %d %d", 32 >> 1, 32 >> 0);
printf ("\n %d %d", 32 >> -1, 32 >> -0);
getch ();
}
```

o/p →

```
64    32
0     32
16    32
0     32
```

Functions -:

A function is a self defined block which performs a pre-defined task.

Classification of Function -:

Classification of function can be taken place in two ways -

1. Library Functions.
2. User defined function.

1. Library Function -:

Defined in the language provided along with the compiler.

Ex. printf ();
scanf ();
sqrt (); etc.

2. User Defined Functions -:

These Functions written by the user

Ex. main ()
calculate ()
sum () etc

or any function defined by user.

Ques. Why main () is called user defined function ?

Ans. The main () body is totally supported by as per the user requirements. As the main () will execute the program and developes different executive files every time of the main execution. So main () is called as user defined function.

Function Return type -:

Generally a function has a default nature i.e every function at the time of execution, it has by default retun type is integer and every function returns a value. Recieving the value is optionel.

The return type of the function can be changes as per the user requirement, if it is user defined function.

As per the ANSI-C recomended

```
int main ()
{
  return 0;
}
```

Ex.
```
float main()
{
printf("%d", sizeof(main()));
getch();
}
```

o/p → 4 bytes

"void" return type -:

Inorder to not to define any size or return type for the function, we can write the return type as void. "void" is a datatype which kills the nature of the function.

Placing the return type as void to the function indicates we can-not send any value to the function. The size of void means nothing.

Ex.
```
void main()
{
---
---
}
```

Here void means no return type.

Whenever any function, if the return type is mentioned as void, we can-not pass any value for that function.

Rules to create User defined functions -:

In order to create user defined functions we have to follow some rules and regulations by defining the components, names of the function.

Components of the function -:

1. Function declaration. (Function Prototype)
2. Signature of a function.
3. Function caller.
4. Actual Arguments
5. Formal Parameters.
6. Function callee (declarator)
7. Function definition.
8. Return Type.

1. Function Declaration -:

The function declaration is also called as function prototype.

The function declaration contains -

i) Return type of the function.
ii) Function name
iii) Signature of the function.

For the predefined functions the declarations are available in the header files.
And For user defined functions the user has to provide the declaration explicitely.

Defining the function prototype is very helpfull for the compiler. By telling to the compiler return type of the function, type of arguments and number of arguments. Before execution of program due to this the borden will be reduce at the time of execution.

syntax -

```
Return_Data_type Function_name
(data_type arg1, data_type arg2
 ---------- data_type argn);
```

Ex. int validate_Date (int iDay, int iMonth, int iYear);

Ex. void fact (int);

Ex. void febo (int);

Ex. void power (int, int);

Ex. void sum (int, int);

↓ return type ↓ function name ↓ signature.

Note -:

Function prototype is optional in 'C' language, since 'C' language is not a strongly type checking. In 'C++' language prototype is mandatory since it is strongly type checking.

2. Signature of a function -:

When we declare a function the signature of the function always depends on number of input variables in a program.

3. Function caller -:

The function caller is utilization of function. Function caller can be takes place anywhere in the program and a function caller can be any number of times in a program.

When the function caller is taken place, it immediately jumps to the submodule (callee).

When the function caller is taking place there is no need to mension any return type of the function and the argument type.

4. Actual Arguments -:

The actual arguments of the function will be takes place at the time of function caller. The actual arguments will contains the data.

The arguments can be variable type, constant nos or expressions.

5. Formal Parameters -:

When the function caller is takes place, the values of the actual argument will be takes place to the formal parameter.

```
void main()
{
int iday, imonth, iyear;
- - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - -
ivalidate (iday, imonth, iyear); //caller
                  Actual arguments
- - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - -
}   // end of main()

int ivalidate (int x, int y, int z) //caller.
                Formal Parameter      (declator)
{
- - - - - - - - - -
- - - - - - - - - -   } Function
- - - - - - - - - -     Definition
- - - - - - - - - -
return 0;
// return back to caller.
}
```

6. Function Callee -:

The function callee or declarator is the first line of your definition. It is similar to main(), which is called as seperate module ...lee & contains the formal ...ameters and it contains ...lues recieved from the act... arguments.

The formal parameters ... be variable type. It must ... should satisfy the function ...claration type.

Function Definition -:

One definition ...n-not contain another definition, ...finition will be present only ... but the callers can ...ny number of times.

...turn Statement -: (Return Type)

The return statement will present in the submodule. It can be return back to the caller by sending a value.

The return statement can return only one value at a time to the calling function.

Return statement is possible when the caller of the function will have some return type other than 'void'

Syntax -

return (expression);

Ex. return (iNumber * iNumber);

Ex. return 0;

Ex. return (3);

Ex. return (10 * i);

Ex. return ; // it returns only control not value.

Way to write Function Programs -: A

Function can be write in four different ways -

1. Function with no argument, no return value.

Ex. clrscr();

6. Function Callee -:

The function callee or declarator is the first line of your definition. It is similar to main(), which is called as seperate module Callee f contains the formal parameters and it contains the values recieved from the actual arguments.

The formal parameters must be variable type. It must and should satisfy the function declaration type.

7. Function Definition -:

One definition can-not contain another definition. Definition will be present only one time but the callers can present any number of times.

8. Return Statement -: (Return Type)

The return statement will present in the submodule, It can be return back to the caller by sending a value.

The return statement can return only one value at a time to the calling function.

Return statement is possible when the caller of the function will have some return type other than 'void'

Syntax -

return (expression);

Ex. return (iNumber * iNumber);
Ex. return 0;
Ex. return (3);
Ex. return (10 * i);

Ex. return ; // it returns only control not value.

Way to write Function Programs -: A

Function can be write in four different ways -

1. Function with no argument, no return value.

Ex. clrscr();

2. Function with no argument and return value.

Ex: int getch(void);
OR int getch();

3. Function with argument and no return value.

Ex: gotoxy(int x, int y);

4. Function with argument and return value.

Ex: int printf(const char *format,...);

Based on these four categories function programs can be implement.

Function with no argument, no return value.

```
void printline();   // declaration
void test(void);    // declaration

int main()
{
 clrscr();
printline();        // caller
printf("Functions provides modular approach\n");
test();             // caller
printf("Back to main() module\n");
getch();
return 0;
}

void printline(void)   // callee
{
 int i;
for(i=1; i<=40; i++)
printf("-");
printf("\n");
}      // end of printline()

void test()   // callee
{
printf("It is a sub module\n");
}      // end of test()
```

O/P →

Functions Provides modular Approach
It is a sub module.
back to main() module.

Advantages of Functions :-

1. Functions provides modularity for the program. Due to the modular program it is easy to code and debug.

2. Functions supports source code reusability i.e. once a function is written, it can be called from any other module without having to rewrite the same. This saves time in rewritting the same code.

Note -

Program compilation always starts from top to bottom, and program execution always stars from main to main.

Ex. Example for compilation first and execution next -

```
void printline ()
{
  ____
  ____
}
```

```
void test ()
{
  ___
  ___
}

void main ()
{
  ____
  ____
  ____
}
```

In the above example, I not mension the decloration since compilation starts always from top to bottom. so the defmition already readen by the compiler before execution. so declarations are not neccessary.

Note -

The variables which has been defined in one module or scope, that variables will be used in that module only. It is not possible to use in another module without decoration.

Ex. WAP to accept a number. Print its factorial with the help of function.

