Functions

Objectives

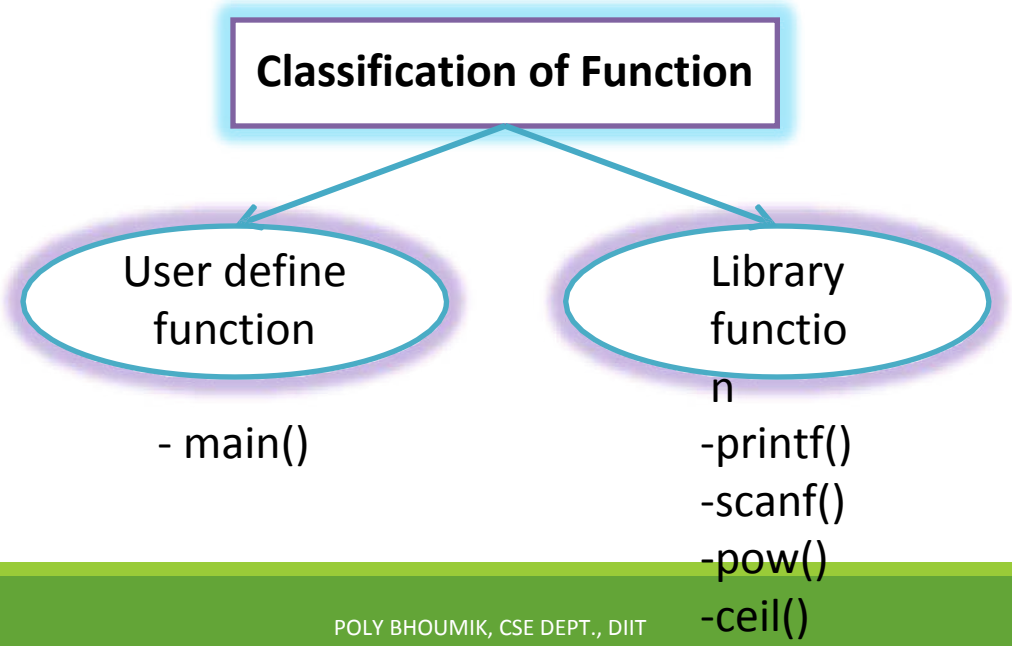
- **Explain the use of functions**
- **Explain the structure of a function**
- **Explain function declaration and function prototypes**
- **Explain the different types of variables**
- **Explain how to call functions**
- **Call by Value**
- **Call by Reference**
- **Explain the scope rules for a function**
- **Explain function pointers**

Functions

- A function is a self-contained program segment that carries out a specific, well-defined task
- Functions are generally used as abbreviations for a series of instructions that are to be executed more than once
- Functions are easy to write and understand
- Debugging the program becomes easier as the structure of the program is more apparent, due to its modular form
- Programs containing functions are also easier to maintain, because modifications, if required, are confined to certain functions within the program

What is a function

- A large program in c can be divided to many subprogram
  The subprogram posses a self contain components and have well define purpose.
- The subprogram is called as a ***function***
- Basically a job of ***function*** is to do something
- C program contain at least one function which is main().



Advantages of function

- It is much easier to write a structured program where a large program can be divided into a smaller, simpler task.
- Allowing the code to be called many times Easier to read and update.
- It is easier to debug a structured program where there error is easy to find and fix

ELEMENTS OF USER-DEFINED FUNCTIONS

- There are **three elements** related to functions
  - **Function definition**
  - **Function call**
  - **Function declaration**
- **The function definition** is an independent program module that is specially written to implement the requirements of the function
- To use this function we need to invoke it at a required place in the program. This is known as **the function call**.
- The program that calls the function is referred to as the **calling program or calling function**.
- The calling program should declare any function that is to be used later in the program. This is known as the **function declaration or function prototype**.

