Unit- 5

User - defined function (UDF)

Topics :-

- Concepts of User-defined functions
- Prototypes
- Defⁿ of function
- Parameters
- Parameters Passing
- Calling a function
- Macros
- Pre-processing

Prepared By :-
Ms. Jinal Zala
(ME CE), IIET

⇒ Need for User-defined Function :-

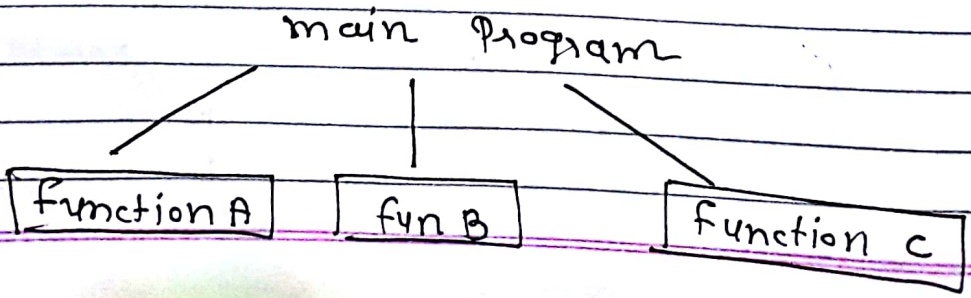
↦ Every program must have a main function to indicate where the program has to begin its execution.

↦ It is possible to code program in only the main() function. but it leads to a number of problems when program length is large.

↦ The program may become too large & complex and as a result the task of debugging, testing & maintaining becomes difficult.

↦ If a program is divided into functional parts, then each part may be independently coded & later combined into a single unit.

↦ These independently coded programs are called as subprograms that are much easier to understand, such subprograms are refered to as 'functions'.



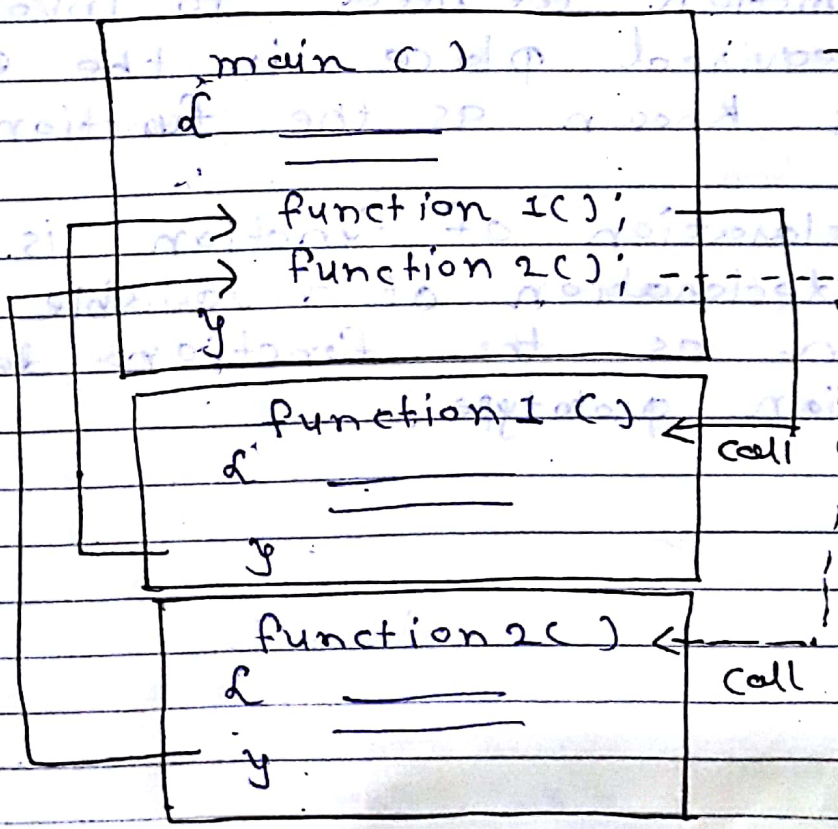
↳ Defⁿ :-

↦ A function is a self-contained block of code that performs a particular task.

↳ Advantages :-

1. It facilitates top-down modular programming.
2. Length of a source program can be reduced.
3. It is easy to use & also easy to find any error in any function.
4. A function may be used by many other programs.