```
void fact();          // declaration
void main()
{
  clrscr();
  fact();             // caller
  getch();
}

void fact()           // callee
{
 int num;
 long f=1;

 printf("Enter the number :");
 scanf("%d", &num);

 if (num==0 || num==1)
 {
 printf("Factorial is 1");
 getch();
 exit(0);
 }

for(; num>=2; f=f*num--);
printf("Factorial is %dd", f);
}
```

c/p →

```
Enter the number : 4
Factorial is 24
```

Ex. Write a program to print sum of two nos with the help of functions.

```
void sum();           // declaration
void main()
{
 clrscr();
 sum();               // caller
 getch();
}

void sum()            // callee
{
 int a, b, sum;
printf("Enter two numbers :");
scanf("%d %d", &a, &b);
sum= a+b;
printf("Sum is : %d", sum);
}
```

2. Function with no argument and no return type :-

In this category there is a communication from the caller and there is no communication from the callee (submodule)

Ex. Write a program to accept two numbers as a base and power. Find out the power value with the help of function.

```
void power(int, int); //declaration
void main()
{

int b, e;
clrscr();

printf("Enter base and power : ");
scanf("%d %d", &b, &e);
getch(); power(b,e); //caller
}       getch();

void power(int b, int e) //callee
{

int p = 1;
if(e==0)
{
printf("Power is : 1 \n");
getch();
exit(0);
}

while(e)
{

p = p*b;
e--;
}
printf("Power is : %d", p);
}
```

o/p →

Enter base and power : 2 5
Power is 32

Note :-

The actual argument and formal parameter's names can be change or same but their physical memory locations are different.

Ex. Write a program to accept a number and Findout its factorial

```
void fact (int);        // declaration
void main()
{
 int num;
 clrscr();
printf("Enter the number : ");
scanf("%d", &num);
fact(num);              // caller
getch();
}

void fact(int num)      // callee.
{
 long f=1;
if(num==0 || num==1)
 {
printf("Factorial is : 1 \n");
getch();
exit(0);
 }
 for(; num>=2; f=f*num--);
printf("Factorial is : %ld", f);
 }
```

o/p → Enter the number : 5

Ex. Write a program to print "WELCOME" in the center of the output screen.

```
void move(int r, int c)  // declaration
{
 int i;
clrscr();
for(i=1; i<r; i++)
 {
 printf("\n");
 }
printf("%*c", c+1, 32);
 }
void main()
{
 int x, y;
 move(12, 35);          // caller
printf("WELCOME");
getch();
 }
```

Ex. Write a program to accept a number. Display the given number in words

```
#include <stdio.h>
void main()
{
 int txt(long);          // declaration
long int num, t;
clrscr();

printf("Enter the number : ");
scanf("%ld", &num);

if (num == 0)
printf("ZERO");
if( num >= 10000000)
{
  t = num/10000000;
if (t <= 20)
txt (t);                 // caller
else
{
txt (t/10 * 10);         // caller
txt (t %10);             // caller
}

txt (10000000);          // caller
}

num = num % 10000000;

if (num >= 100000)
{
if  t = num/100000;

if ( t <= 20)
txt (t);                 // caller
else
{
txt (t/10 *10);          // caller
txt (t % 10);            // caller
}

txt (100000);            // caller
}

num = num % 100000;
if (num >= 1000)
{

t = num/1000;
if ( t <= 20)

txt (t);                 // caller
else
{

txt ( t/10 * 10);        // caller
txt ( t % 10);           // caller
}
```

```
txt (1000);          // caller
}

num = num % 1000;
if ( num >= 100)
{

txt (num/100);       // caller
txt (100);           // caller
}

num = num % 100;
if (num <= 20)

txt (num);           // caller
else
{

txt (num/10 * 10);   // caller
txt (num % 10);      // caller
}

getch ();
}

txt ( long num)      // callee
{

switch (num)
{

case 1 : printf (" One");
         break;

case 2 : printf (" Two");
         break;

case 3 : printf (" Three");
         break;
case 4 : printf (" Four");  break;
case 5 : printf (" Five");
         break;

case 6 : printf (" Six");
         break;

case 7 : printf (" Seven");
         break;

case 8 : printf (" Eight");
         break;

case 9 : printf (" Nine");
         break;

case 10 : printf (" Ten");
          break;
```

```
case 11: printf("Eleven");
         break;

case 12: printf("Twelve");
         break;

case 13: printf("Thirteen");
         break;

case 14: printf("Fourteen");
         break;

case 15: printf("Fifteen");
         break;

case 16: printf("Sixteen");
         break;

case 17: printf("Seventeen");
         break;

case 18: printf("Eighteen");
         break;

case 19: printf("Ninteen");
         break;

case 20: printf("Twenty");
         break;

case 30: printf("Thirty");
         break;

case 40: printf("Fourty");
         break;

case 50: printf("Fifty");
         break;

case 60: printf("Sixty");
         break;

case 70: printf("Seventy");
         break;

case 80: printf("Eighty");
         break;

case 90: printf("Ninty");
         break;

case 100: printf("Hundred");
          break;

case 1000: printf("Thousand");
           break;

case 100000: printf("Lakh");
             break;
```

```
case 11: printf("Eleven");
         break;

case 12: printf("Twelve");
         break;

case 13: printf("Thirteen");
         break;

case 14: printf("Fourteen");
         break;

case 15: printf("Fifteen");
         break;

case 16: printf("Sixteen");
         break;

case 17: printf("Seventeen");
         break;

case 18: printf("Eighteen");
         break;

case 19: printf("Ninteen");
         break;

case 20: printf("Twenty");
         break;

case 30: printf("Thirty");
         break;

case 40: printf("Fourty");
         break;

case 50: printf("Fifty");
         break;

case 60: printf("Sixty");
         break;

case 70: printf("Seventy");
         break;

case 80: printf("Eighty");
         break;

case 90: printf("Ninty");
         break;

case 100: printf("Hundred");
          break;

case 1000: printf("Thousand");
           break;

case 100000: printf("Lakh");
             break;
```

```
case 10000000: printf("Crore");
                break;
    }
}
```

o/p →

```
Enter the number : 456789456

Fourty Five Crore Sixty Seven Lak
Eighty Nine Thousand Four Hundred
Fifty Six.
```

Ex. Write a program to accept a decima no. Print its equivalent octal.