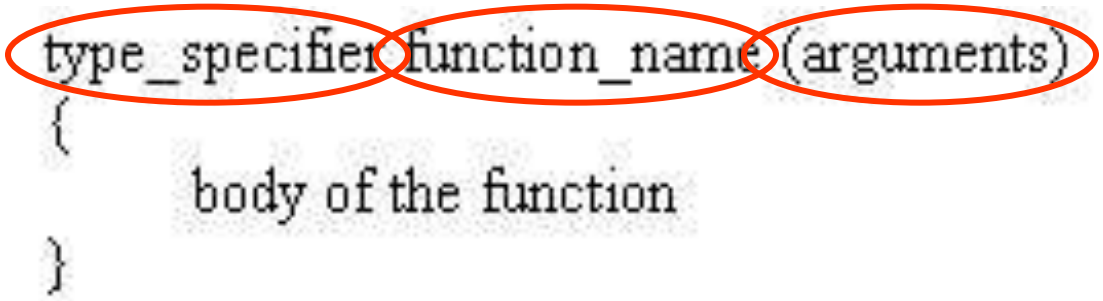
ELEMENTS OF USER-DEFINED FUNCTIONS

- **Function declaration or prototype** - informs compiler about the function name, function parameters and return value's data type.
- **Function call** – This calls the actual function
- **Function definition** – This contains all the statements to be executed.

Sno	C Function aspects	Syntax
1	Function definition	return_type function_name(arguments list) { Body of function; }
2	function call	function_name (arguments list);
3	function declaration	return_type function_name (argument list);

The Function Structure

- The general syntax of a function in C is :



- The type_specifier specifies the data type of the value, which the function will return.
- A valid function name is to be assigned to identify the function
- Arguments appearing in parentheses are also termed as formal parameters.

Example

```
1:    #include <stdio.h>
2:
3:    long cube(long x); /* Function prototype */
5:    long input, answer;
6:
7:    int main( void )  8:     {
9:    printf("Enter an integer value: ");  10:
scanf("%d", &input);
11:   answer = cube(input);
12:   printf("\nThe cube of %ld is %ld.\n", input,
answer);  13:
14:   return 0;


15: }
16:
17: long cube(long x) /* Function definition */
18: {
19:   long
x_cubed;
20:   x_cubed = x * x *
21:    x;  return
22:    x_cubed;
23: }
```

Arguments/formal parameter

Return data type

Actual parameters

/* calling function */

- Function names is cube
- Variable that are requires is long
- The variable to be passed on is X(has single arguments)—value can be passed to function so it can perform the specific task. It is called arguments

Output

Enter an integer value:4

The cube of 4 is 64.

Invoking a Function

- A semicolon is used at the end of the statement when a function is called, but not after the function definition
- Parentheses are compulsory after the function name, irrespective of whether the function has arguments or not
- Only one value can be returned by a function
- The program can have more than one function
  - The function that calls another function is known as the calling function/routine
  - The function being called is known as the called function/routine

How the function works

- C program doesn't execute the statement in function until the function is called.
- When function is called the program can send the function information in the form of one or more argument.
- Data is passed from the calling function to a called function by specifying the variables in a argument list.
- **Argument** list cannot be used to send data. Its only copy data/value/variable that pass from the calling function.
- The called function then performs its operation using the copies.

Function prototypes

- Provides the compiler with the description of functions that will be used later in the program
- Its define the function before it been used/called
- Function prototypes need to be written at the beginning of the program.
- The function prototype must have :

A return type indicating the variable that the function will be return

Syntax for Function Prototype

return-type function_name(arg-type name-1,...,arg-type name-n);

Function Prototype Examples

- double squared(double number);
- void print_report(int report_number);
- int get_menu_choice(void);

Function Declaration

- Declaring a function becomes compulsory when the function is being used before its definition
- The address() function is called before it is defined
- Some C compilers return an error, if the function is not declared before calling
- This is sometimes referred to as Implicit declaration