↳ Flow of Muti-function Program :-



- It is called as 'Modular Programming'. Fun 1 & Fun 2 are different modules.

↦ Elements of User-defined functions :-

↦ In order to make use of a user-defined function, we need to establish three elements that are related to functions,

1.) function Defn :-
2.) function call :-
3.) function declaration :-

↦ The function defn is an independent program module that is specially written to implement the requirements of the function.

↦ This function we need to invoke it at a required place in the program. This is known as the function call.

↦ The declaration of function is required like declaration of a variable. This is known as the function declaration or function prototype.

Function Declaration :- (Function Prototype)

→ Like variables, all functions in a C program must be declared, before they are invoked or called in the main function.

→ A function declaration consists of four parts:-

- function type
- function name
- parameter list
- terminating semicolon.

Syntax :-

```
function_type  fun_name (parameter list);
```

Ex:- `void printline (void);` ← No parameters

Ex:- `int sum ( int a, int b);`

fun_type, name, parameters.

Page No.
Date

Defⁿ of functions:-

↦ A function defⁿ or function implementation shall include the following elements,

1. function name
2. function type
3. list of parameters
4. local variable declaration
5. function statements
6. Return statements

↦ All the six elements are groupe d into two parts,
- function Header
- function body

↳ format:-

```
Function-type fun_name (parameters list)
{
    local variable declaration;
    executable function statement;
    ....
    ....
    return statement;
}
```

(fun-type is void so return statement Not Required)

← No parameters so we can put empty bracket or void

Ex:-
```
void printline (void)
{   int a;
    pfc" Enter a";
    Sfc"%d",&a);
    Pfc"%d",a);
}
```

fun type is int so return statement needed

Ex:-
```
int sum(int a, int b)        <- Parameters
{  int c,d;   <- local variable
   c = a+b;
   Pfc("%d",c);
   d = a-b;
   return d;   <- so return statement
}
```

→ Function call:-

→ A function can be called by simply using the function name followed by a list of parameters or empty () bracket if parameters will be not passed.

Ex:-
```
void main()
{
   clrscr();
   printline();   // function call
   getch();
}
void printline(void)   <- fun defn
{  int a;
   pfc"Enter a:");
   Sfc"%d",&a);
   Pfc"%d",a);
}
```

When function is calling the control goes to the fun defn & executes all statement written in fun defn at the place of calling.

Page No.
Date

Examples:-

⇒ CWAP to Print Your Name Five times using User-defined function.

↦ 1st method:-

```
void printline(void);  ← function Prototype

void main()
{
    clrscr();
    printline();
    printline();
    printline();
    printline();
    printline();
    getch();
}

void printline(void);
{
    printf("My name is ABC\n");
}
```

Five times we have called function so, it will print five times which you have wrote in function def^n.

→ fun definition

↦ 2nd method:-

```
void printline(void);  ← function Prototype
void main()
{ clrscr();
    printline();  // function call (1 time)
    getch(); }
void printline(void)
{   int i;
  for(i=1; i<=5; i++)
  {   pf("My name is ABC\n");
  }
}
```

function defn (Here we have used loop for print five times)