```
#include <math.h>
void octal (int);          //declaration
void main ()
{
  int num;
clrscr ();

printf ("Enter any number :");
scanf ("%d", &num);

octal (num);               //caller
getch ();
}

void octal (int num)       //callee
{

int p=0;
long int oct=0;

while (num >0)
{
oct = oct + ((num%8)* pow(10,p));
num = num/8;
p++;
}

printf ("Its equivalent octal is :
                        %ld", oct);

}
```

o/p →

```
Enter any number : 458
Its equivalent octal is : 712
```

Ex. Write a program to implement the febonex xeries.

```
void febo (int);           //declaration
void main ()
```

```
{
int num;
clrscr();
printf("Enter any number:");
scanf("%d", &num);
febo(num);          // caller
getch();
}

void febo(int num)   // callee
{
int t, t1=0, t2=1;

printf("Febonex Series Upto %d is:
                    \n", num);

if(num==0)
printf("%d", t1);

else if(num==1);

printf("%d %d", t1, t2);
while((t=t1+t2)<=num)
{

printf("%d", t);
t1=t2;
t2=t;
}
}
```

o/p →

```
Enter any number : 56
Febonex Series Upto 56 is:
0 1 1 2 3 5 8 13 21 34 55
```

3. Function with argument and return value :-

In this category there is a communication on both sides (that is from the caller function as well as from the callee function). A return value will be sended back to the caller function.

Ex. Write a program to accept a number. Findout how many numbers of 1's is present in the equivalent binary of that given number.

```
void main()
{
```

```
int countones (int);    // declaration
int num, k;
clrscr ();
printf ("Enter any number : ");
scanf ("%d", &num);

k = countones (num);    // caller

printf ("number of 1 for %d is: %d",
                        num, k);
getch ();
}

int countones (int n)   // callee
{
int cnt = 0;
while (n)
{
 if (n % 2)
 cnt ++;

 n = n/2 ;
}

return cnt;
}
```

o/p →

```
Enter any number : 45
number of 1 for 45 is : 4
  (because binary of 45 is 101101)
```

Ex Write a program to accept a number and findout its factorial.

```
long fact (int);        // declaration
void main ()
{

int num;
long t;
clrscr ();
printf ("Enter the number : ");
scanf ("%d", &num);

t = fact (num);         // caller

printf ("Factorial of %d" is : %ld",
                        num, t);

getch ();
}
long fact (int num)     // callee
```

```
{
long f=1;
if (num==0 || num==1)
return 1;
for (; num>=2; f=f*num--);
return f;
}
```

o/p →
Enter the number : 5
Factorial of 5 is : 120

Note -

The difference between exit and return statement is, exit come out from the program and return come out from the sub-module and back to caller.

In the above program we can call the fact function without temporary variable —

```
printf("Factorial for %d is: %ld",
                    n, fact(n));
```

Write a program to accept two nos as base and power. Findout the power value.

```
long power (int, int);  // declaration
void main()
{
 int b, e;
 long k;
 clrscr();
 printf("Enter base and Power :");
 scanf("%d %d", &b, &e);

 k = power (b, e);  // caller

 printf("Power Value is : %ld", k);
 getch();
}

long power (int b, int e)  // callee
{

int p=1;
if (e==0)
return 1;
while (e)
{
  p = p * b;
  e--;
```

```
    }
    return p;
    }
```

o/p →

```
Enter base and power : 3 4
Power value is : 81
```

Ex. Write a program to display 1 to 10 natural numbers using function's third category.

```
long natural (int);  // declaration

void main ()
{
  int i, n = 10;

  clrscr ();

  for ( i = 1; i <= n; i++)

  printf ("%d", natural (i));  // caller
  getch ();
  }

long natural (int i)  // callee
{
  return i;
}
```

o/p →

1 2 3 4 5 6 7 8 9 10

In this program nearly 10 times function is calling and the value is returning to the function caller. It takes lot of borden when the number of calls are more. So series of example patterns are not possible with return statement, since return statement can return only one value.

	Decleration	Calling
1.	int sum(int, int);	→ i1 = sum(i2, i3);
2.	void sum(int, int);	→ sum(i1, i2);
3.	float sum(float, float);	→ f1 = sum(f2, f3);
4.	char sum(char, int, float);	→ c1 = sum(c2, i2, f2);
5.	void kiran (void);	→ kiran ( );

6. int kiran (void); i1 = kiran ();

7. void abc (int, char); abc (i1, c1);

8. float kiran (char, int, float, float); f1 = kiran (c1, i1, f2, f3);

Different way of function calls and return values :-

1. return (55); // valid
2. return (60); // valid
3. return (x+y+z); // valid
4. calls (a, 25, d); // valid
5. callsum (10+2, 25%3, d); // valid
6. callsum (a, callsum (25, 10, 4), d); // valid
7. callsum (a, 25, d) * callsum (a, 25, d) + 23; // valid.

Returning more than one value :-

First method :-

```
void main ()
{
int a = 10, b = 20, c = 30;
int x, p;

x, p = sumprod (a, b, c);   // invalid
        (left side should be only 1 variable)
printf ("%d %d", x, p);
}

sumprod (int x, int y, int z)
{
 int xx, pp;
 xx = x+y+z;
 pp = x*y*z;

return (xx, pp);   // not possible.
}
```

this program will execute but it will give wrong output.

Second method :-

```
void main ()
{
int a = 10, b = 20, c = 30;
int x, p;
x = sumprod (a, b, c);
p = sumprod (a, b, c);
```

```
printf ("%d %d", s, p);
}

sumprod (int x, int y, int z)
{
   int ss, pp;
   ss = x+y+z;
   pp = x*y*z;

   return (ss);
   return (pp);
}
```

this program will execute but it also will not give the correct output. it will return only one value.

Third Method -:

```
void main ()
{

int a = 10, b = 20, c = 30;
 int s, p;

s = sumprod (a, b, c, 1);
p = sumprod (a, b, c, 2);
printf ("%d %d", s, p);
}

sumprod (int x, int y, int z, int code)
{

int ss, pp;
ss = x+y+z;
pp = x*y*z;

if (code == 1)
 return (ss);
else
 return (pp);
}
```

it is the correct way of writting the program.

o/p → 60 6000

Note -:

In the above program the return statement can also be written like as -

```
return (code == 1 ? x+y+z : x*y*z);
```

but

```
code != 1 ? return (x+y+z) : return (x*y*z);
```

is invalid

Nested Function :-

A nested function takes place if more than one function is called as a single line then the principle is called as Nested function.

It is mostly usefull when a series of arithmatical transaction fall in a single formula.

The result of particular function need to be dependent on the operation carried out by another function.

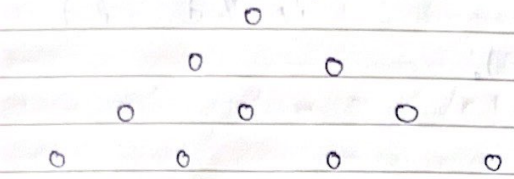
Ex.

