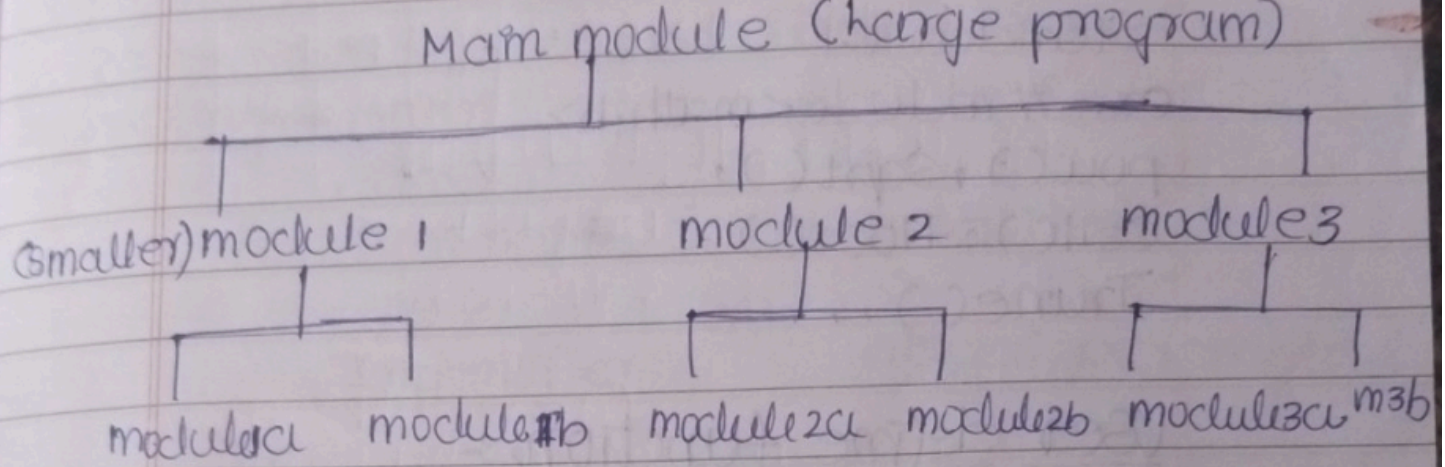
3. Functions

→ Function is a sub part of a program used to profrom sepecific task at is excuted individually.



Advantages of function :-

1. code reusability
2. once a function is created it can be be used for many times.
2. using functions we can impliment modular programming.
3. larger programs are divide into smaller programs.
4. Improve the readability of code.
5. debuging (error identification) of the code is easier.
6. using functions program implimentation become easy.

Types of functions :-

1. Built-in-Functions (or) predefined (or) Standard.
ex:- #include<math.h>
pow(), sqrt(),
ceil(), floor(),
True()....

2. User defined functions are one for which the compiler generates machine code at compile time.

User define function :-

→ User define functions are written by user
→ Every function has three parts
1. Function Declaration / Function prototype
2. Function call / calling Function
3. Function Defination / called function

1. function deblaration :-

→ This is declared before the main function

Syntax :-

returntype functionname(parameterlist);

- returntype → void, int, char, float
- functionname → Indentifier
- parameterlist → formal parameters

ex:- void add();
int add(int a, int b);
int add();

2. Function Call :-
→ This is declared in the main function
Syntax :-

function name (parameters list);
↓ ↓
Indentifier actual parameters

ex:-
add();
add(a,b);

3. Function definition :-
→ Function definition should be After the main function.
Syntax :-

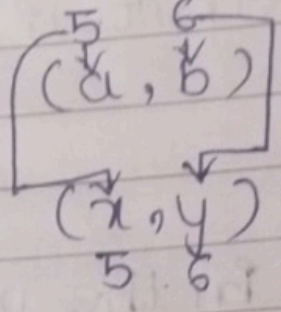
returntype functionname (parameterlist)
↓
formal Parameter
{
//actual logic
}

ex:- void add()
{
logic
}

ex:- int add()
{
//logic
return c;
}

→ No need of using same names

ex:-



→ That values copied into another

Actual parameter:-

→ The parameters specified in calling function are said to be actual parameter.

Formal parameter:-

→ The parameter specified in called function are said to be formal parameter

→ The value of actual parameter is always copied to formal parameter

→ Based on parameter and return type functions are classified into four types.

1. without parameters and without return value.
2. With parameters and without return value
3. Without parameters and with return value
4. With parameters and with return value.

Date
Page

Without parameter
- with return
- without return

With parameter
- with return
- without return

→ It terminate only in main function()
→ After function defmition it will go to function call.

1. Without parameters & without return value

```
#include<stdio.h>
void add(); // Function Declaration
int main()
{
   add(); // Function call
}
void add()  // Function definition
{
   int a, b, c;
   printf("Enter a, b values");
   Scanf("%d %d", &a, &b);
   c=a+b;
   printf("c=%d", c);
}
```

2. without paramaters & with return value

```
# include <stdio.h>
  int add();
int main()
{
  int res;
  res = add();
  printf("res = %d", res);
}
int add()
{
  int a, b, c;
  printf("enter a, b values");
  scanf("%d %d", &a, &b);
  c = a+b;
  return c;
}
```

3. With parameters & withaitreturn valu.

```
# include <stdio.h>
void add(int a, int b); // Function declarat
int main()
{
int a, b;
printf("Enter a, b values");
scanf("%d %d", &a, &b); // 10 20
```

```
add(a,b);  // Function call
   └→ actual parameters
}
void add(int a, int b)  // Function definition
          └→ Formal parameters
{
    int c;
    c = a+b;
    printf("c = %d\n", c);
}
```

4. with parameters and with return value

```
#include<stdio.h>
int add(int a, int b);
int main()
{
  int a,b, result;
  printf("enter a&b values");
  scanf("%d %d", &a, &b);
  res = add(a,b);
  printf("res = %d\n", res);
}
int add(int a, int b)
{
return(a+b);
}
```

Q. write a c program to find the factorial of a given number.

```
#include<stdio.h>
void fact();
int main()
{
    fact();
}
void fact()
{
    int n, i, fact=1;
    printf("enter n value");
    scanf("%d", &n);
    for(i=1; i<=n; i++)
    {
        fact = fact * i;
        printf("%d", fact);
    }
```

Q.

```
#include <stdio.h>
int fact();
int main()
{
    int fact;
    fact=fact();
    printf("fact = %d", fact);
}
```

```
int fact()
{
    int n, i, fact=1;
    printf("enter n value");
    scanf("%d", &n);
    for(i=1; i<n; i++)
    {
        fact=fact*i;
        return fact;
}
```

9.

```
#include<stdio.h>
void fact(int n);
int main()
{
    int n,
    printf("enter n value");
    scanf("%d", &n)
    fact(n);
}
void fact(int n)
{
    int n, fact=1, i;
    for(i=1; i<n; i++)
    {
        fact=fact*i;
        printf("fact=%d\n", fact);
}
```

Q.

```
#include<stdio.h>
int fact(int n);
int main()
{
    int n, i, fact=1;
    printf("enter n value");
    scanf("%d",&n);
    fact=fact(n);
    printf("fact=%d", fact);
}
int fac(int n)
{
    int i, fact=1;
    for(i=1;i<n;i++)
    fact=fact*i;
    return fact;
}
```

Q. write a c program to print Fibonosics series upto n.

```
#include<stdio.h>
void fib(int,n);
int main()
{
    int n;
    printf("enter n value");
```

```
scanf("%d",&n);
fib(n);
}
void fib(int n)
{
 int a=0,b=1,c;
 printf("%d %d",a,b)
    c=a+b;
 while(c<=n)
 {
  printf("%d",c);
   a=b;
   b=c;
   c=a+b;
 }
}
```

III-Unit

1. defination function and Advantages
2. Built in function with examples.
3. types of user defme function based on parameters and return type.
4. factorial and fibanosis programs.
5. C program diagram.
6. operators with examples.
7. primitive datatype, bytes, range in terms of 2 powers.
8. Array - defnations, types defnations, Syntax, memory list.