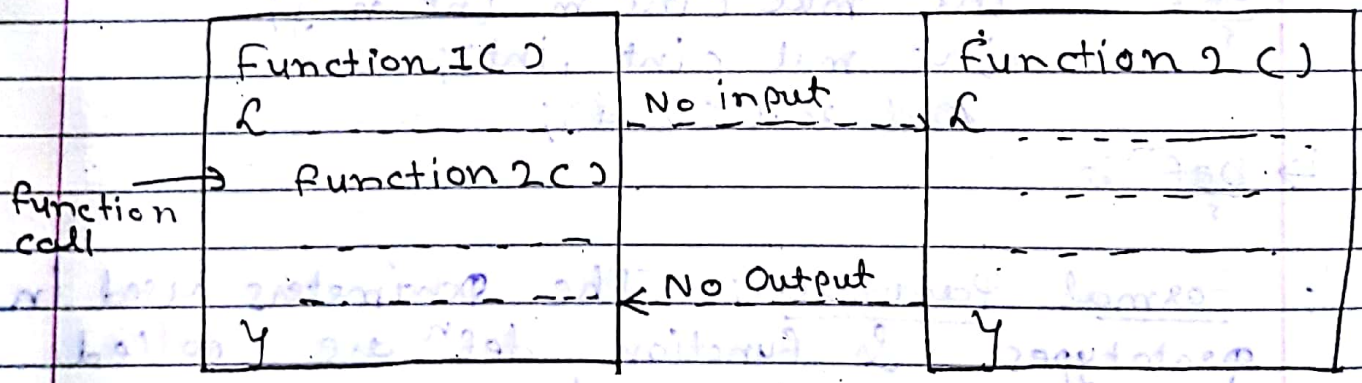
⇒ Categories of functions :-

1. Functions with no arguments & no return values.
2. Functions with arguments & no return values.
3. Functions with arguments & one return value.
4. Functions with no arguments & one return value.

1.) Functions with no arguments & no return values:

↦ When a function has no arguments, it doesn't receive any data from the calling function.

↦ Similarly, when it does n't return a value, the calling function doesn't receive any data from the called function.



No data communication bet^n functions

↳ Example :-

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

No arguments

No return value

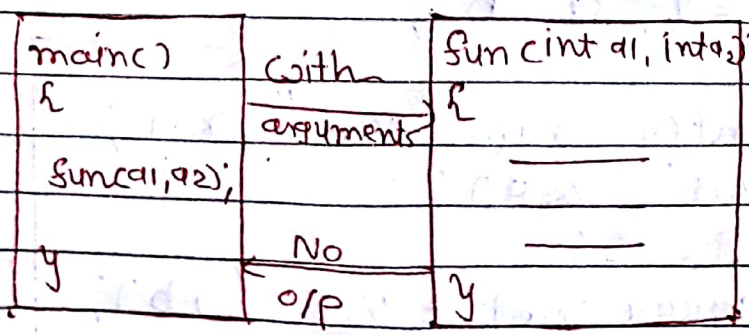
```
void sumc();    // declaration
void mainc()
{
    clrscr();
    sumc();     // fun call
    getch();
}
void sumc()     // fun defn
{
    int a=10, b=5;
    printf("Sum = %d", a+b);
}
```

O/P :-

Sum = 15

2. Functions with arguments but no Return value :-

⇒ In this category calling function has contain formal parameters but no return value.



↳ Example:- Sum of two integers using function.

```
# include <stdio.h>
# include <conio.h>
void Sum (int a, int b); // declaration
void main()
{
    int i, j;
    clrscr();
    printf("Enter two integer \n");
```

```
scanf("%d %d", &i, &j);
sum(i,j); // Function call
          └→ actual parameters
}
```

Formal Parameters

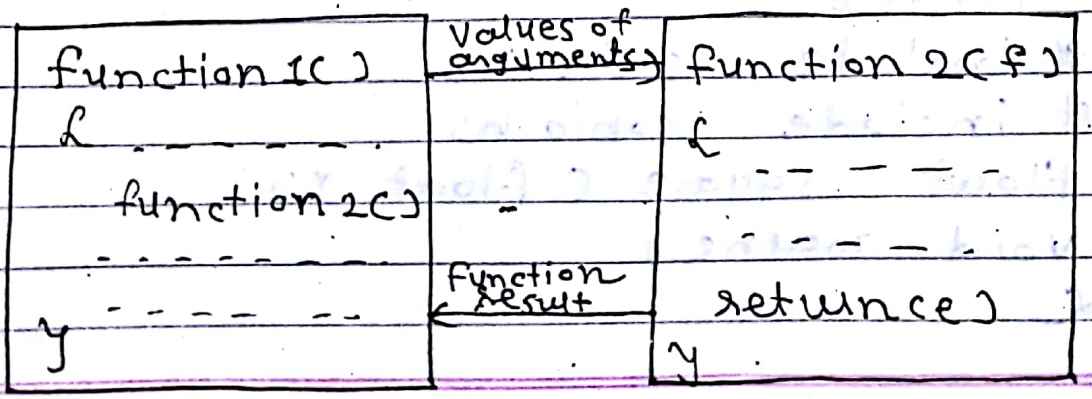
```
void sum(int a, int b) // function defn
{
    int c;
    c = a+b;
    printf("Sum of two numbers = %d", c);
}
```

Output :-

```
Give two integer numbers
2 3
Sum of two numbers = 5
```

3. Functions with Arguments & Return value :-

↔ In this category calling function receives argument from the called function & return result to the called function.