```
#include <stdio.h>
void main()
{

int add (int, int);
int mult (int, int);
int sub (int, int);
float div (int, int);

int a, b, c, d, e, f, g, h, R;

R = sub(mult(add(a,b), add(c,d)),
        add(div(f,g), h));
printf("\n The result of expression
        is %d", R);
getch();
}

float div(int x, int y)
{
return ((float)x/(float)y);
}

int mult(int x, int y)
{
return (x*y);
}

int add(int x, int y)
{

return (x+y);
}

int sub(int x, int y)
{
return (x-y);
}
```

o/p →

The result of expression is 39.

Ex: write a program to print

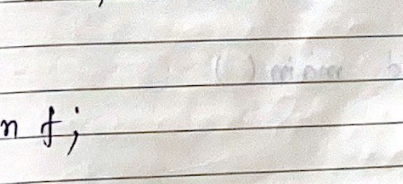


```
# include <stdio.h>
# include <conio.h>
void main()
{
unsigned long fact(int);
int n, ncr, i, j, k;
clrscr();
printf("\n Enter the number:");
scanf("%d", &n);
k = 3 * n;

for(i=0; i<n; i++)
{

printf("%*c", k, 32);
for(j=0; j<=i; j++)
{

ncr = fact(i)/(fact(j)*fact(i-j));
printf("%-6d", ncr);
}
```

```
printf("\n");
}
getch();
}

unsigned long fact(int x)
{
 int i, x=10;
unsigned long fact(int x)
{
int i, x=10;
unsigned long f=1;

for(i=1; i<=x; i++)
{

 f = f * i;
}

return f;
}
```

o/p →

```
Enter the number : 4
```



Ex: Write a program to find the sum of the following series.

$1 + x^2/2! + x^4/4! + \cdots$

```
# include <stdio.h>
# include <conio.h>
# include <math.h>

unsigned long fact (int x)
{
 if (x<=1)

return 1;
else
return x * fact(x-1);
}

void main()
{

int x, n, p;
float sum=0;
clrscr();

printf("Enter x & n values:");
scanf("%d %d", &x, &n);

for(p=0; p<=n; p=p+2)
{
 sum = sum + (pow(x, p)/fact(p));
}

printf("Sum is %.3f", sum);
getch();
}
```

o/p →

Enter x & n values: 3 5

sum is 8.875

Ex.

```
void main()
{
 clrscr();
printf("Genious %d", fun(123));
getch();
 }

fun (int n)
{

return (printf("%d", n);
}
```

o/p → 123Genious 3

Ex. Write a program to display all the numbers in binary format of 16-bits from 0 to 32767.

```
#include <stdio.h>
void showbits(int);
void main()
{
  int j;
clrscr();
for(j=0; j<=32767; j++)
{
  if (j==-32768)
break;

printf("\n Decimal %d is same
        as binary ", j);

showbits(j);
 }
getch();
 }

void showbits(int n)
{

int i, k, andmask;
for(i=15; i>=0; i--)
{
andmask = 1 << i;
k = n & andmask;

k==0 ? printf("0"): printf("1");
}
}

Ex int main()
{
  unsigned int num;
int j;
```

Function calls -:

There are two types of function calls -

1. Call by value.
2. Call by address.

A function call can be takes place either by value or by address. By default function call always will be takes place by value.

Call by value -:

Sending the value of a variable from the caller to the callee is called as call by value. If any changes made in the subfunction(callee), it will not affect to the caller function variable values.

Ex Write a program to accept two numbers and swap them.

```
void swap(int, int);
void main()
{
 int a, b;
clrscr();
printf("Enter two numbers:");
scanf("%d %d", &a, &b);
swap(a, b);

getch();
}
void swap(int a, int b)
{

b = (a+b) - (a=b);

printf("After swapping in subfunction:");

printf("%d %d", a, b);
}
```

o/p →

Enter two numbers: 44 38

After swapping in subfunction: 38 44

Note -:

To swap two numbers a, b with each other we can use this logic also -

```
a^1 = b^1 = a^1 = b ;
```

Ex: Write a program to ~~print~~ accept two numbers. Display all the amstrong number between these two numbers.

First Method:

```
void amstrong (int, int);
void main()
{
  int a, b;
clrscr();
printf("Enter two numbers:");
scanf("%d %d", &a, &b);
amstrong (a,b);
getch();
}

void amstrong (int x, int y)
{

int sum, temp, r, z;
while (x <= y)
{

z = x;
temp = x;
sum = 0;
while(z)
{

r = z % 10;
sum = sum + (r*r*r);
z = z/10;
}

if ( sum == temp)
printf("\n Amstrog : %d", sum);
x++;
   }

}
```

o/p →

```
Enter two Numbers : 1   500

Amstrong : 1
Amstrong : 153
Amstrong : 370
Amstrong : 371
Amstrong : 407
```

Second Method:

Recursion –:

A function which is called by itself is called as recursion.

Recursion is a beautiful process but in implementation of recursion, it takes lot of borden on CPU (operating system) Function execution always on CPU (operating system)

When implementing recursion there is no need of looping process since recursion itself acts as an looping. To make the condition dis-satisfy either use 'if' or ternary operator.

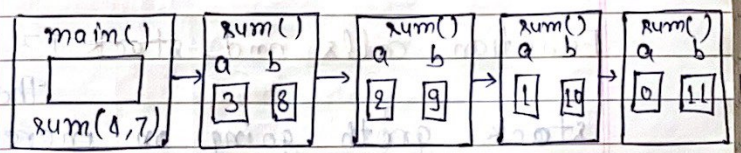
Recursion means code reusability or code repetation.

Ex: write a program to find sum of two numbers with the help of recursion concept.

```
int sum(int, int);
void main()
{
int a, b;
clrscr();
printf("Enter two numbers:");
scanf("%d %d", &a, &b);
printf("Sum is: %d", sum(a, b));

getch();
}

int sum(int a, int b)
{
if (a==0)
return b;
else
sum(--a, ++b);
}
```

O/p → Enter two numbers: 4 7
Sum is: 11

Execution of this program is as-

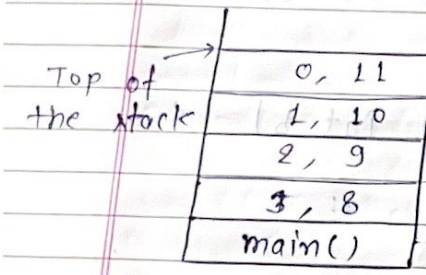


The recursion internal deceplin works on the stack on the principle of Last in First out (LIFO)

In the above program user accepted

a = 4 , b = 7

so the implementation of stack will be as -



In this program it nearly takes 4 times stack incrementing i.e. every function call will takes place on stack deciplene, as we know that any function works on operating system under the decipline of stack.

Function calls and stack (3 of 5)

The stack groth going on incremented when no. of function call will be more.

So whenever recursion is neccessary only implement the procedure. Unneccessary don't use the recursion concept otherwise the performance of the system will be degraded.

Ex. Write a program to implement fibonacci series with the help of recursion.