Q. Write a C program to print Transpose of a given matrice.

```
#include<stdio.h>
int main()
{
   int a[20][20], n, i, j;
printf("enter size of array");
Scanf ("%d", &n);
printf("enter array elements");
for(i=0; i<n; i++)
 {
   for(j=0; j<n; j++)
    {
      Scanf("%d", &a[i][j]);
    }
}
for(i=0; i<n; i++)
  {
    for(j=0; j<n; j++)
     {
       printf("%d", a[j][i]);
     }
  }
return 0;
}
```

eg:- $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

output:- $A = A^T$ $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$

Q. write a program to print opposite diagonal elements in a given matrice.

```
#include<stdio.h>
int main()
{
 int a[20][20], n, i, j;
printf("enter size of array");
Scanf("%d", &n);
printf("enter array elements");
for(i=0; i<n; i++)
{
 for(j=0; j<n; j++)
 {
 Scanf("%d", &a[i][j]);
 }
}
for(i=0; i<n; i++)
{
 printf("%d", a[i][n-i-1]);
}
return 0;
}
```

~~for principle diagonal~~ elements

for principle diagonal elements:-

```
for(i=0; i<n; i++)
{
    printf("%d", a[i][i]);
}
return 0;
}
```

Q. Sum of individual
write a program to find the sum of individual digit of a given number using non-Recursion function.

```
#include<stdio.h>
void sum( );
int main( )
{
      sum( );
}
void sum( )
{
   int n, sum=0, rem;
   printf("enter n value");
   scanf("%d", &n);
   while(n!=0)
   {Rem = n%10;
    sum1 = sum1 + Rem;
    n = n/10;
   }
   printf("sum1 = %d", sum1);
}
```

```
#include<stdio.h>
int sum( );
int main( )
{
      int [illegible];
      [illegible] = sum( );
```

```
printf("[illegible] = %d", [illegible]);
}
int sum()
{
    int n, sum1=0, Rem;
    printf("Enter n value");
    Scanf("%d",&n);
    [illegible]
    while(n!=0)
    { Rem=n%10;
      sum1=sum1+Rem;
      n=n/10;
    }
    [illegible]
  return sum1;
}
```

```
#include<stdio.h>
void sum(int n);
int main()
{
    int n
  printf("enter n value");
  Scanf("%d",&n);
  sum(n);
}
void sum(int n)
{
    int rem,sum1=0;
    while(n!=0)
```

```
    {   Rem = n%10;
        Sum = Sum + Rem;
        n = n/10
    }
    printf("Sum1 = %d", Sum1);
}
```

```
#include<stdio.h>
int sum(int n);
int main()
{
    int n, res;
    printf("enter n value:")
    scanf("%d", &n);
    res = sum(n);
    printf("Sum = %d\n", res);
}
int sum(int n)
{
    int rem, sum1=0;
    while(n!=0)
    {
        rem = n%10;
        sum1 = sum1 + rem;
        n = n/10;
    }
    return sum1;
}
```

Q. Write a c program to find the Reverse of a given digit.

```
#include<stdio.h>
void sum();
int main()
{
   sum();
}
void sum()
{
   int n, sum1=0, rem;
printf("enter n value");
scanf("%d",&n);
while(n!=0)
{  Rem = n%10;
   sum1 = sum1*10 + Rem;
    n = n/10;
}
   printf("sum1 = %d", sum1);
}
```

```
#include<stdio.h>
int sum();
int main()
{
   int res;
    res = sum();
    printf("res = %d", res);
}
```

Q. Write a c program to find sum of n natural numbers using non-recursive function.

```
#include<stdio.h>
int sum_natural(int n);
int main()
{
    int n, res;
printf("enter n value");
scanf("%d", &n);
 res = sum_natural(n);
printf("sum of n natural numbers=
                %d\n", res);
}
int sum_natural(int n)
{
    int i, sum=0;
    for(i=1; i<=n; i++)
    {
        sum = sum + i;
    }
  return sum;
}
```

i=1 i<=5 sum=0
i=1 1<=5
sum= 0+1=1
i=2 2<=5
sum = 1+2=3
i=3 3<=5
sum = 3+3=6

Date
Page

(8.) write a c program to check wheather the given number is Armstrong or not
153, 370, 371 → Armstrong no.

```
#include<stdio.h>
int armstrong(int n);
int main()
{
    int n, res;
printf("%d ",&n);
scanf("%d",&n);
 res=armstrong(n);
 if(res==n)
 printf("Amstrong number");
 else
 printf("Not Armstrong");
}
int armstrong(int n)
{
  int rem, S=0;
  while(n!=0)
  {
    rem = n%10;
    sum = sum+(rem*rem*rem);
    n = n/10;
  }
  return S;
}
```

(9.) write a c program to check whether the given number is palindrome or not

```
#include<stdio.h>
int palindrome( )
int main( )
{
    int n, res;
printf("enter n value");
Scanf("%d",&n);
res = palindrome(n);
If (res==n)
printf(" palindrome number");
else
printf (" Not palindrome number");
}
int palindrome(int n)
{
   int rem, sum=0;
   while (n!=0)
   {rem = n%10;
    sum= sum*10+rem;
     n=n/10;
   }
 return sum;
}
```

Q. write a c program to check toheather the given number is prime or not.

```
# include <stdio.h>
int prime()
int main()
{
  int n, res;
printf("enter n value");
Scanf("%d", &n);
res = prime(n);
if(res == 2)
printf("prime no");
else
prnf("not prime no");
}
int prime(int n)
{
 int c=0, i;
 for(i=0; i<=n; i++)
 {
   if(n%i==0)
     c++;
 }
 return c;
}
```

Q. write a c program to display factors of a given numbers.

```
#include<stdio.h>
void factors(int n)
int main()
{
    int n,i;
printf("enter n value");
scanf("%d",&n);
factors(n);
}
void factors(int n)
{
    int i;
   for(i=1;i<=n;i++)
    {
      if(n%i==0)
       printf("%d",i)
    }
}
```

Q. write a c program to find power of a given number.

```
#include<stdio.h>
int power(int b, int e);
int main()
{
    int b,e,res;
printf("enter base, exponent:\n");
```

```
Scanf("%d%d",&b,&e);
  res=power(b,e);
printf("power of %d^%d=%d",b,e,res);
}
int power(int b,int e)
{
 int i,x=1;
 for(i=1;i<=e;i++)
 {
   x=x*b;
 }
return x;
}
```

(9.) write a c program to find gcd of three numbers.

```
#include<stdio.h>
int gcd(int a, int b);
int main()
{
    int a,b,c,res;
 printf("enter a,b,c values");
 Scanf("%d%d%d",&a,&b,&c);
res=gcd(gcd(a,b),c);
print("gcd of %d and %d=%d\n",a,b,res);
}
int gcd(int a, int b)
{
   int i,g;
```

```
for(i=1;i<=a&& i<=b; i++)
{
   if (a%i==0 && b%i==0)
   {
      g=i;
   }
}
return g;
}
```

(9.) Write a cprogram to find LCM of three number.

```
#include<stdio.h>
int gcd(int a, int b);
int main()
{
   int a,b,c,res;
   printf("enter a,b,c values");
   scanf("%d %d %d",&a,&b,&c);
   res = gcd(gcd(a,b),c) ;
printf("LCM of 3 numbers = %d\n"
                  res);
}
int gcd(int a, int b)
{
   int i,g,l;
   for(i=1;i<=a&& i<=b;i++)
   {
   if (a%i==0 && b%i==0)
```

```
{
    q = r;
}
l = (a*b)/q;
return l;
}
}
```

Q. write a c program to find sum of all even or odd numbers in given range using non recursion function. from 1 to n.

```
#include<stdio.h>
int sum_odd_even(int s, int e);
int main()
{
    int e, res1;
printf("enter end value");
scanf("%d", &e);
printf("sum of even and odd number
    with in the range");
res1 = odd_sum(1,e);
printf("odd sum = %d\n", res1);
res2 = even_sum(2,e);
printf("even sum = %d\n", res2);
}
int sum_odd_even(int s, int e)
{
    int i, sum = 0;
```

```
for(i=s; i<=e ; i=i+2)
{
    sum = sum+i;
}
return sum;
}
```

1, 2, 3, 4, 5

odd = 1+3+5 = 9

even = 2+4 = 6

Date
Page

Built-in functions :-

Library / prefined / standard.

→ These are the functions who's defination is already known to the complier.
→ The user should known the function declaration or usage of function.
→ All this standard function declarations are present in header files.
ex:- printf(), scanf(), fprintf(), fscanf(). → stdio.h

pow(), sqrt(), ceil(), round()
→ math.h

malloc(), calloc(), realloc(), exit()
→ stdlib.h
standard library header file.

is lower(), isupper(), isdigit() → ctype.h

Under 'Math.h' Functions :-

1. Absolute function :- It returns the absault value of a given integer or real number.
Syntax :- int abs(float);
float abs(float);

Eg:- abs(-3) = 3
abs(-3.5) = 3.5
abs(3) = 3

2. ceil():- It returns the highest integer value which is near to the given number.
Syntax:- int ceil(float);
Ex:- ceil(1.2) = 2
ceil(-3.4) = -3

3. floor():- It returns the least integer value which is near to the given number.
Syntax:- int floor(float);
Ex:- floor(1.2) = 1
floor(-3.4) = -4

4. Truncate function : trunc()
It returns the integer value by truncating / removing the decimal part.
Syntax:- int trunc(float);
Ex:- trunc(3.4) = 3
trunc(-1.2) = -1

5. round():- It returns the nearest integer value depends on the decimal part.