↳ Example:-1) Sum of given two integers using function.

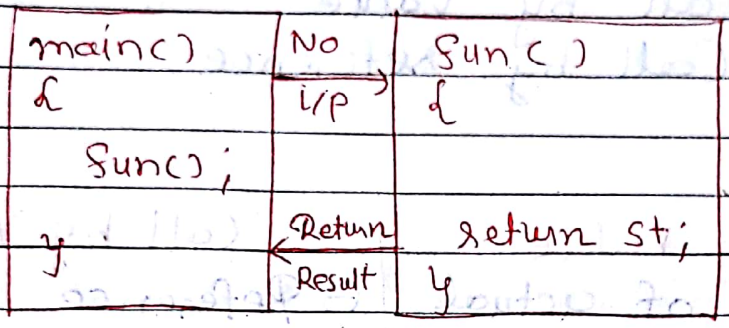
```
#include <stdio.h>
#include <conio.h>
int sum (int a, int b);

void main()
{
    int i, j;
    int ans;                    ['ans' is integer because our function will return value in integer]
    clrscr();
    printf("Enter two numbers:");
    scanf("%d %d", &i, &j);
    ans = sum(i, j);
    printf("Sum of two numbers = %d", ans);
    getch();
}

int sum (int a, int b)
{
    return a+b;  // return integer value
}
```

4. Functions with no arguments but with Return value:-

↦ In this category calling function has no contain any Parameters but return the value to the called function.



Ex:-5

```
int sum();                    // fun declaration
void main()
{
   clrscr();
   printf("sum = %d", sum());
   getch();                   // fun call
}
int sum()                     // fun defn
{
   int a=5, b=10;
   return(a+b);
}
```

No arguments

Return value

O/p:-

Sum = 15

↳ Def^{n}:-

1. Formal Parameters:- The parameters used in prototypes & function def^{n} are called as 'formal parameters.'

2. Actual Parameters:- The parameters used in function calls are called 'actual Parameters.'

↦ Parameter Passing to function :-

↦ When a function is called from other function, the parameters can be passed in two ways :-

- ↳ Call by value
- ↳ Call by reference.

Call by Value	Call by Reference
↦ A copy of actual parametess is passed into formal parameters.	↦ Reference of actual parameters is passed into formal parameters.
↦ Changes in formal parameters will not result in changes in actual parameters	↦ changes in formal parameters will result in changes in actual parameters.
↦ Pointers are not used.	↦ Pointers are used.
↦ Ex :-	↦ Ex :-

```
void swap(int a,int b);
void main()
{
  int x=10, y=5;
  swap(x,y); ← value
Pf("After swapping
x=%d, y=%d",x,y);
getch();
}
```

```
                    Pointers
                    ↓
void swap(int *a, int *b);
void main()
{
  int x=10, y=5;
  swap(&x,&y); ← Address
Pf("After swapping
x=%d, y=%d",x,y);
getch();
}
```

```
void swap (int a, int b)
{
    int t;
    t = a;
    a = b;
    b = t;
pf ("a = %d, b = %d",
      a, b);
}
```

⇒ O/p:- a = 5, b = 10
5 After swapping
x = 10, y = 5 ← Actual as it is

```
void swap (int *a, int *b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
pf ("a = %d, b = %d",
    *a, *b);
}
```

⇒ O/p:-
a = 5, b = 10
After swapping
x = 5, y = 10
Both swap

↳ Scope of Variables :- (Local & Global) :-

↦ Scope means the region of a program in which a variable is available for use.

Local variable	Global variable
↦ declared inside the function body.	↦ Declared outside the function body.
↦ Not initialized automatically.	↦ Initialized automatically by value 0 or depend on type.
↦ Use of local variables advisable.	↦ Too much use of global variables make program difficult.

↳ The Scope, Visibility & lifetime of Variables :-

↦ In C different variables are depend on the different storage class a variable may assume.

Types of different Storage class :-
1.) Automatic Variable
2.) External Variable
3.) Static Variable
4.) Register variables

Visibility :- The program's ability to access a variable from the memory.
lifetime :- The life time of a variable is the duration of time in which a variable exists in the memory during execution.