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

address()
{
        .
        .
        .
}
```

Function Definitions

- It is the actual function that contains the code that will be execute.
- Should be identical to the function prototype.

Syntax of Function Definition

return-type function_name(arg-type name-1,...,arg-type name-n) ---- **Function header**

```
{
declarations;
statements;
return(expression);
}
```

Function Body

DEFINITION OF FUNCTIONS

- A function definition, also known as function implementation shall include the following elements;
  - **Function name**;
  - **Function type;**
  - **List of parameters;**

  } **Function header**

  - **Local variable declaration;**
  - **Function statements; and**
  - **A return statement.**

  } **Function body**

- All six elements are grouped into two parts, namely,
  - **Function header** (First three elements); and
  - **Function body** (Second three elements)

Function Definition Examples

```
float conversion (float celsius)
{
float fahrenheit;  fahrenheit = celcius*33.8  return fahrenheit;
}
```

The function name‟s is conversion

This function accepts arguments celcius of the type float. The function return a float value.

So, when this function is called in the program, it will perform its task which is to convert fahrenheit by multiply celcius with 33.8 and return the result of the summation.

Note that if the function is returning a value, it needs to use the keyword return.

Function calling:

```
#include <stdio.h>  int
   show ( ) {
printf("It's a function");
{
Int main ( )
 {
Show ();
}
```

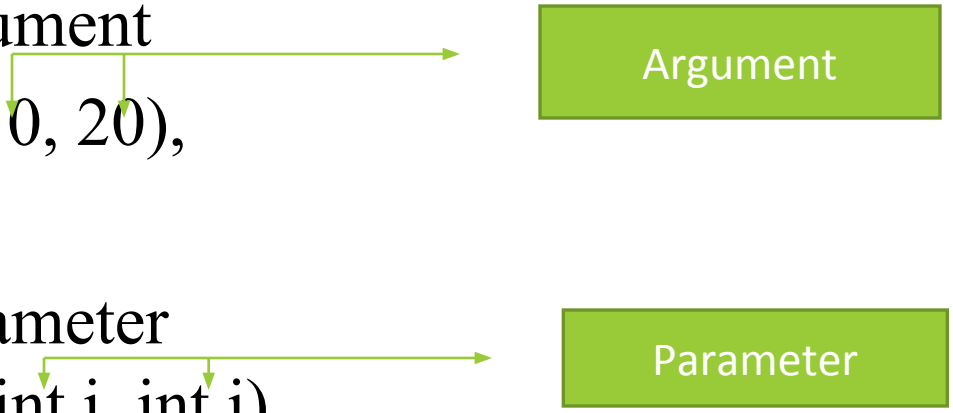

function call

Function Arguments:

- If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.
- The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
- While calling a function, there are two ways that arguments can be passed to a function:

Example of Parameter and Argument:

```
Main () {
//Argument
add (10, 20),
}
// Parameter
Add (int i, int j)
{  int j;
j=i+j;
}
```



Actual and Formal parameter:

```
#include <stdio.h>
void main(void)
{
void test_function(int) ;  test_function(1) ;\\actual parameter
}
Void test_function(i_recd_value) \\formal parameter
int i_recd_value ;
{
cout << "Value passed = " << i_recd_value ;
return ;
}
```

Function return types

Can be any of C‟s data type:
char int float
long……

Examples:

```
int func1(...)        /* Returns a type int.
float func2(...)      /* Returns a type float. */
void func3(...)       /* Returns nothing. */
```

Data Type of a Function