Syntax: int round(float);
ex: round(2.1)=2
round(2.5)=3
round(2.9)=3

power Function: The power(x,y) function returns the value of the x raised to the power of y i.e., x^y
ex: pow(2,3)=8 $2^3=8$

Sqrt Function: sqrt() returns non-negative square root of a number. error occurs if the number is negative.
ex: sqrt(25)=5
sqrt(256)=16.

parameter passing in function:

(callby value and call by reference)

There are two ways of passing parameters from calling function to called function.
1. call by value (direct value)
2. call by Reference (Address of valu

1. call by value :-

→ In call by value method, the value of the actual parameters is copied into the formal parameters
→ A copy of the value is passed into the function.
→ changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.
→ Actual and formal are created at the different memory location.

```
#include<stdio.h>
void swap(int a, int b);
int main()
{
  int a=10;
  int b=20;
printf("Before swapping the values
      a= %d, b=%d\n", a, b);
swap(a, b);
printf("After swapping values a=%d
    b= %d\n", a, b);
}
void swap(int a, int b)
{
  int temp;
```

```
temp = a;
a = b;
b = temp;
printf("After swapping values a=%d,
        b=%d\n", a, b);
}
```

Before swap = a=10 ; b=20
After swap = a=10 ; b=20

2. call by Reference :-

→ In call by reference, the address of the variable is passed into the function call as the actual parameter.
→ An Address of value is passed into the function.
→ changes made inside the function validate outside the function also. The values of the actual parameters do change by changing the formal parameters.
→ Actual and formal arguments are created at the same memory location

Note :-

- we make use pointers(*) in call by reference
- pointer

```
temp = a;
a = b;
b = temp;
printf("After swapping values a=%d,
        b= %d\n", a, b);
}
```

Before swap = a=10 ; b=20
After swap = a=10 ; b=20

2. call by Reference :-

→ In call by reference, the address of the variable is passed into the function call as the actual parameter.
→ An Address of value is passed into the function.
→ changes made inside the function validate outside the function also. The values of the actual parameters do change by changing the formal parameters.
→ Actual and formal arguments are created at the same memory location.

Note :-

- we make use pointers(*) in call by reference
- pointer

```
#include <stdio.h>
void swap(int *a, int *b);
int main()
{
  int a=10;
  int b=20;
  printf("befor swapping the value
          a=%d, b=%d\n",a,b);
  swap(&a,&b);
  printf("after swapping values
          *a=%d, b=%d\n",*a,*b);
}
void swap(int *a, int *b)
{
   int temp;
    temp=*a;
    *a=*b
    *b=temp;
} // printf("after swapping values
        a=%d, b=%d\n",*a,*b);
```

Date
Page

Passing Arrays to Function :-

1. passing individual elements
2. passing entire array.

1. passing individual elements

```
#include<stdio.h>
void fun(int x);
int main()
{
  int a[3]={10,20,30};
             0  1  2

  fun(a[1]); //20
}
void fun(int x)
 {
  x=x+20  // 20+20=40
  printf("x=%d\n",x); //40
}
```

2. entire array

```
#include<stdio.h>
void fun(int x[3]);
int main()
{
  int a[3]={10,20,30};
  fun(a);
}
void fun(int x[3])
{
```

```
int i;
for(i=0; i<3; i++)
printf("%d\n", x[i]);
}
```

Non - Recursion :-

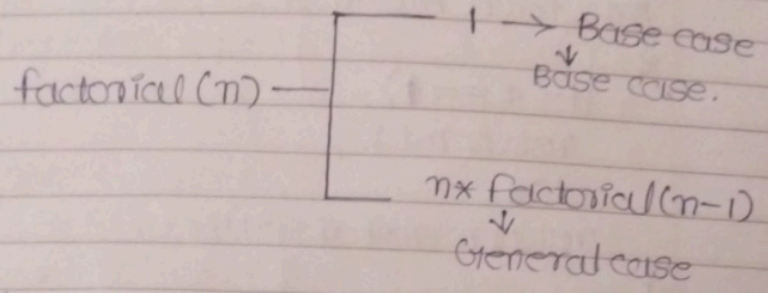
→ A function containing any loop statement in its implemenation is called non-Recursion.

Recursion :-

→ A function calling itself.
(If and else statement)
→ It contain
→ to write a recursion function we need two types.
1. Base case
2. General case.

1. Base case :- every recursion function must have base case based on that it stops the recursion.
2. General case :- The solution of function is return in terms of n is called General case.

Base case :-



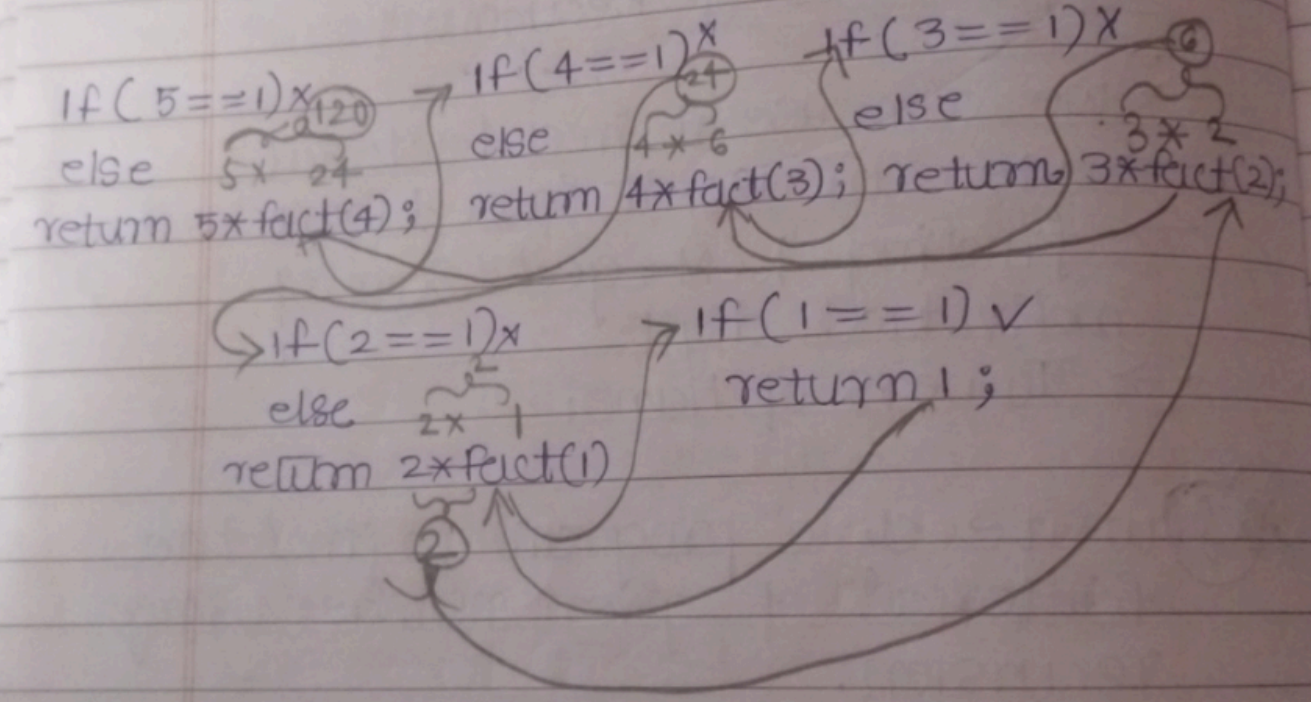
Applications of Recursion :-

→ we can use it in a factorial
→ we can use it in a fibonocciss series
→ finding LCM, gcd, sum of individual digit
→ Towers of Hanoi

(Q.) write a c program to find the factorial of given number using recursion.

```
#include<stdio.h>
int fact(int n);
int main()
{
    int n, res;
    printf("enter n value");
    Scanf("%d", &n);
    res = fact(n);
    printf("factorial = %d", res);
}
```

```
int fact(int n)
{
    if(n==1)
    return 1;
    else
    return n*fact(n-1);
}
```



8. C program to print fibonacci series

fib(n) —
- 0; 1; } Base case
- fib(n-2)+fib(n-1) General case

```
#include<stdio.h>
int fib(int n);
int main()
{
 int res,n;
printf("enter n value");
Scanf("%d",&n);
printf("Fibonacci series is;");
for(i=1;i<=n;i++)
{
 res=fib(i);
 printf("%d",res);
}
}
int fib(int i)
{
 if(i==1)
 return 0;
 else if(i==2)
  return 1;
else
return fib(i-2)+fib(i-1);
}
```

n=1 i=1
i=1 1<=1
res=fib(1)

if(1==1)
return 0;

printf → 0

i=2 2<=1 X

n=2
i=1 1<=2
res = fib(1)

if (1==1)
return 0;

printf – 0
i=2 2<=2
res = fib(2);

if (2==1) x
else if (2==2);
return 1;

printf – 1

printf – 0, 1

n=3
i=1 1<3
res = fib(1);

fib(1) = 0
fib(2) = 1

i=2 2<=3
res=fib(2)
i=3 3<=3
if (3==1) x
else if (3==2) x
else
return fib(i-2) + fib(i-1);

fib(1) fib(2)