```
void main()
{
 long fibo(int);
int n, r;
clrscr();
printf("Enter the size of r:");
scanf("%d", &r);

for(n=0; n<r; n++)

printf("%ld", fibo(n));
getch();
}

long fibo(int n)
{
 long fib;
if(n==0)
 fib=0;
```

```
else
if (n == 1)
  fib = 1;
else
fib = fibo(n-1) + fibo(n-2);
return fib
}
```

o/p →

Enter the size of r: 7

0 1 1 2 3 5 8

In this program if my many number n input is less then performance will not degraded. if my number input is nearly 40 or 45, the performance of the system will be degraded and output will take some time to execute the process

Function calls and stack (3 of 5):

The stack is shown below -

Local variables of function
Arguments to function
user-defined function
Local variables of main
Arguments to main
Function main

STACK.

Function calls and stack (4 of 5):

Ex. Write a program to find the sum of two integers.

```
void fnsumprint(int ivalue1, int ivalue2);

int main(int argc, char **argv)
{

int inumber1=10, inumber2=20;
clrscr();
printf("Enter two numbers: ");
sconf("%d %d", &inumber1, &inumber2);
```

```
fnsumprint(inumber1, inumber2);
getch();
return 0;
}

void fnsumprint(int ivalue1, int ivalu
{

int iresult;
iresult = ivalue1 + ivalue2;

printf("Sum is : %d", iresult);
}
```

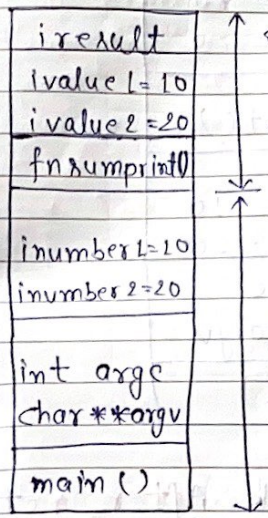
o/p →
Enter two numbers : 55 8
Sum is : 63

its internal processing will be like this –

Program Stack when executing main.

Local variables	iNumber 1 = 10
	iNumber 2 = 20
Formal parameter of main.	int argc
	char ** argv
	main()

2. Program Stack when executing fnsumprint()



ivalue1 and ivalue2 are copies of inumber1 and inumber2.

3. Program stack when executing iresult = ivalue1 + ivalue2;

(Code in fnsumprint act upon copies ... inumber1 and inumber2 only)

iresult = 30
ivalue 1 = 10 ivalue 2 = 20
fnsumprint()
inumber 1 = 10 inumber 2 = 20
int argc char ** argv
main()

Program Stack After returning from fnsumprint() back to main.

iresult = 30
~~ivalue 1 = 10~~ DATA TO BE INVALIDATED ~~ivalue 2 = 20~~
fnsumprint()
inumber 1 = 10 inumber 2 = 20
int argc char ** argv
main()

Note -

When a function is executing a runtime stack will be takes place and the variables will be takes place for the function entry and exit at a time only one function call can be takes place. Due to this the same memory which is used for another function that will be takes place for the next function call.

Scope -:

The scope of a variable refers to that portion of the program where the variables can access. The scope indicates they are accessible in some portion of the program and in the other portion they are not accessible.

When a variable is required in a program, it can be declared as local variables and global variables.

1. Local Variables -:

Local variables are declared inside a function. Their scope is only within the function in which they are declared. These variables can not been access outside the function. Local variables exists only till the function.

Initial values of local variables are garbage values.

Note.-

Local variable initial value is not a garbage value. due to some storage class specifier the initial value is garbage.

2. Global variables -:

The variables that are declared outside all the functions are called as global variables. These variables can be accessed by all the functions.

The global variable will exists for entire life cycle of the program.

By default the initial value of gor global variables is zero.

Note -

When we compare local variables and global variables, local scope is best rather than global scope since global variables are present throught the program, due to this wastage of memory. Due to global variables it is also difficult to debug the program.

Ex

```
main ()
{
  int i = 5;
sample ();
sample ();
sample ();
}

sample ()
{
printf ("%d", i);
  i++;
}
```

O/p → Error

Life -:

The life of a variable reffers how long a variable will exist or retain its value

Ex.
```
void main()
{                                   // functional scope
int a=10;
clrscr();
printf("%d\n", a);
{                                   // Anonymous scope
 int b;
 b = 20;
 a++;
 b++;
printf("%d\n", a);
printf("%d\n", b);
}
printf("%d\n", a);
getch();
}
```

o/p →
```
10
11
21
11
```

In this program 2 scopes are present. One is functional scope and another one is anonymous scope. Anonymous scope is the scope which does not have any name.

In this program the a variable can exists within the functional scope - ~~when we using~~ and anonymous scope. but b variable can exist only in anonymous scope. when we using outside the anonymous the life already lost so immediately raises an error.

Storage classes -:

The storage classes specifies what type of storage class the initial values, the memory locations and the scope and the life of the variables, define by storage classes.

According to the storage classes the scope and life will be change.

In 'C' language there are 4 types of storage classes -

1. Automatic Storage classes.
2. Static Storage classes.
3. Extern Storage classes.
4. Register Storage classes.

The storage classes specifiers are -

1. auto
2. static
3. extern
4. register
5. typedef

Program Stack and Heap -:

During Execution of a program the storage of program and data is as follows -

- The executable code is stored into the code / Text document.
- The global variables are stored into data segment.
- The heap memory is used for dynamic memory allocation.
- The local variables are stored into the stack.

Heap -:

A section of memory within the user job area that provides capability for dynamic allocation.

Code segment	← Executable code
Data segment	← Global Variables
Heap	← Dynamic memory
Stack	← Local Variables

The code segment and the data segment can be define as a permanent memory allocation (PMA) Depends on the storage classes the memory location will be changes.

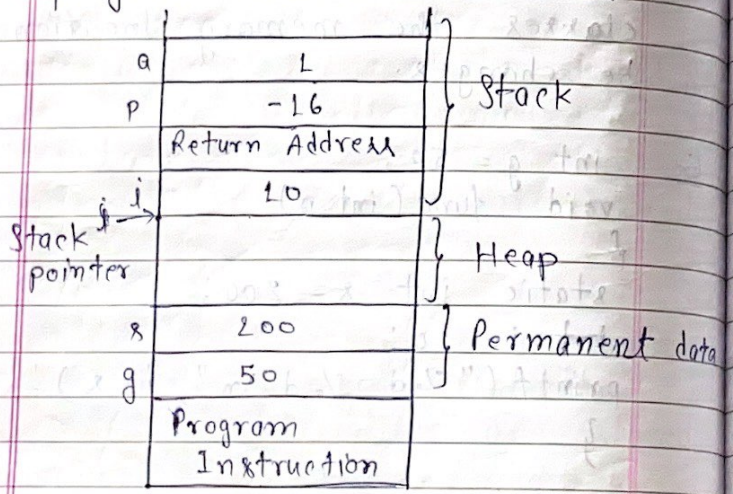
Ex

```
int g = 50;
void func(int p)
{
static int s = 200;
int i = 10;
printf("%d %d \n", i, s);
}
```