```
type_specifier function_name (arguments)
{
        body of the function
}
```

- The type_specifier is not written prior to the function squarer(), because squarer() returns an integer type value
- The type_specifier is not compulsory if an integer type of value is returned or if no value is returned
- However, to avoid inconsistencies, a data type should be specified

Variables

Local Variables

- Declared inside a function
- Created upon entry into a block and destroyed upon exit from the block

Formal Parameters

- Declared in the definition of function as parameters
- Act like any local variable inside a function

Global Variables

- Declared outside all functions
- Holds value throughout the execution of the program

CATEGORY OF FUNCTIONS

- C function with arguments (parameters) and with return value
- C function with arguments (parameters) and without return value
- C function without arguments (parameters) and without return value

Sno	C function	C function

1	with arguments and with return values	int function (int); // function declaration
		function (a); // function call
		int function(int a) // function definition
		{statements; return a;}

2	with arguments and without return values	void function (int); // function declaration
		function(a); // function call
		void function(int a) // function definition
		{statements;}

3	without arguments and without return values	void function(); // function declaration
		function(); // function call
		void function() // function definition
		{statements;}

4	without arguments and with return values	int function ( ); // function declaration
		function ( ); // function call

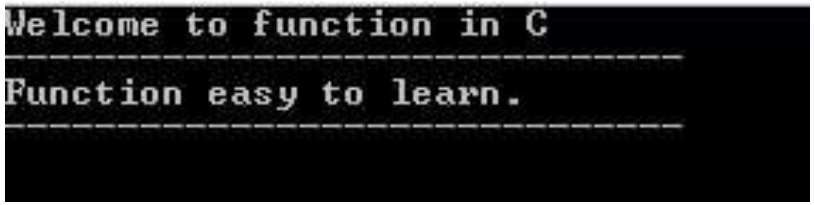
A function with no arguments and no return value

- Called function does not have any arguments
- Not able to get any value from the calling function Not returning any value
- There is no data transfer between the calling function and called function.

```
#include<stdio.h>  #include<conio.h>  void
printline();

void main()
{
printf("Welcome to function in C");
printline();
printf("Function easy to learn.");
printline();
getch();
}

void printline()
{
int i;  printf("\n");  for(i=0;i<30;i++)
{       printf("-");       }  printf("\n");
}
```



A function with no arguments and a return value

- Does not get any value from the calling function
- Can give a return value to calling program

```
#include <stdio.h>  #include <conio.h>  int send();

void main()
{
int z;  z=send();
printf("\nYou entered : %d.",z);  getch();
}

int send()
{
int no1;
printf("Enter a no: ");  scanf("%d",&no1);
return(no1);
}
```

```
Enter a no: 46

You entered : 46.
```

A function with an argument or arguments and returning no value

- A function has argument/s
- A calling function can pass values to function called , but calling function not receive any value
- Data is transferred from calling function to the called function but no data is transferred from the called function to the calling function
- Generally Output is printed in the Called function
- A function that does not return any value cannot be used in an expression it can be used only as independent statement.

```
#include<stdio.h>  #include<conio.h>  void add(int
x, int y);

void main()
{
add(30,15);
add(63,49);
add(952,321);
getch();
}

void add(int x, int y)
{
int result;  result = x+y;
printf("Sum of %d and %d is %d.\n\n",x,y,result);
}
```

```
Sum of 30 and 15 is 45.

Sum of 63 and 49 is 112.

Sum of 952 and 321 is 1273.
```

A function with arguments and returning a values

- Argument are passed by calling function to the called function
- Called function return value to the calling function
- Mostly used in programming because it can two way communication
- Data returned by the function can be used later in our program for further calculation.

```
#include <stdio.h>  #include <conio.h>  int add(int x,int y);

void main()

{
int z;

z=add(952,321);
printf("Result %d. \n\n",add(30,55));
printf("Result %d.\n\n",z);

getch();
}
int add(int x,int y)

{
int result;  result = x + y;  return(result);
}
```

Result 85.

Result 1273.

Send 2 integer value x and y to add()

Function add the two values and send back the result to the calling function

int is the return type of function

Return statement is a keyword and in bracket we can give values which we want to return.

Calling The Functions

- Call by value
- Call by reference

Call by Value:

- Call by Value, means that a copy of the data is made and stored by way of the name of the parameter.
- Any changes to the parameter have **NO** affect on data in the calling function.

Example Call by Value:

```
#include<stdio.h>  void c(int
 n, int m) {  int temp;
temp = n;
n = m; m = temp;
}
int main() {
int n=50,m=70;  c(n,m);
printf("\nNumber      : %d",n);
 printf("\nNumber 2 :%d",m);
return(0); }
```

Example program for C function (using call by value):

```
#include<stdio.h>
void swap(int a, int b);          // function prototype, also called function declaration
int main()
{
        int m = 22, n = 44;
        printf(" values before swap m = %d \nand n = %d", m, n);
        swap(m, n);     // calling swap function by value
}
void swap(int a,
int b)
{
int tmp;
tmp = a;  a = b;
b = tmp;
        printf(" \nvalues after swap m = %d\n and n = %d", a, b);
        }
```

Output

values before swap m = 22 and n =44
values after swap m = 44 and n = 22

Call by Reference:

A reference parameter "refers" to the original data in the calling function. Thus any changes made to the parameter are ALSO MADE TO THE ORIGINAL variable.

Example of Call by Reference:

```
#include <stdio.h>
void swap(int *x, int *y);
 int main () {
int a = 100; int b = 200;
printf("Before swap, value of a : %d\n", a );
printf("Before swap, value of b : %d\n", b );
swap(&a, &b);
printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b ); return
0;
}
```

Example program for call by referrence

```
#include<stdio.h>
void swap(int *a, int *b); // function prototype, also called function declaration
int main()
  {
        int m = 22, n = 44;
        // calling swap function by reference
        printf("values before swap m = %d \n and n = %d",m,n);
         swap(&m, &n);
  }
void swap(int *a, int *b)
  {
        int tmp;
        tmp = *a;
        *a = *b;
        *b = tmp;
        printf("\n values after swap a = %d \nand b = %d", *a, *b);
  }
```

Output

values before swap m = 22 and n =44
values after swap m = 44 and n = 22

Nesting Function Calls

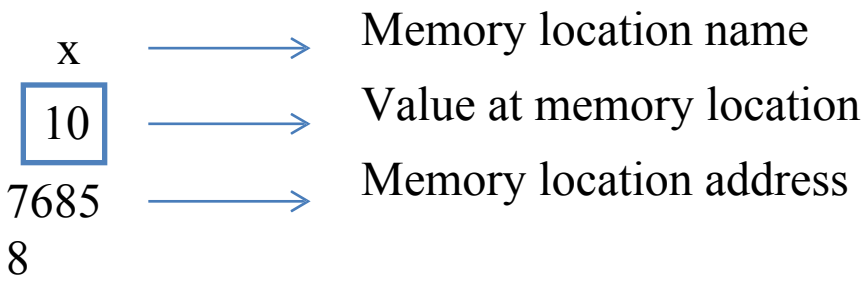
```
main()
   {
      .
      .
      palindrome
();
      .
      .
   }
```

```
palindrome()
   {
      .
      .
      getstr();
      reverse();
      cmp();
      .
      .
   }
```

Pointers

- Variable that declared occupies a memory according to it size
- It has address for the location so it can be referred later by CPU for manipulation T he „*“ and „&“ Operator

Int x=
10



We can use the address which also point the same value.

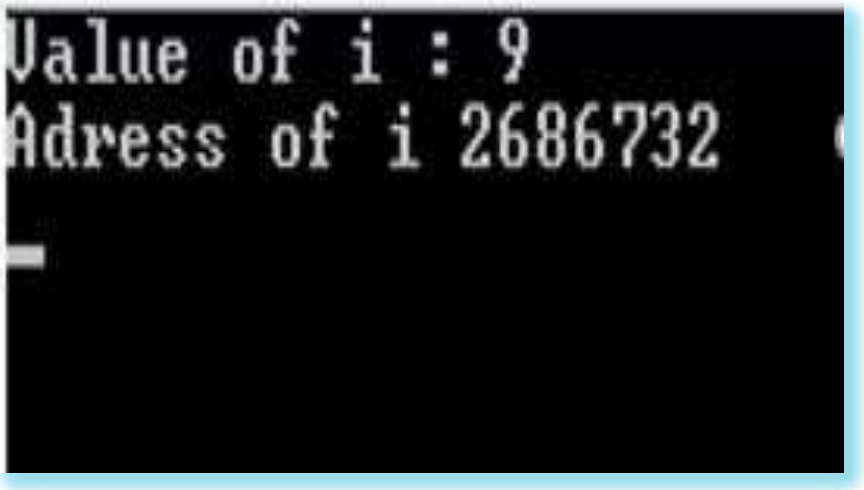
```
#include <stdio.h>  #include
<conio.h>


void main()