1 2
0 1
0+1 = 1

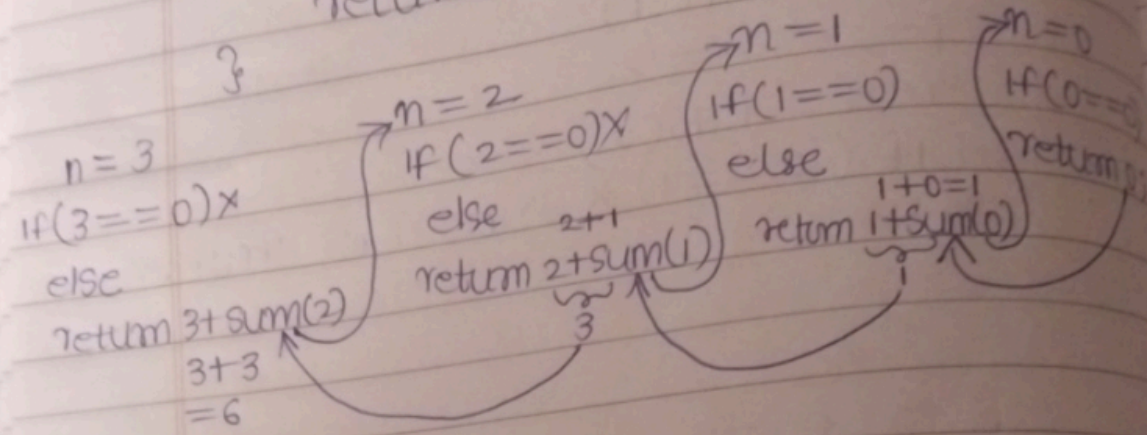
printf → 0, 1, 1

n = 2
i = 1 1<=2
res = fib(1)
i = 2 2<=2
res = fib(2)
if (2==1) *

Q. write a c program to find sum of n natural numbers using recursion.

```
#include<stdio.h>
int sum(int n);
int main()
{
   int n, res;
printf("enter n value");
scanf("%d", &n);
res = sum(n);
printf("sum of n natural numbers
      = %d \n", res);
}
int sum(int n)
{
   if (n==0)
     return 0;
   else
```

return n+sum(n-1);
}



(9) c program to find sum of individual digits of a given number.

```
#include<stdio.h>
int sum(int n);
 int s=0;
int main()
{
 int n, res;
 printf("enter n value");
 scanf("%d",&n);
  res = sum(n);
printf("sum of individual digits=
        %d\n",res);
}
int sum(int n)
{
 if(n==0)
  return 0;
 else
```

```
{ rem = n%10;
  s = s+rem;
  sum(n/10);
}
return s;
}
```

n = 456 ==0x

456==0x
else
{ rem=6
s=0+6=6
sum(456/10)

45==0x
else
rem=5
s=5+5=11
sum(45/10)

4==0x
else
rem=4
s=15
sum(4/10)

Q. C program to find reverse of a given number.

```
#include<stdio.h>
int sum(int n);
int s=0;
int main()
{
int n, res;
printf("enter n value");
scanf("%d",&n);
 res = sum(n);
printf("sum of reverse of a given no
        = %d\n", res);
}
 int sum(int n)
{
  if(n==0)
  return 0;
  else
{  rem = n%10;
     s = s*10+rem;
```

gcd(a, b) — [b=0 return a; / gcd(b, a%b)]

```
        sum = (n/10);
    }
    return s;
}
```

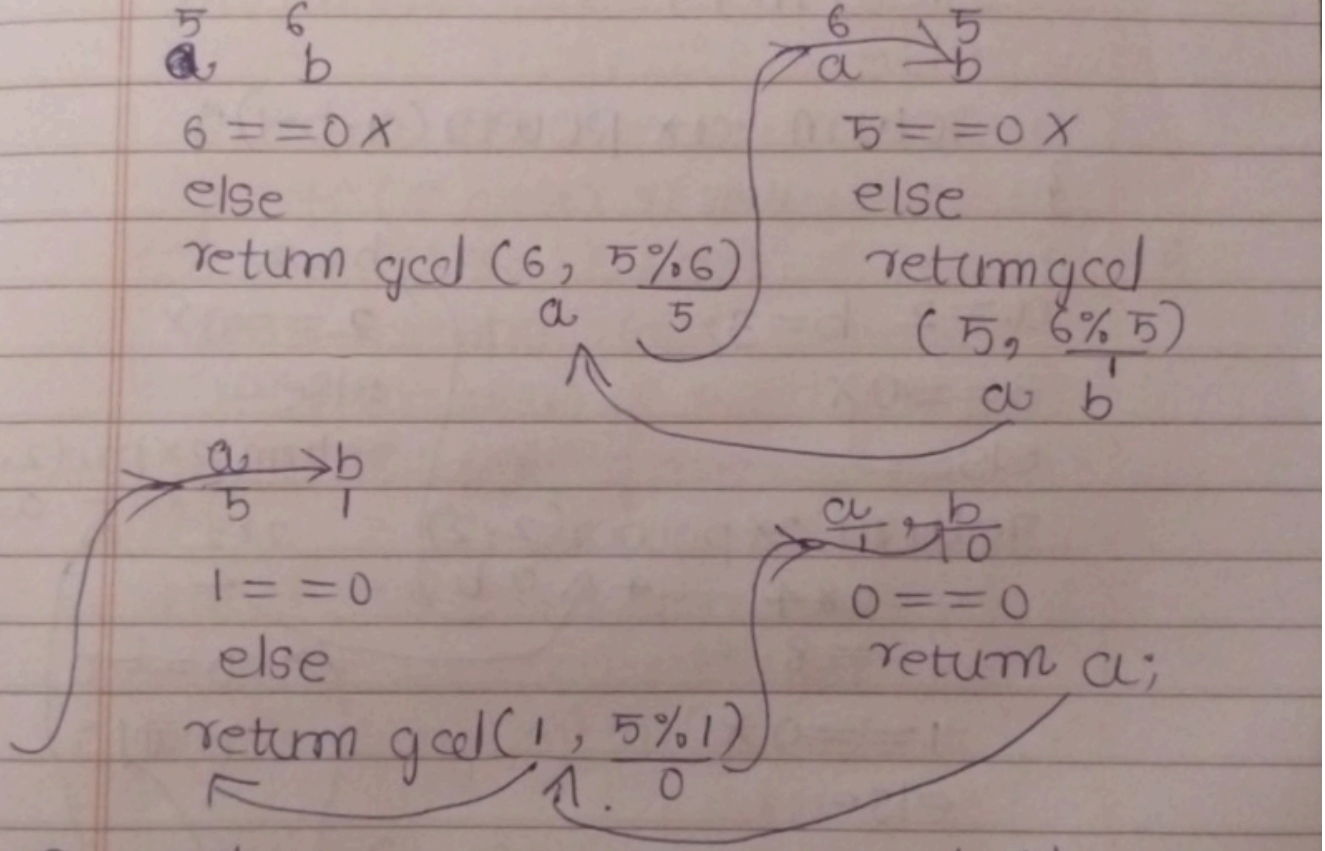
H/w Q. write a c program to find the number given no is Armstrong or not
Q. palinchrone or not.

Q. write a c program to find gcd, lcm of given number.

```
#include<stdio.h>
int gcd(int a, int b);
int main()
{
    int a, b, res;
    printf("enter a,b values");
    scanf("%d%d", &a, &b);
    res = gcd(a,b);
    printf("gcd of two numbers = %d\n",
                res);
    lcm = (a*b)/res;
    printf("lcm of 2 number = %d\n",
                lcm);
}
int gcd(int a, int b)
{
    if (b==0)
    return a;
```

```
else
 return gcd(b, a%b);
}
```

power(a,b) — b==0 return 1; / return a*pow(a,b-1);



8. write a c program to find the power of given number.

```
#include<stdio.h>
int power(int a, int b);
int main()
{
 int a, b, res, item;
 printf("Enter a, b, values");
 scanf("%d %d", &a, &b);
```

```
    res = power(a,b);
    printf("power of a,b = %d\n",res);
}
int power(int a, int b)
{
    if (b==0)
     return 1;
    else
    return a* power(a,b-1);
}
```

a=2 b=3

3==0 X

else

return 2* power(2,2)

2*4

=8

1==0 X

else

return 2* power(2,0)

2*1

=2

b=2

2==0 X

else

return 2* pow(2,1)

2*2 ab

2*2

4

4 ab

0==0

return 1;

Q. write a c program to find the given number is Armstrong or not

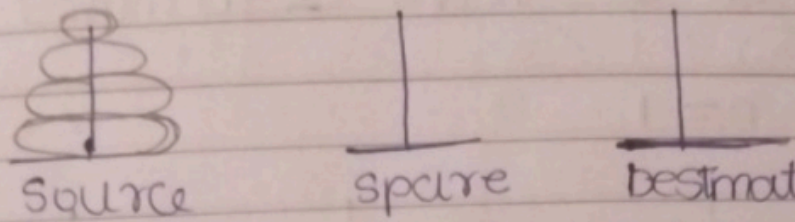
```
# include<stdio.h>
int Armstrong(int n);
int s=0;
int main()
{
 int n, res ;
 printf("enter n value");
 scanf("%d",&n);
 res= Armstrong(n);
 if(res==n)
printf("Amstrong number");
else
printf("Not Amstrong number");
}
int Amstrong(int n)
{
 int rem, s=0;
 if(n==0)
  return 0;
  else
 { rem=n%10;
     s=s+(rem*rem*rem);
      Amstrong(n/10);
 }
 return s;
}
```

Date
Page

(9.) palindrome or not.

```
#include<stdio.h>
int palindrome(int n);
int main()
{
    int n, rem;
printf("enter n value");
scanf("%d",&n);
res = palindrome(n);
if (res==n)
printf("palindrome");
 else
printf("Not palindrome");
}
int palindrome (int n)
{
  int rem, s=0;
   if(n==0)
    return 0;
   else
  {
     rem = n%10;
      s = s*10+Rem;
      palindrome(n);
  }
   return s;
}
```

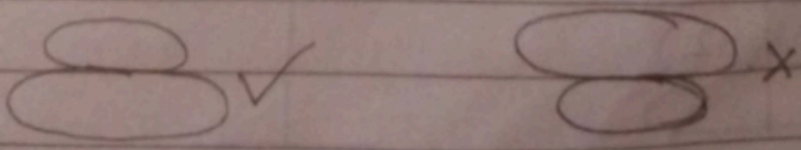
Q. Write a c program to find Towers of Hanoi.