1. Automatic variables :- (local variables.)

⊢ declared inside a function.

⊢ They are created when the function is called & destroyed automatically when the function is exited.

⊢ We may also use the keyword auto to declare automatic variables.

```
main()
{
    auto int num;
    - - -
}
```

2. External Variables :- (Global Variable)

⊢ Variables that are both alive and active throughout the entire program are known as external variables.

⊢ If we want to declare external variable in inside main() function then we can do it by puting extern keyword with data-type.

```
main()
{
    extern int y;
    - - - - -
}
```

3. Static Variables:-

- The value of static variables persists until the end of the program.
- A variable can be declared static using the keyword static like

```
static int x;
```

4. Register Variables:-

- We can tell the compiler that a variable should be kept in one of the machine's register, instead of keeping in the memory.
- Since a register access is much faster than a memory access.

```
register int count;
```

↳ Example :-

```
# include <stdio.h>
void display();
int fun1(void);
int fun2(void);
int fun3(void);
int x;                    // global
void main()
{
```

```
    x = 10;
    printf ("x = %d \n", x);
    printf ("x = %d \n", fun1());
    printf ("x = %d \n", fun2());
    printf ("x = %d \n", fun3());
}

int fun1(void)
{
    x = x+10;
return (x); }
int fun2(void)
{
    int x;
    x = 1;
    return (x);
}
int fun3(void)
{
    x = x+10;          // global
    return (x);
}
```

Output :-

x = 10
x = 20
x = 1
x = 30

Example:-

```
#include <stdio.h>
#include <conio.h>
void stat (void);
void main()
{
    int i;
    for (i=1; i<=3; i++)
    stat();
}
void stat (void)
{   auto int x=0; // Automatic Variable
    static int x=0; // Static Variable
    x = x+1;
    printf ("x = %d\n", x);
    y = y+1;
    printf ("y = %d\n", y);
}
```

Output :-

```
x = 1
y = 1
x = 2
y = 1
x = 3
y = 1
```

Examples :-

I. Write a calculator program (add, subtract, multiply, divide). Prepare udf for each function.

```
#include <stdio.h>
#include<conio.h>
int add (int ,int );
int sub (int, int );
int mul (int ,int );
float div (int, int);

Void main ( )
{   int n1, n2 ;
    Printf ("Enter n1 & n2");
    Scanf (" %d %d", &n1, &n2);
    Printf("Addition of numbers is %d",
                                add(n1,n2));
  Printf (" In Subtraction = %d ", sub (n1,n2));
 Printf (" In multiplication = %d ", mul(n1,n2));
  Printf (" In Division = %f ", div (n1,n2));

int add (int N1, int N2 )
{ return (N1+N2);
}
int sub (int N1, int N2 )
{
   return (N1-N2);
}
int mul (int N1, int N2 )
{
  return (N1*N2); }
```

```
float div (int N1, int N2)
{
    return (N1/(float)(N2));
}
```

2. Write a function find out maximum out of three numbers.

```
int max (int, int, int);
void main()
{
    int n1, n2, n3, res;
    printf("Enter any three numbers: \n");
    scanf("%d %d %d", &n1, &n2, &n3);
    res = max(n1, n2, n3);
    printf("max num = %d", res);
    getch();
}
int max(int t1, int t2, int t3)
{
    if (t1>t2 && t1>=t3)
    return t1;
    else if (t2>=t3)
    return t2;
    else
    return t3;
}
```

3. Write a function power that computes x raised to the power y for integer x and y and returns double type value.

Date: / /
Page No.:

↦

```
double power(int, int);
void main()
{
    int x, y;
    double res;
    printf("Enter the value for base = ");
    scanf("%d", &x);
    printf("Enter the value of power = ");
    scanf("%d", &y);
    res = power(x, y);
    printf("%d raised to the power
        %d is = %lf \n", x, y, res);
    getche();
}

double power(int x, int y)
{   int i;
    double r = 1;
    for(i=1; i<=y; i++)
    {
        r = r*x;
    }
    return r;
}
```

↳ o/p :-

Enter the value for base :- 5
Enter the value of power = 3
5 raised to the power 3 is = 125.000