{
int i=9;

printf("Value of i : %d\n",i);
printf("Adress of i %d\n", &i);

getch();
}
```



& show the address of the variable

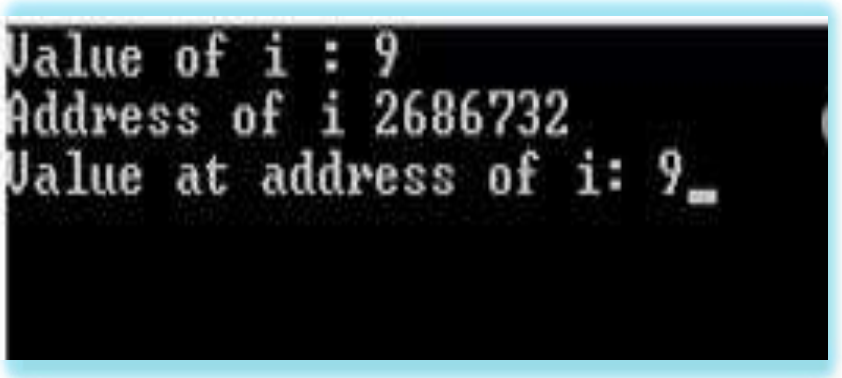
```
#include <stdio.h>  #include <conio.h>


void main()

{
int i=9;

printf("Value of i : %d\n",i);  printf("Address of
i %d\n", &i);  printf("Value at address of i: %d",
*(&i));

getch();
}
```



* Symbols called the value at the address

Recursion

A **recursive function** is a function that calls itself through another function.

Factorial of an integer n!

n! = n (n-1)(n-2)(n-3)…….1

1! = 1

2! = 2 (1) = 2

3! = 3 (2) (1) = 6

3! is also 3 (2!) = 6 because 2! = 2(1)

4! = 4 (3) (2) (1) = 24

4! is also 4 (3!) = 24

EXPRESSING FACTORIAL OF N AS A RECURSIVE FUNCTION

5!

5 * 4!

4 * 3!

3 * 2!

2 * 1!

1

THEREFORE, N! = N (N-1)!

Program to find the factorial of an integer

```
#include <stdio.h>
int factorial( int );
int main()
{
 int j;
    printf("Enter an integer \n");
 scanf("%d",&j);
    printf("Factorial is %d \n",factorial(j));
    return 0;
}

int factorial( int number)
{
 if (number <= 1)
         return 1;
 else
    return (number  *  factorial (number - 1));
}
```

Example:

```
#include <stdio.h>
 int sum(int n);
```

```
int main(){
int num, add;
printf("Enter a positive integer:\n");
scanf("%d",&num);
 add=sum(num);
 printf("sum=%d",add);
}
int sum(int n){  if(n==0)  return n;
else
return n+sum(n-1);      /*self call to function sum() */
}
```

FORMAL PARAMETER LIST

- The parameter list declares the variables that will receive the data sent by the calling program.
- They serve as input data to the function to carry out the specified task.
- They represent actual input values, they are often referred to as formal parameters.
- These parameters can also be used to send values to the calling programs
- **The parameter is known as arguments.**
  - **float quadratic (int a, int b, int c) { ….. }**
  - **double power (double x, int n) { ….. }**
  - **int sum (int a, int b) { ….. }**
- There is no semicolon after the closing parenthesis
- The declaration parameter variables cannot be combined