→ It is a mathematical puzzle consists of three rods and n disks initially the source rod includes n disks in different sizes where the small disk is on the top of large disk.
→ The objective is to arrange the stack of disk of the source rod should be moved to the destination rod with the help of spare rod and must be arranged in the same fashion as in the source rod.

Rules:-

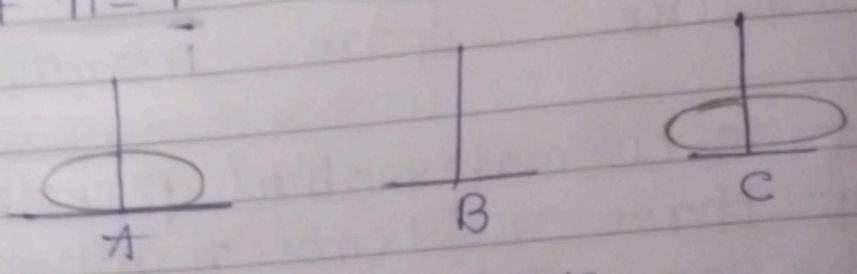
→ only one disk must be moved during each iteration.
→ Always it is allowed to fit the top most disk from the rod.
→ It is allowed to place the small disk on top of large disk



Note:-

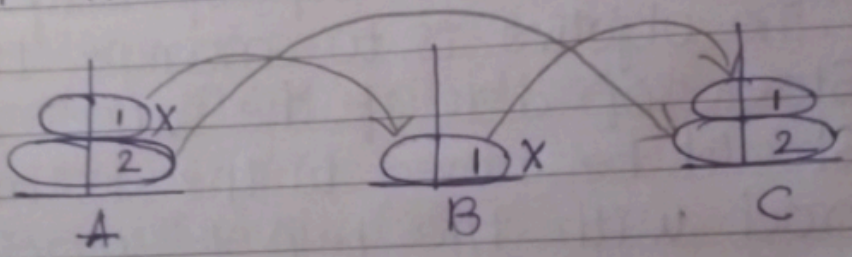
→ If we have a n number of disks then we have $\boxed{2^n - 1}$ moves are done.