* WAP using function to find maximum of two nums.

```
int max (int, int);
void main()
{
    int a, b, z;
    pf("Enter a & b:");
    sf(" %d %d", &a, &b);
  z=max(a, b);
  pf("max num = %d", z);
   getch();
}

int max (int x, int y)
{
     if (x>y)
     return (x);
     else
     return (y);
}
```

* WAP to make function which receives number as argument & return sum of digit of that number,

```
int sum(int);
void main()
{
    int num;
    printf("Enter num:");
    scanf("%d", &num);
    printf("Sum of digits = %d",
                        sum(num));
    getch();
}

int sum(int n)
{   int s=0,a;

    while(n!=0)
    {
        a = n%10;
        s = s+a;
        n = n/10;
    }
    return(s);
}
```

* write a function which accepts a string & returns the length of the string.

```
int  strlength (char s[]);

Void main()
{
    char s[100];
    int l;
  Pf("Enter string:");
   gets(s);

   l = strlength(s);

Pf("String length =%d", l);
getch();

}

int  strlength (char s[])
{
    int len=0;
    while (s[len]!='\0');
   { len++;
    }

   return(len);
}
```

* Write a function which takes 2 numbers as parameters & return GCD of the 2 numbers.

→ GCD :- Greatest Common divisor

Ex :- $n_1 = 24 = 2 \times \boxed{2 \times 2 \times 3}$ → common
$n_2 = 60 = \boxed{2 \times 2 \times 3} \times 5$

So, GCD = $2 \times 2 \times 3 = (12)$

→
```
int gcd(int, int);
void main()
{
    int n1, n2, res;
    pf("Enter two nums:");
    sf("%d %d", &n1, &n2);
    res = gcd(n1, n2);
    pf("Gcd = %d", res);
    getch();
}

int gcd(int n1, int n2)
{   int ans, i;
    for(i=1; i<=n1 && i<=n2; i++)
    {
        if(n1%i==0 && n2%i==0)
            ans=i;
    }
```

```
return ans;
}
```

* Write a function which takes 2 numbers as parameters & returns the LCM of two nums.

LCM :- Lowest common multiple
Least common multiple

n1 = 30 = 2 X 3 X 5
n2 = 45 = 3 X 3 X 5

LCM = 2 X 3 X 3 X 5 = (90)

```
int lcm (int, int);
void main()
{
    int n1, n2, res;
    pf("Enter n1 & n2:");
    sf("%d %d", &n1, &n2);
    res = lcm (n1, n2);
    pf("LCM = %d", res);
    getch();
}
int lcm (int n1, int n2)
{
```

```
int i, ans, gcd;
for (i=1; i<=n1 && i<=n2; i++)
{
    if (n1%i==0 && n2%i==0)
        gcd=i;
}

ans = (n1*n2)/gcd;

return ans;
}
```

Ex:- $n_1 = 45 = 5 \times 3 \times 3$
$n_2 = 20 = 2 \times 2 \times 5$

LCM = $5 \times 3 \times 3 \times 2 \times 2$ = 180

$$\frac{n_1 * n_2}{gcd} = \frac{45 \times 20}{5} = \frac{900}{5} = 180$$

1⇒ The Pre-Processor:-

↦ The pre-processor, is a program that processes the source code before it passes through the compiler.

↦ It operates under the control of what is known as preprocessor command lines or directives.

↦ They all are begin with the symbol # and do not require a semicolon at the ends.

↳ Ex:- # define, # include

↳ Example:-

	Directive	function
1.	# define	- Defines a macro substitution
2.	# include	- Specifies the files to be included
3.	# if	- Test a compile-time condition
4.	# else	- specifies alternatives when # if test fails.

↦ These directives can be divided into three categories:

1. Macro substitution directives
2. file inclusion directives
3. Compiler control directives

1. Macro substitution directives:-

↦ Macro substitution means to replace the macro by the macro string.

```
# define macro-name string
```

Ex:-
```
# define MAX 10
```

Here, the preprocessor replaces the macro MAX every where in program by 10.

⊢ macro is mainly used for fast execution.

2. File inclusion:-

⊢ The file inclusion directive starts from # include.

```
# include <file-name>
```

Ex:- # include <stdio.h>

3. Compiler control directives:-

⊢ In ANSI more preprocessor directives are used for condition.

Example :- # if, # elif, # else

Prepared by
Jinal Zalavadiya