```
void main()
{
int a = 1;
clrscr();
printf("%d \n", a);
printf("%d \n", g);

func(-16);
printf("OVER \n");
getch();
}
```

o/p → 1
50
10 200
OVER

stack implementation for this program will be like this -



Storage Classes	Specifier (keyword)	Initial Value	Memory Location	Scope	Life
Automatic	auto	Garbage	Stack	Within the body	Entire body
Static	static	Zero	PMA	Within the function	Entire Program
External (global)	extern	Zero	PMA	All the function	Entire Program
Register	register	Garbage	Register	Within the body	Entire Body

Note -

By default any variable storage class is automatic.

1. auto -:

Ex
```
main()
{
auto int a = 5;
printf("%d", a);

getch();
}
```

o/p → 5

```
main()
{
{
auto int a=5;
printf("%d",a);
}
getch();
}
```

o/p → 5

In these two patterns the second session of the programming is more illegate to write without wastage of the memory blocks.

Ex
```
main()
{
int i;
clrscr();
for(i=1; i<=10; i++)
printf("%d %d \n", i, fun(i));
getch();
}

fun(int x)
{
auto int x;
x = 100;
x += x;
return x;
}
```

O/p →

1	101
2	102
3	103
:	:
9	109
10	110

In this program every variable is automatic storage class. Due to this the life will be present within the body.

When we calling the subfunction the x local variable will be initializing first time with garbage value and then x = 100. Once the body is been closed the value of the x variable will be lost since life is only entire body. Once again the function call x will be allocated new memory allocation with a value of 100. This process continues untill the end of the function.

2. Static -:

Ex
```
main()
{
static int i;
clrscr();
printf("%d", i);
getch();
}
```

O/p → 0

Ex.

```
main()
{
clrscr();
stat();
stat();
stat();
getch();
}

stat()
{
static int i;
printf("%d", i);
i++;
}
```

o/p → 0 1 2

Here static variable will be initialize only one time. automatic variables every time will initialize.

Ex.

```
main()
{
clrscr();
stat();
stat();
stat();
getch();
}

stat()
{
static int i;
printf("%d %u\n", i, &i);
i++;
}
```

o/p →

```
0   858
1   858
2   858
```

Here &i displays the address of 'i' variable.

Ex

```
main()
{
int i;
clrscr();
for(i=1; i<=10; i++)

printf("%d %d\n", i, fun(i));
getch();
}

fun(int i)
{
```

```
static int x = 100;
  x += i ;
return x;
}
```

o/p →

1	101
2	103
3	106
4	110
5	115
6	121
7	128
8	136
9	145
10	155

Note -

When static variable is initializing rather than zero, it has to be initialize at the time of declaration only or else it will not affect.

Ex.

```
main()
{
 int i;
clrscr();
for (i=0 ; i<5 ; i++)
{
static int a=0;
int b=0;
a++;
b++;
printf("a = %d  b = %d\n", a, b);
}
getch();

return 0;
}
```

o/p →

a = 1	b = 1
a = 2	b = 1
a = 3	b = 1
a = 4	b = 1
a = 5	b = 1

Ex.

```
int r();
int main()
{
 clrscr();
for (r(); r(); r())

printf("%d", r());
getch();

return 0;
}
```

```
int r()
{
int static num = 7;
return num--;
}
```

o/p → 5 2

Ex.

```
main()
{
auto int a;
register r;
static s;
extern e;
typedef t;

clrscr();

printf("\n%d", sizeof e);
printf("\n%d", sizeof a);
printf("\n%d", sizeof r);
printf("\n%d", sizeof s);
printf("\n%d", sizeof(t));
getch();
return 0;
}
```

o/p → 2 2 2 2 2

Note -

Here a, r, s and e are variables and 't' is an alias name so 't' is not a variable.

3. External (Global) -:

The variables which can be define outside of all functions, such types of variables are called as Global Storage Classes.

Ex.

```
int i;
main()
{
int i = 10;
printf("%d", i);
getch();
}
```

o/p → 10

Ex.

```
int i;
main()
{
printf("%d", i);
getch();
}
```

O/p → 0

Some important points :-

1. When the external (Global) variable and local variable have the same name then always the priority will be taken place for the local variable.

2. When local variable is not present then there is a chance of global variable.

3. When any local variable wants to convert into global variable then ~~there is no necessary to define~~ we can use the keyword of extern to convert into global variable.

4. If a variable is defined outside of all the function then there is no necessary to define the 'extern' keyword.

Ex:
```
main()
{
extern int i;
printf("%d", i);
}
```

When we compile this program there will not be short of any errors but when we run this program it raises on error as linker error, since extern variable always checks at the time of execution.

So we must should be define the variable globally for the purpose of memory allocation.

Ex
```
int i;                    // declaration
                          definition
main()
{
  extern int i;           // declaration.
printf("%d", i);
}
```

O/p → 0

In this program there are 2 types of statements are present one is declaration and other is definition. Declaration will be taken place with the keyword of extern. Definition will be taken place by defining outside the

main (). Generally declaration never contains any memory. Definition allocates the memory.

Ex. int i; // definition
main (-)
{

extern int i = 10; // declaration.
printf ("%d", i);
}

o/p → Error.

Note –

1. The extern int i = 10 is only the declaration at the time of declaration. We can-not initialize the external variable.

2. The difference between declaration and definition is, in the definition of a variable space is reserved for the variable and some initial value is given to it. Whereas in declaration only identifies the type of the variable for a function.

Thus definition is the place where the variable is created are assigned storage where as the declaration refers to the places where the nature of the variable is stored but not storage is allocated.

Ex. int j;
main ()
{
extern int i;
j = 12;
printf ("%d", j);
getch ();
}

o/p → 12

Ex. int g;
void xyz ()
{
++g;
printf ("%d", g);
}
void main ()
{
clrscr ();
++g;

```
    xyz ();
    getch ();
    }
```

o/p → 2

Ex.
```
void xyz ()
{
 ++g;
}
int g;

void abc ()
{
 ++g;
}

void main ()
{
 clrscr ();
 abc ();
 xyz ();
getch ();
}
```

o/p → Error.

Since compilation starts from top to bottom.

When ++g is read by the compiler, it immediately raises an error, since it is not declared.

Ex.
```
void kiran ()
{
 extern int g;
 ++g;
}

int g;
void xyz ()
{
 ++g;
}

void abc ()
{
 ++g;
}

void main ()
{
 clrscr ();
 ++g;
 xyz ();
 abc ();
 kiran ();
```

```
printf ("%d", g);
getch ();
}
```

o/p → 4

Ex.