① If $n = 1$



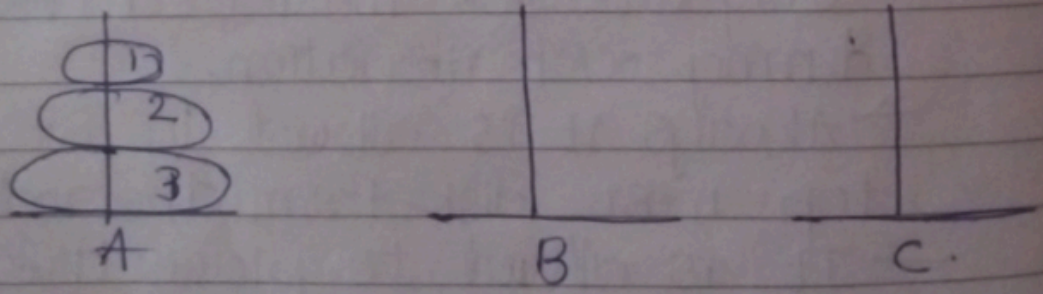
A to C move

② If $n = 2$

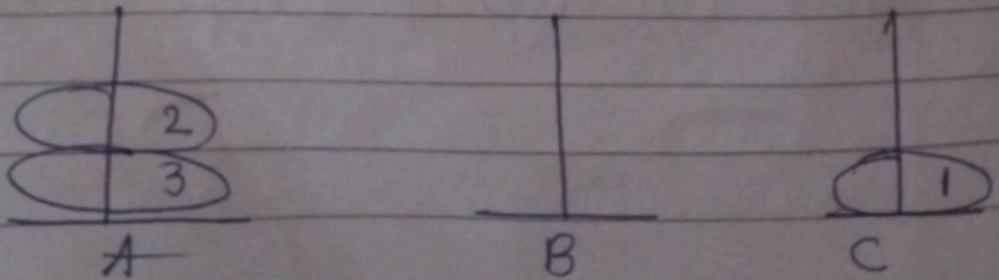


A to B
A to C
B to C

③ If $n = 3$

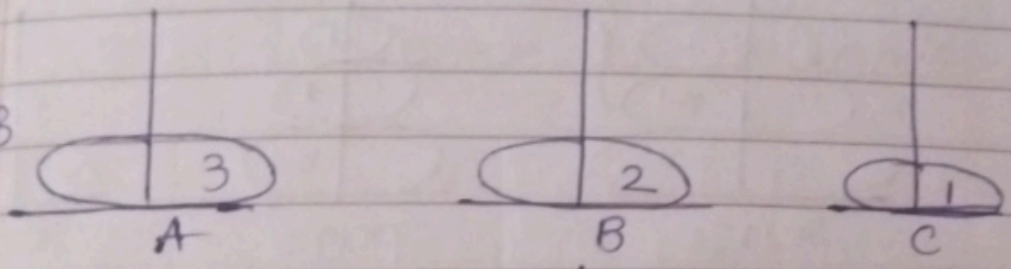


A to C

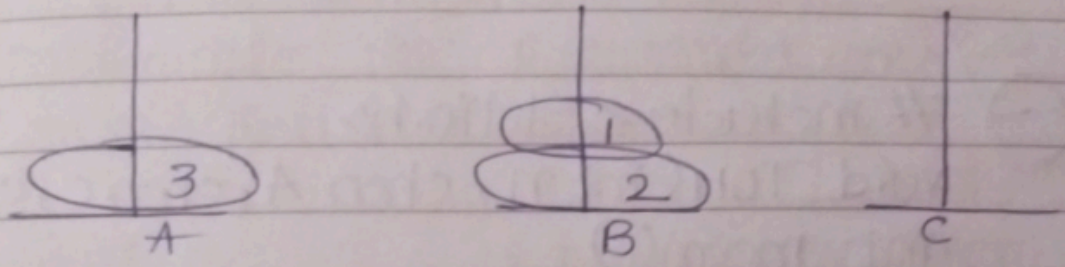


Date
Page

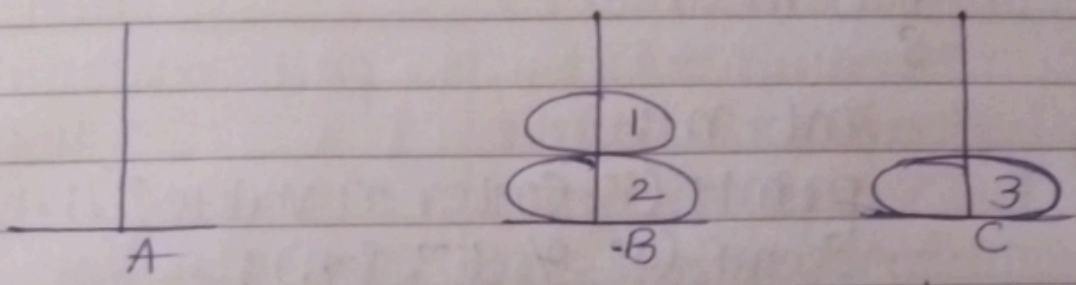
A to B



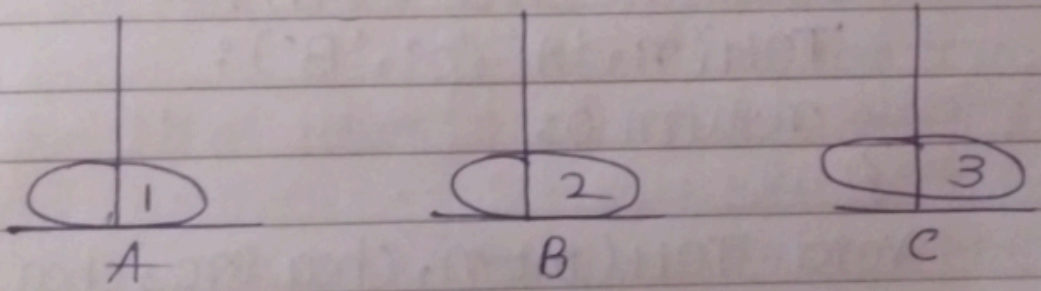
C to B



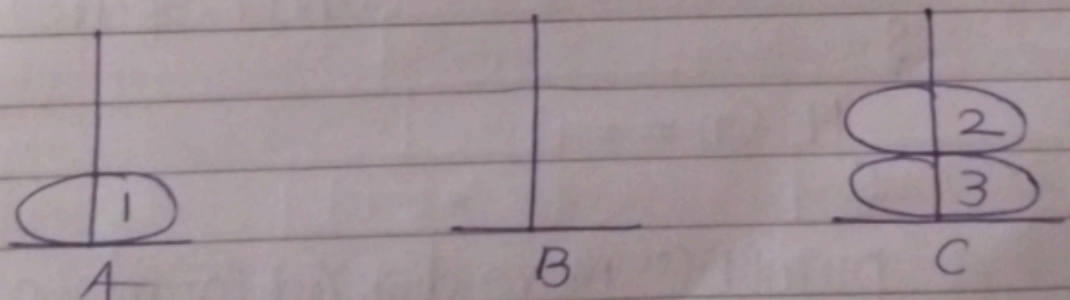
A to C



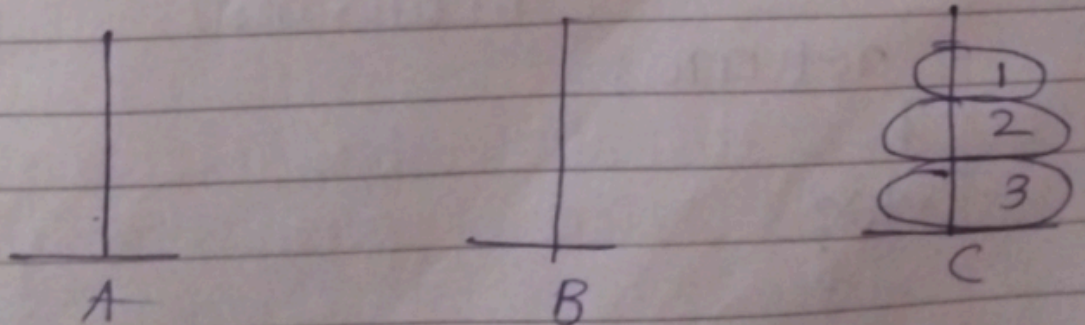
B to A

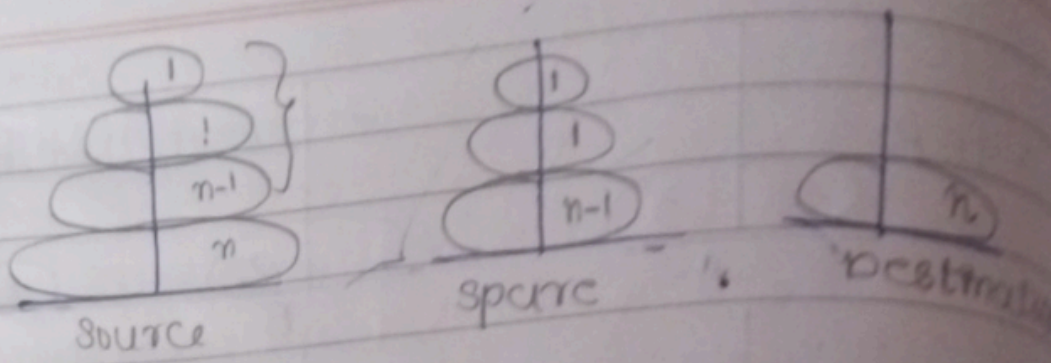


B to C



A to C






```
#include<stdio.h>
void TOH(int n, char A, char C, char B);
int main()
{
  int n;
  printf("enter n value");
  scanf("%d", &n);
  TOH(n, 'A', 'C', 'B');
  return 0;
}
void TOH(int n, char src, char dest,
                    char spare)
{
  if (n==1)
  {
printf("Move disk%d from %c to %c\n",
              n, src, des);
  return;
}
else
   {
     TOH(n-1, source, spare, pest);
printf("Move disk %d from %c to %c\n",
                    n, dest);
                      dest spare
```

Date
Page

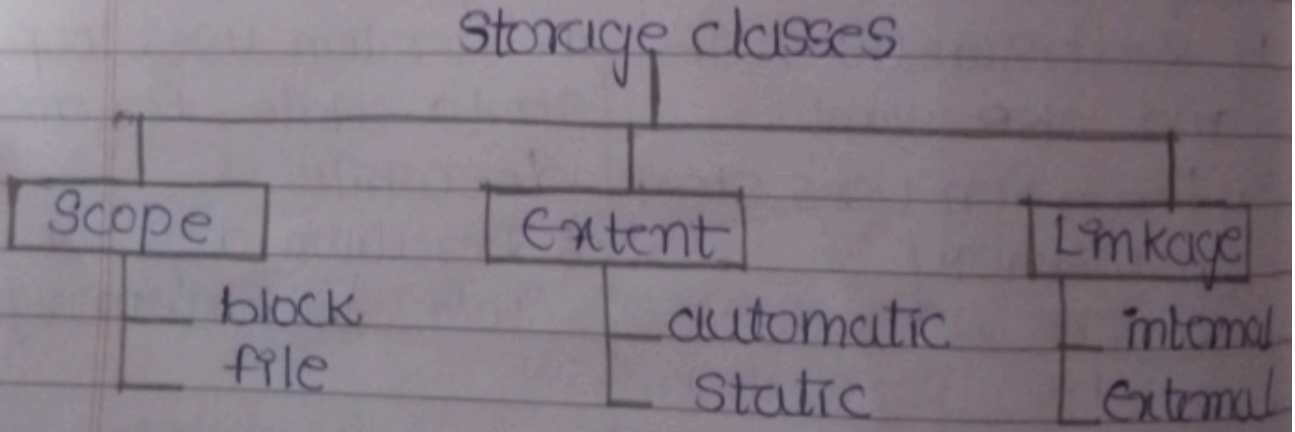
```
    TOH(n-1, spare, dest, source);
}
}
```

Difference b/w Recursion and non-Recursion.

Recursion	Iteration.
1. A function calling itself	1. A set of statements which are repeatedly executing.
2. when the base case satisfies recursions terminates.	2. when the condition fails, iteration terminates
3. when the base case fails it goes to infinite loops	3. when the condition is true all the time, it goes to infinite.
4. Recursion uses if and else stmts.	4. Iteration uses loop stmts while, for and do-while.
5. Recursion uses stack memory	5. Iteration does not use stack memory.

STORAGE CLASSES :-

→ Each and every variable is stored in computer memory.
→ Every variable in C language is associated with datatype and storage class
→ The Syntax :-
storage class datatype variable name;
ex :- auto int a;
int a;
→ Each storage class is associated with 3 features.
→ Storage class specifies the scope, extent, linkage.



block, automatic, internal — local variable for declaration of particular block.

file, static, external - Global variable declaration of variables for entire program code.

Scope :- The visibility of a varaible in the program is called a scope.

It is two types :-

1. block or function scope (Local)
2. file scope (Global)

The statements which are written with in opening and closing braces is block scope.

ex :-

```
int main()
{
    int a=0;
        {
            int a=20;
            printf("%d", a); //20
        }
    printf("%d", a); //10
}
```

2. File scope (Global)

The variables which are declared for the entire program.

```
#include<stdio.h>
int x=10;
void add();
int main()
{
    x=x+10;
    printf("%d", x); //20
```

```
add();
}
void add()
{
x = x + 30;
printf("%d", x); // 50
}
```

EXTENT :-

→ It is also called storage duration or lifetime of a variable.

→ extend defines how much time the variable is alive in the program.

These are 2 types :-

1. automatic
2. static.

1. Automatic extent :- The life time of variable within the block or function

→ The variable is assigned with memory when it enters into the block and gets released when it exit from the block.

2. Static extent :- The life time of variable is throughout the file (program)

→ All static variables are initialised to zero.

** Re-initialization of variable is not possible

LINKAGE :-

A large application program may be broken into several modules, with each module potentially written by a different programmer. Each module is a separate source file with its own objects.
We can define two types of linkage :-
1. Internal
2. External.

1. Internal linkage :- An object with an internal linkage is declared and visible in one module. Other modules cannot refer to this object.

2. External linkage :- An object with an external linkage is declared in one module but is visible in all other modules with a special keyword, extern.

Storage classes

1. Automatic Storage class
2. Register Storage class
3. Static Storage class
4. External Storage class

1. Automatic

The keyword is auto
The default storage class is auto
all automatic variables are stored in Ram.
Automatic variables are declared but it is not initialized, then it contains garbage value.

Scope : block (local)
extent : automatic
Linkage : Internal
default : Garbage value

Syntax :- auto datatype variable name;
Ex :- auto int a;

Ex :- #include <stdio.h>
int main()
{
auto int j=1;
{
auto int j=2;
{printf("%d
auto int j=3;
{
printf("%d ", j); //3
}

```
printf("%d", j); //2
}
printf("%d", j); //1
}
```

output :- 3, 2, 1

2. Register Storage class :-

→ Register variables are similar to auto variables. the only difference is register variables stored in cpu register instead of memory (RAM)
→ Register variables are faster than normal variables
→ mostly programmer use register to store frequently used variables.
→ programmers can store only few variables in the cpu register because size of register is less.

Syntax :- register datatype variable;
ex :- register int a;

```
#include<stdio.h>
int main()
{
  int m1 = 5;
  register int m2 = 10;
  printf("the value of m1 : %d", m1);
  printf("\n the value of m2 : %d", m2);
  return 0;
}
```

The value of m1 : 5
The value of m2 : 10.

Key word	Register.
Declaration	Inside a function/block.
Storage	CPU registers
Default initial value	Garbage value
Scope (visibility)	Within the function
Lifetime (Alive)	Till the end of function / block

Static

Syntax :- static datatype variable;

→ It is a variable which stores its value thorough out the program.
→ The variable can be declared as local or global.
→ A local static variable will be created and initialized only once when the function is executed for the first time.

Auto INT

```
#include<stdio.h>
void show();
int main()
{
  show();
  show();
  show();
}
void show();
{
  int a=1;
  printf("a=%d\n",a);
  a++;
}
```

output:- a=1
a=1
a=1

Static int

```
#include<stdio.h>
void show();
int main()
{
  show();
  show();
  show();
}
void show();
{
  static int a=1;
  printf("a=%d\n",a);
  a++;
}
```

output:- a=1
a=2
a=3

keyword	static(local)	static(global)
Declaration	inside a function/ block	outside all functions
Storage	RAM	RAM
default initial value	zero	zero
Scope(visibility)	with in the function / block	outside all functions

Lifetime (Alive)	Till the end of program	Till the end of program.

Extern

→ The keyword is extern.
→ These are the variables which are declared outside above all the functions.
→ These are also known as global variables
→ These variables can be accessed anywhere in the program and in other programs also

Syntax:- extern int n;
extern datatype variable name;

ex:-
```
# include <stdio.h>
extern int i;
int main()
{
    printf("the value of extern variable is = %d", i);
return 0;
}
# include <stdio.h>
int i = 48;
```

Date.....
Page.....

keyword	extern
Declaration	· outside of all variable
Storage	RAM
Default initial value	zero
Scope	Global multiple file
Life time	Till the end of the program.

Storage class	Declaration	Default initial value	Storage	Scope (visibity)	Lifetime (Alive)
auto	Inside the block/function	Garbage value	RAM	Within block/ function	Till the end of block/Function
Register	Inside the block	Garbage value	CPU register	within block	Till the end of block
static (Local)	Inside the block	Zero	RAM	with in block	Till the end of block
Static (Global)	outside all the function	Zero	RAM	Global (outside all the functn)	Till the end of the program
Extern	Outside all the variable	Zero	RAM	Global (multiple files)	Till the end of program.

Date
Page

pointers:-

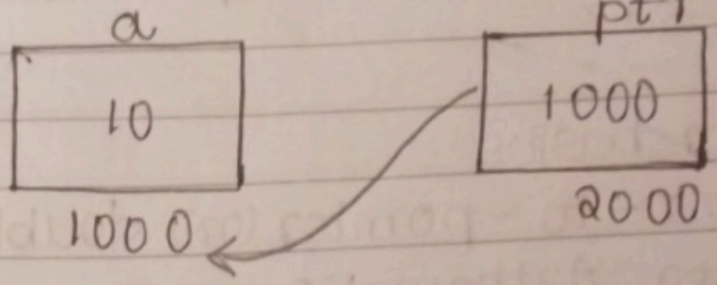
1 pointer basics
pointer-to-pointer (or) double pointer
pointer Arthemetic
Generic pointer
Array of pointer
function returning pointers
Dynamic memory allocation.

→ pointer is a derived datatype.

→ A pointer variable holds the Address of another.
* → indirection operator
Declaration of a pointer:-
Syntax:- datatype *pointer variable;
Ex:- int *ptr;
float *ptr;

Access variable using pointer:-
→ once a pointer variable is declared and initialized we can access the value of variable using pointer variable that is * operator.
→ where * operator is also known as indirection operator.
Syntax:- *pointer variable = & variable;
Ex:- *ptr = &a;
(or)

```
int *ptr=&a;
int a=10;
```




```
#include<stdio.h>
int main()
{
  int a=10;
  int *ptr;
  ptr= &a;
printf("value of a=%d\n",a); //10
printf("Address of a=%u\n",&a);//1000
printf("Address of ptr=%u\n",ptr);//1000
printf("value of ptr=%d\n",*ptr);//10
}
```

ptr =&a
ptr = 1000
&ptr = 2000

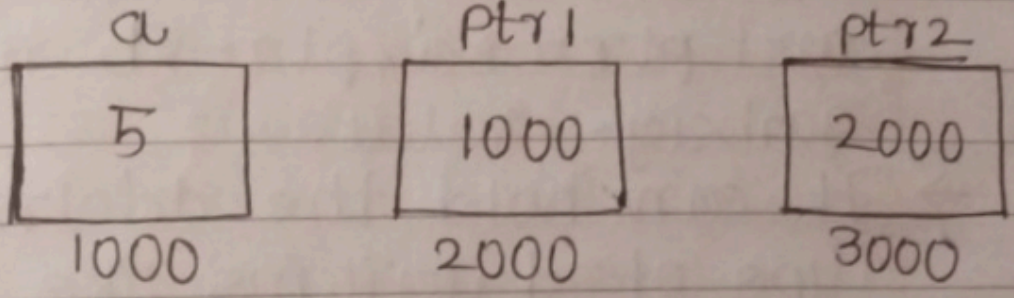
Applications :-

→ It will allocate size at runtime

- Dynamic memory allocation.
- Array & string
- File handling system
- To access and manipulate the address.
- More efficient
- Save memory space

pointer to pointer (or) Double pointer.

→ The pointer variable which holds the address of another pointer variable.



✓
```
#include<stdio.h>
int main()
{
  int a=5;
  int *ptr1, **ptr2;
  ptr1 = &a;
  ptr2 = &ptr1;
printf("value of a = %d\n", a);
pf("Address                    &ptr2
                               &ptr1
                               , &a);
                               , ptr1);
                               , ptr2);
                               , *ptr1);
                               , **ptr2);
```

type conversion = *(data type *)ptr;

Generic pointer (or) void pointer;

ptr = &a
ptr = &b
ptr = &z
it will only check and print ptr=&z and ptr=&a, ptr=&b it will print garbage value.

→ It can hold the address of any type of datatitems like int, float, char, double etc.

→ To read or print the data we must use type conversion.
((datatype *))

ex:

```
# include <stdio.h>
{
    void *ptr;
    int a = 10;
    float b = 20.5;
    char z = 'A';
    ptr = &a;
printf("value of a=%d", *(int *)ptr);
 ptr = &b;
printf("value of b=%f", *(float *)ptr);
   ptr = &z;
printf("value of z=%c", *(char *)ptr);
}
```

pointer variable = ptr1, ptr2
pointer values = *ptr1, *ptr2

pointer Arthematic :-

→ pointer variables are used to store address of variables.
→ Address of any variable is an unsigned integer value that is a numerical value. so, we can perform Arthematic operations on pointer values.
+, -, *, /, %
✓ a=5, b=7
- ptr1 = &a
- ptr2 = &b

(*ptr1) + (*ptr2) = 5+7
(*ptr1) - (*ptr2) = 5-7
(*ptr1) * (*ptr2) = 5*7
(*ptr1) / (*ptr2) = 5/7
(*ptr1) % (*ptr2) = 5%7

→ But when we perform arthematic operation on pointer variable the result depend on amount of memory required by the variable to which the pointer is pointing.
→ In c programming language we can perform the following arthematic operations on pointers.
1. Addition 2. Substraction
3. Increment 4. Decrement

int a = 5
int ptr = &a

① ptr = ptr + 3
= 1000 + 3 x (size of datatype)
= 1000 + 3 * 2
= 1006

② ptr = ptr - 3
= 1000 - 3 * 2
= 1000 - 6
= 994

③ ptr++
ptr = ptr + 1
= 1000 + (1 * datatype)
= 1000 + 2
= 1002

→ In c programming language the addition or subtraction operation on pointer variables is represented by using the formula.
current address of ptr ± (value to be added / subtracted * size of datatype)

```
#include<stdio.h>
int main()
{
int a, *int ptr;
float b, *float ptr;
char c, *char ptr;
int ptr = &a; // Assume address of a is 1000
```

```
float ptr=&b; // Assume address b is 2000
char ptr=&c; // " " c is 3000
printf("Address of int ptr=%u\n",
    int ptr); // 1000
printf("Address of float ptr=%u\n",
        float ptr); // 2000
printf("Address of char ptr=%u\n",
        char ptr); // 3000;
int ptr = int ptr+3 // 1000+(3*4)=1012
float ptr = float ptr+4 // 2000+(4*4)=2016
char ptr = char ptr+5 // 3000+(5*1)=3005
```

→ Increment & Decrement operations on pointer variables can be calculated as current address ± (1 * size of (datatype)) value of ptr.

```
#include<stdio.h>
int main()
{
  int a, *int ptr; float b, *float ptr.
  int ptr++; // 1000+(1*4)=1004
  float ptr++; // 2000+(1*4)=2004
  char ptr++; // 3000+(1*1)=3001
printf("Address of int ptr=%u\n",
        int ptr); // 1004
printf("Address of float ptr=%u\n",
        float ptr); // 2004
printf("Address of char ptr=%u\n",
        char ptr); // 3000)
```

→ passing pointers to functions :-
(refer call by reference both theory & program)

→ pointers & Arrays :-

→ The elements of array can be accessed by using pointers

→ continuous memory locations are allocated for all the elements of the array by the compiler.