```
int i;
main()
{
extern int i;
clrscr();
printf ("%d", i);
fun();
getch();
}
fun()
{
printf("%d", i);
}
```

o/p → 0 0

Ex:

```
int i;
main()
{
extern int i;
int i = 56;
clrscr();
printf("%d", i);
fun();
getch();
}
fun()
{
printf("%d", i);
}
```

o/p → 56 56

Ex

```
void rec (int i)
{
printf ("\n%d", i);
if( i<=2)
rec (i+1);
printf("\n%d", i);
}
void main()
{
clrscr();
rec(1);
getch();
}
```

o/p → 1 2 3
3 2 1

Ex

```
void abc()
{
  int i;
static int x;
  i = ++x;

printf("\n %d %d", i, x);
 if (i<=2)
 abc();

printf("\n %d %d", i, x);
}

void main()
{
 clrscr();
 abc();
 abc();
 getch();
}
```

o/p →

1	1
2	2
3	3
3	3
2	3
1	3
4	4
4	4

Ex

```
int g;

void abc()
{
 int a1;
static int ax;
a1 = ++ax;
++g;

printf("\n %d %d %d", a1, ax, g);

if(a1<=2)
abc();
printf("\n %d %d %d", a1, ax, g);
}

void kiran()
{

int i;
static int x;
i = ++x;
++g;

printf("\n %d %d %d", i, x, g);
 abc();
if (i<=2)

kiran();
```

Ex:

```
#include <stdio.h>
void main()
{
static int a = 25;
void cdecl conv1();
void pascal conv2();
conv1(a);
conv2(a);
getch();
}

void cdecl conv1(int a, int b)
{
printf("\n %d %d", a, b);
}

void pascal conv2(int a, int b)
{
printf("\n %d %d", a, b);
}
```

O/P →

```
25  0
0  25
```

This program will execute as per the 'c' compiler, but not as per the 'c++' compiler. Since 'c++' is the strongly type checking and 'c' is an loosely type checking.

Ex:

```
void cdecl fun1(int, int);
void pascal fun2(int, int);

void main()
{
int a=5, b=5;
clrscr();

fun1(++a, a++);
fun2(b++, ++b);
getch();
}

void cdecl fun1(int p, int q)
{
printf("cdecl: %d %d\n", p, q);
}

void pascal fun2(int p, int q)
{
printf("pascal: %d %d", p, q);
}
```

```
printf("\n %d %d %d", i, s, g);
}

void main()
{
  clrscr();
getch();

kiran();
getch();
}
```

O/P →

1	1	1
1	1	2
2	2	3
3	3	4
3	3	4
2	3	4
1	3	4
2	2	5
4	4	6
4	4	6
3	3	7
5	5	8
5	5	8
3	3	8
2	3	8
1	3	8

Ex.

```
void fun(auto int _)
{
printf("%d", _);
}

main()
{
 fun(23);
return 0;
}
```

O/p → Error.

Explanation –

Since function execution on run time STACK that is CPU, storage classes will be taken place based on the RAM.

If we use 'register' instead of 'auto' then it will execute successfully and output will be 23.

Here Program executes since register is a part of CPU. So for the parameter we can define the registers

Ex:

```
#include <stdio.h>
void main()
{
static int a = 25;
void cdecl conv1();
void pascal conv2();
conv1(a);
conv2(a);
getch();
}

void cdecl conv1(int a, int b)
{
printf("\n %d %d", a, b);
}

void pascal conv2(int a, int b)
{
printf("\n %d %d", a, b);
}
```

o/p →

```
25  0
0   25
```

This program will execute as per the 'c' compiler, but not as per the 'c++' compiler. Since 'c++' is the strongly type checking and 'c' is an loosely type checking.

Ex

```
void cdecl fun1(int, int);
void pascal fun2(int, int);

void main()
{
int a=5, b=5;
clrscr();

fun1(++a, a++);
fun2(b++, ++b);
getch();
}

void cdecl fun1(int p, int q)
{
printf("cdecl: %d %d\n", p, q);
}

void pascal fun2(int p, int q)
{
printf("pascal: %d %d", p, q);
}
```

o/p →

cdecl : 7 5
pascal : 5 7

Note –

In 'C' style execution always from ~~right~~ to left and in 'pascal' style it starts from left to right.

4. Register –:

Register is a seperate storage class. The value of the variables are stored in CPU registers.

If the variable is used at many places in a program, it is better to declare variables as registers. As the value stored in CPU registers can always be access faster than the one which is stored in the memory.

Generally in CPU there will be less no. of registers available. A normal CPU has minimum 14 registers, and each register size is 2 bytes.

When registers are implementing which are the variable access less memory of bytes define such type of datatype. CPU registers are in microcomputers usually 16 bit registers. They can-not hold a float, double or long. Always better to define the variables of automatic storage class rather than registers.

Ex

```
void main()
{
register float a;
register float b;
register float c;
}
```

We can-not give any gaurenty that all the data will storing registers, since registers are of very less capacity. By default they send back to the memory by defining automatic storage class.

Ex.

```
main()
{
int j;
for(j=1; j<255; j++)
{

j++;
}
}
```

```
main()
{
register int j;
for(j=1; j<255; j++)
{ ①
j++;
}
}
```

in the second session memory is accessing nearly 766 times! So to access memory very fastly defining the registers is always best. At this time of situations registers need to be define.

Note —

The external storage class is not best to use comparing with other storage classes.

Ex. What does the hexa number E78 in Radix 7

First of all convert Hexa number E78 into its equivalent decimal-

$E78 = (E*16^2) + (7*16^1) + (8*16^0)$

$= (14*16^2) + (112) + (8)$

$= 3584 + 112 + 8$

$= 3704$

Now convert the decimal number 3704 into Radix 7 (base 7)

7 | 3704		R
7 | 529		1
7 | 75		4
7 | 10		5
7 | 1		3
	0	1

i.e. $(3704)_{10} = (13541)_7$

so $(E78)_{16} = (13541)_7$

Ex.

```
#include <stdio.h>
int i;
int fun();
int main()
{
 while(i)          // dir- satisfied
{
 fun();
main();
}
```

```
printf("HELLO \n");
return 0;
}

int fun()
{
printf("HI");
}
```

o/p → HELLO

Note -

In the above program i is a global variable which initial value is 0 (by default) so while condition will be dis-satisfy.

```
int main()
{
  int a, b;
a = sumdig(123);
b = sumdig(123);
printf("%d %d \n", a, b);
return 0;
}
int sumdig(int n)
{
int s, d;
if (n!=0)
{
  d = n%10;
  n = n/10;

s = d + sumdig(n);      //recursion
}
else
return 0;
return s;
}
```

o/p → 6 6

In this example the function is calling with a value of 123 from the main, at the time of the statement s = d + sumdig(n);

so 3 + sumdig(12);
once again the call will take place 2 + sumdig(1); and once again 1 + sumdig(0);

1 + sumdig(0)
2 + sumdig(1)
3 + sumdig(12)
main()

When the condition dis-satisfied the else statement is executing. It returns 0 for the last function calls. i.e. sumdig(0).

Ex:
```
int main()
{
 int fun(int);
 int i = fun(10);
 clrscr();
 printf("%d\n", --i);
 return 0;
 }

int fun(int i)
{
 return (i++);
}
```

o/p → 9

```
#include <stdio.h>
int n, R;
 main()
{
 R = 0;
 clrscr();
printf("Enter any number:");
 scanf("%d", &R);
printf("\n%d %d", fun(n), R);
 getch();
}

int fun(int n)
{
 if (n>3) return
 R = 5;
 R = 6;
 return (1);
 }
```

o/p →
Enter any number: 3
1 0
Enter any number: 8
5 0

Ex:
```
#include<stdio.h>
main()
{
 int x = 3;
clrscr();
 f(x);
printf("MAIN");
 getch();
}
f(int n)
```

```
{
printf("F");
if(n!=0)
f(n-1);
}
```

o/p → FFFFMAIN

Ex.