→ The base address is the location of the first element with index zero of the array.

ex :- int a[5] = {10, 20, 30, 40, 50}

Index	a[0]	a[1]	a[2]	a[3]	a[4]
value	10	20	30	40	50
Address	1000	1004	1008	1012	1016

↓

base address

a = a[0] = 1000

↔ Normal representation for reading 10 elements in array.

```
for (i=0; i<=n; i++)
scanf ("%d", &a[i]);
```

a[0] = (a+0) = *(a+i) = 10

(1000+0) = *(1000+0) = 10

a[1] = (a+1)

(1000+1*4) = *(1004) = 20

a[i] = (a+i) = *(a+i);

Read & print 1D array elements using pointers.

Read

```
for(i=0; i<n; i++)
scanf("%d", (a+i));
```

print

```
for(i=0; i<n; i++)
printf("%d", *(a+i)); & print
```

Q) write a c program to read 1D array elements using pointers.

```
#include<stdio.h>
int main()
{
  int a[5], i, *ptr;
ptr = a or ptr = &a[0];
printf("enter array elements:");
for(i=0; i<5; i++)
scanf("%d", (a+i));
printf("Array elements:");
for(i=0; i<5; i++)
print("%d", *(ptr+i));
}
```

→ Representation of 2D array elements:

a[i] — *(a+i) — 1D

a[i][j] — *(*(a+i)+j) — 2D

Q) write a c program to print 2D array elements using pointers

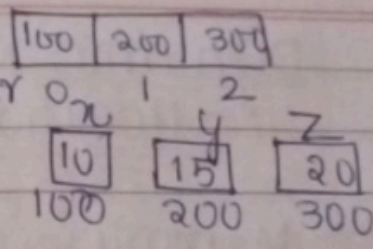
```
#include<stdio.h>
int main()
{
    int a[3][3]={10,20,30,40,50,60,70,80,90};
    int *ptr,i;
    ptr = &a[0][0];
    for(i=0; i<3; i++)
    {
     for(j=0; j<3; j++)
      {
       printf("%d", *(*(a+i)+j));
      }
     printf("\n");
    }
}
```

✓ Arrays of pointers :-

→ Array is a collection of variables of same datatype.

→ Smilliary Array of pointer is a collection of Address this contains address of more than one variable of same datatype in it.

Ex:
```
#include<stdio.h>
int main()
{
 int *ptr[3];
 int x=10, y=15, z=20;
 ptr[0]=&x;
```




```
ptr[1] = &y;
ptr[2] = &z;
for (i=0; i<3; i++)
printf ("Address of ptr[%d] = %d\n",
        (ptr+i), *(ptr+i));
}
```

→ Function returning pointers:-

→ A pointer which keep a Address of a function. is known as Function pointer.

✓
```
#include <stdio.h> int *abc();
int main()
{
    int *ptr;
    ptr = abc();
printf("Address of ptr = %u\n", ptr);
printf("Value of ptr = %d\n", *ptr);
}
int *abc()
{
   int *p, a = 100;
      p = &a;
     return p;
}
```

Dynamic Memory allocation / Memory allocation functions :-

→ Memory allocation is mainly of two types.
1. Static memory allocation
2. Dynamic memory allocation

→ Static memory allocation means memory is allocated to variables at complie.

→ Dynamic memory allocation means memory is allocated to variables at run time or Execution Time.

→ The memory used by the program is mainly categories into three types:
1. The program instructions and global variables are stored in a region known as permanent storage area.
2. The local variables are stored in Stack memory.
3. memory space b/w two areas is available for Dynamic allocation during execution of program this free region is called heap.
4. the size of heap is keeps on changing

local variables	} Stack
Free memory	} Heap
Global variables	} Permanent storage area
Instructions.	

→ there are four allocation functions which are used to allocate the memory for variables at run time.

1. malloc ( )
2. calloc ( )
3. realloc ( )

used for memory allocation.

→ free ( ) is used for releasing the memory when not required.

#include <stdlib.h> — header file.

1. Malloc ( )

→ Malloc ( ) is a block memory allocation function

→ The return type of malloc ( ) function is void

→ It contains the address of first byte of memory allocated from heap.

→ If the memory is successfully allocated then it returns the address otherwise it returns null.

→ The default value stored in the memory is garbage

Syntax:-

```
void * malloc (sizeof (datatype));
```

Ex:- `int * x;`

`x = (int *) malloc (sizeof (int));`

```
void * malloc (n * sizeof (datatype));
```

→ for Arrays

5 * 4 = 20

9. write a c program to read elements dynamically using malloc() function and interchange them using call by reference.

```
#include <stdio.h>
#include <stdlib.h>
int swap (int *x, int *y);
int main()
{
    int *a, *b;
  a = (int *) malloc (sizeof (int));
  b = (int *) malloc (sizeof (int));
printf ("enter a,b values:");
Scanf ("%d %d", a, b);
 Swap (a,b);
printf("After swapping a=%d b=%d\n",
                *a, *b);
}
int swap(int *x, int *y)
{
```

```
int temp;
temp = *x;
*x = *y;
*y = temp;
}
```

o/p :- Enter a, b values
10 20
After swapping
a = 20, b = 10

Q. write a c program to find sum of n given number using malloc() function.

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

int main()
{
  int *a, sum=0, n;
  printf("enter n value");
  scanf("%d", &n);
  a = (int *) malloc(n*sizeof(int))
printf("enter array elements");
for(i=0; i<n; i++)
{
  scanf("%d", (a+i));
  sum = sum + *(a+i);
}
```

```
    printf("sum=%d\n", sum);
}
```

(8.) write a program to read the element of 1D array using pointers and print them in reverse order.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int *a, n, i;
    printf("enter n value");
    scanf("%d", &n);
    a = (int *) malloc(n * sizeof(int));
    printf("enter array elements");
    for(i=0; i<n; i++)
    scanf("%d", (a+i));
    printf("Array elements are");
    for(i=n-1; i>=0; i--)
    printf("%d", *(a+i));
}
```

8. calloc()
→ calloc() is a continuous memory allocation function
→ It is generally used for Arrays
→ the return type of calloc() is void
→ It returns address of first byte of memory allocated from heap.
→ If the memory is succesfully allocated then it return the address otherwise it return null.
→ The default value for all the variables is zero.
Syntax:- void *calloc(int size, size of (datatype));

Ex:- int *x;
x = (int *) calloc(2, size of (datatype))
x = (int *) calloc(2, sizeof (int));

✓ Q. Write a program to read and print elements of 1D Array using calloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *a, n, i;
printf("enter n value");
scanf("%d", &n);
```

```
a=(int *) calloc (n, sizeof (int));
printf("enter array elements");
for(i=0; i<n; i++)
  scanf("%d", a+i);
printf("Array elements are:");
// for(i=n-1; i>=0; i--)
  printf("%d", *(a+i)); //
}
```

realloc ():- reallocation

→ reallocation is used to change the size of memory block without losing the old data.
→ The return type of realloc is void
→ The default values are null values

Syntax:-

```
void * realloc(void * old pointer,
                newsize * (Size of (datatype
```

Ex:- `int *x;`

```
x=(int *)calloc(2, Size of (int));
x=(int *)realloc(x, 10* Size of (int)
```

```
# include <stdio.h>
# include <stdlib.h>
int main()
{
  int *a, n, i, n1;
```

```
printf("enter n value");
scanf("%d",&n);
a=(int*)calloc(n,sizeof(int));
printf("enter array elements:");
for(i=0;i<n;i++)
scanf("%d",(a+i));
printf("Array elements are:\n");
for(i=0;i<n;i++)
scanf("%d",*(a+i));
printf("enter new size:");
scanf("%d",&n1);
a=(int*)realloc(a,n1*sizeof(int));
printf("enter array elements:");
for(i=0;i<n1;i++)
scanf("%d",(a+i));
printf("Array elements are:\n");
for(i=0;i<n1;i++)
printf("%d",*(a+i));
}
```

free() :-

→ free() is used to release the memory which is allocated by using any one of the memory allocation functions.

Syntax: void free(void * pointer);

```
int *a;
free(a);
```

→ It is always better to assign NULL for a pointer variable after using free() function.

```
free(a);
a = NULL;
```