```
#include <stdio.h>
void main();
double dbl=20.4530, d=4.5710;
double dblvar3;

void main(void)
{
 double dbln(void);
clrscr();
printf dblvar3=dbln();

printf("%.2f \t %.2f \t %.2f \n",
        dbl, d, dblvar3);

getch();
}
dbl
double dbln(void)
{
double dbln(void)
double dblvar3;
dbl = dblvar3=4.5;
return (dbl + d + dblvar3);
}
```

o/p → 4.50 4.57 13.57

Ex.

```
int fun(int);
void main()
{

float k=3;
clrscr();
fun(k=fun(fun(k)));

printf("%.3f",k);
getch();
}

fun(int i)
{
 i++;
return i;
}
```

o/p → 5.000

Ques. How to define a function that can recieve any number of arguments?

Ans. int _cdecl printf(const char *format,...);

Ex.

```
int main()
{
 float n = 1.54;
 clrscr();
printf("%f %f\n", ceil(n), floor(n));
 getch();
}
```

o/p → 2.000000
1.500000

Note -

ceil means ~~upper~~ value.
floor means down value.

Ex.

```
int get();
int main()
{
 const int x = get();
clrscr();
printf("%d", x);
getch();
return 0;
}
int get()
{
 return 20;
}
```

o/p → 20

Ex.

```
#include <stdio.h>
void fun(int _)
{
 printf("%d", _);
}

main()
{
clrscr();
fun(100);
return 0;
}
```

o/p → 100

Ex.

```
void kir(int a, int b)
{
 printf("\n%d %d", a, b);
}
 void main()
{
 int i = 1;
kir(++i, i++);
kir(++i, ++i);
printf("\n%d", i);
}
```

o/p → 3 1
5 4
5

Ex:
```
void main()
{
int p=23, f=24;
clrscr();
packman(p,f);
printf("p=%d, f=%d", p, f);
getch();
}
packman(q,h)
int q,h;
{
q = q+q;
h = h+h;

return q;
return h;
}
```

o/p → p=23, f=24

Ex:
```
void main()
{
int x, num=1;
clrscr();
x = call(num);
printf("%d", x);
getch();
}

int call(int num)
{
static int x=0;
if(num<4)
{
x = x+num;
call(num+1);          //recursion
return x;
}
}
```

o/p → 6

Ex:
```
void main()
{
register a,b,sum;
scanf("%d %d", &a, &b);
sum = a+b;
printf("%d", sum);
}
```

o/p → Error.

Note - Address of a register variable can not be accessed.

Ex:
```
int call (register int, register int);
void main ()
{

int temp;
temp = call (5, 25);
printf ("%d", temp);
getch ();
}

int call (int register x; int register y)
{

static int num;
num = (~x + y + 1);
return num;
}
```

o/p → 20

Explanation –

$-6 + 25 + 1 = 20$

Ex:
```
main ()
{
extern int i;
i = 20;
printf ("%d", sizeof (i));
}
```

o/p → Error.

Explanation –

extern int i; is a declaration with no physical memory. It should contain a physical memory If we write like int i; then there will be no error because it contains physical memory of 2 bytes.

Ex:
```
main ()
{
int i = 4;
clrscr ();
if (i > 5)
printf ("HELLO");
else
f (1);
}

f (int j);
{
if (j > 0)
f (j - 1);
else
if (j == 0)
```

```
return;
printf ("HI");
}
```

o/p → HIHI

Ex

```
static int a = 6;
extern int a;
main()
{
printf("%d", a);
}
```

o/p → 6

Explanation –

The combination reffers always from top to bottom. In this program extern and static storage location will be the same and due to the compilation top to bottom, which is we define at the first statement that variable value only is accessible, remaining variable will not be taken place. You can define extern and static as outside the name but not automatic storage class.

Ex

```
main()
{
auto int i = 0;
printf("%d \n", i);
{
 int i = 2;
printf("%d \n", i);
{
 i += 1;
printf("%d \n", i);
}
printf("%d \n", i);
}
printf("%d \n", i);
}
printf("%d \n", i);

printf("%d \n", reset());
printf("%d \n", ret10());
printf("%d \n", reset());
printf("%d \n", ret10());
getch();
}

int reset()
{
 int j = 0;
return(j);
}
```

```
int ret10()
{
static int i=10;
i+=1;
return(i);
}
```

o/p → 0
2
3
3
0
0
11
0
12

Ex:
```
static int i=50;
int print(int i);
void main(void)
{
static int i=100;
clrscr();
while(print(i))
{
printf("%d\n", j);
i--;
}
getch();
}
int print(int x)
{
static int i=2;
return(i--);
}
```

o/p → 100 99

Ex:
```
int i=10;
main()
{
extern int i;
{
int i=20;
{
const volatile unsigned i=30;
clrscr();
printf("%d\n", j);
}
printf("%d\n", j);
}
printf("%d\n", j);
getch();
}
```

o/p → 30 20 10

Ex:
```
void main()
{
static int i=5;
if (--i)
{
main();
printf("%d", i);
}
}
```

ge o/p → 0000

Ex:
```
void main()
{
int k = ret(sizeof(float));
clrscr();
printf("\n Here value is %d", ++k);
}

int ret(int ret)
{
ret += 2.5;
return(ret);
}
```

o/p → Here value is 7.

Ex:
```
int x;
int modifyvalue()
{
return (x+=10);
}
int changevalue(int x)
{
return (x+=1);
}
void main()
{
int x=10;
clrscr();
x++;
changevalue(x);
x++;
modifyvalue();
printf("First Output: %d\n", x);
x++;
changevalue(x);
printf("Second Output: %d\n", x);
modifyvalue();
printf("Third Output: %d\n", x);
}
```

o/p → First Output: 12
Second Output: 13
Third Output: 13

Ex.

```
extern int a = 5;
main()
{
void fun();
printf("\n a = %d", a);
fun();
return 0;
}
int a;
void fun()
{

printf("\n in fun a = %d", a);
}
```

o/p → a = 5
in fun a = 5

Pug marks
judge a tige
and age –
weight and

Tigers can run with three legs in the air simultaneously!

```
extern int a = 5;
main()
{
void fun();
printf("\n a = %d", a);
fun();
return 0;
}
 int a;
 void fun()
 {

 printf("\n in fun a = %d", a);
 }
```

o/p → a = 5
in fun a = 5