Problem Solving Through Programming (18ESCS01)

Unit-4: Functions & Pointers

MANJUNATH C R

Functions and Pointers

- FUNCTIONS - DEFINITION & SYNTAXOF A FUNCTION
- TYPES OF FUNCTION PARAMETERS
- PARAMETER PASSING TO FUNCTIONS
- RECURSION
- POINTERS – DEFINITION – INITIALIZATION
- DIFFERENT WAYS OF ASSIGNING POINTERS
- POINTER ARITHMETIC

FUNCTIONS

- A function is a **group of statements** that together perform a task.

- Every **C program has at least one function**, which is **main( ),** and all the most trivial programs can define additional functions.

- A large **C program is divided into basic building blocks called C function.** C function contains set of instructions enclosed by **"{ }"** which performs specific operation in a C program. **Collection of these functions creates a C program.**

- C functions are basic **building blocks in a program**. All C programs are written using functions to **improve re-usability**, **understandability** and to keep track on them.

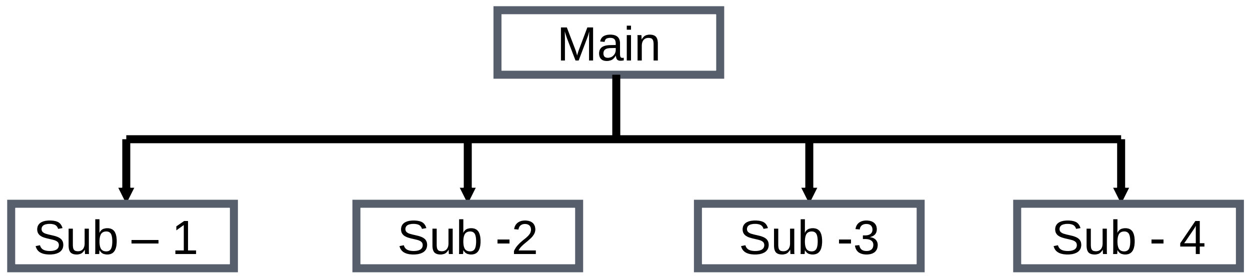
FUNCTIONS

- **Break a large problem into smaller pieces**
  - Smaller pieces sometimes called ‘modules’ or ‘subroutines’ or ‘procedures’ or ***functions***

  - Why functions?
    - **Helps manage complexity**
      - Smaller blocks of code
      - Easier to read
    - **Encourages re-use of code**
      - Within a particular program or across different programs
    - **Allows independent development of code.**
- Functions call or invoke other functions as needed.
- Each function solves one of the small problems obtained using top-down design.

BENIFITS OF A FUNCTION

- It can divide the Program in to Sub – Program Like : -



- Reusability of Code in the Program .
- Cleaner Structure of Main Function .
- Saves time and space.
- Easy to locate and isolate a faulty function for further investigations.
- Length of a source program can be reduced by using functions at appropriate places.

PARTS OF A FUNCTIONS

void main ( )

Return Type

of

Function

Name of Function

Argument

of

Function

FUNCTIONS - SYNTAX

- Syntax of a function is:

```
Return_type    function_name (Argument list)
   {
    statement block;        // Block of Code
    }
```

where:

- **Return type:** Return type can be of any **data type such as int, double, char, void, short etc.**
- **Function Name:** It can be anything, however it is advised to have a **meaningful name for the functions** so that it would be easy to understand the purpose of function just by seeing it's name.
- **Argument list:** Argument list contains variables names along with their data types. These arguments are kind of **inputs for the function**. For example – A function which is used to add two integer variables, will be having two integer argument.
- **Block of code: Set of C statements**, which will be executed whenever a call will be made to the function.

FUNCTIONS - DECLARATION

- A function declaration is as follows:

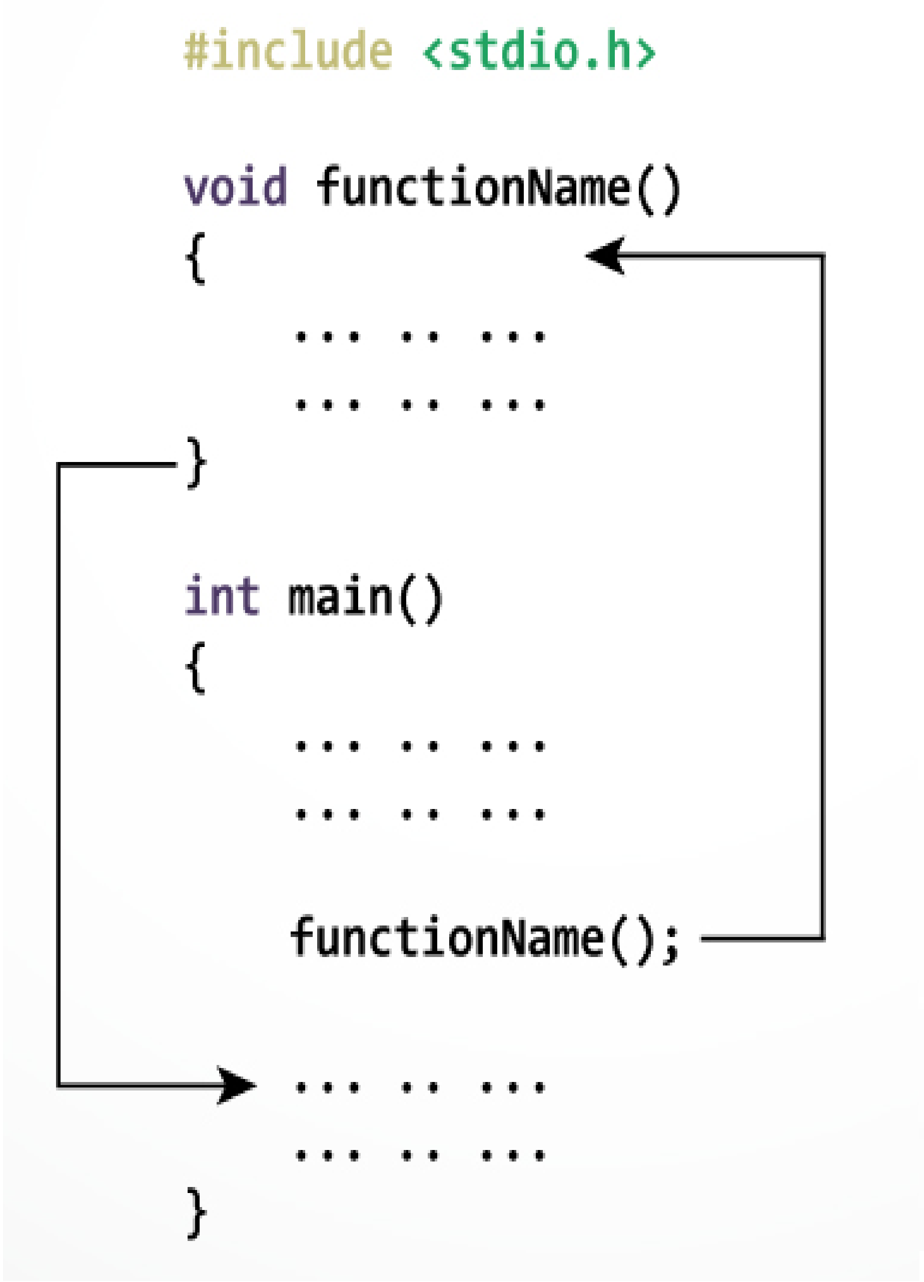
```
int max(int num1, int num2);
```

- Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration:

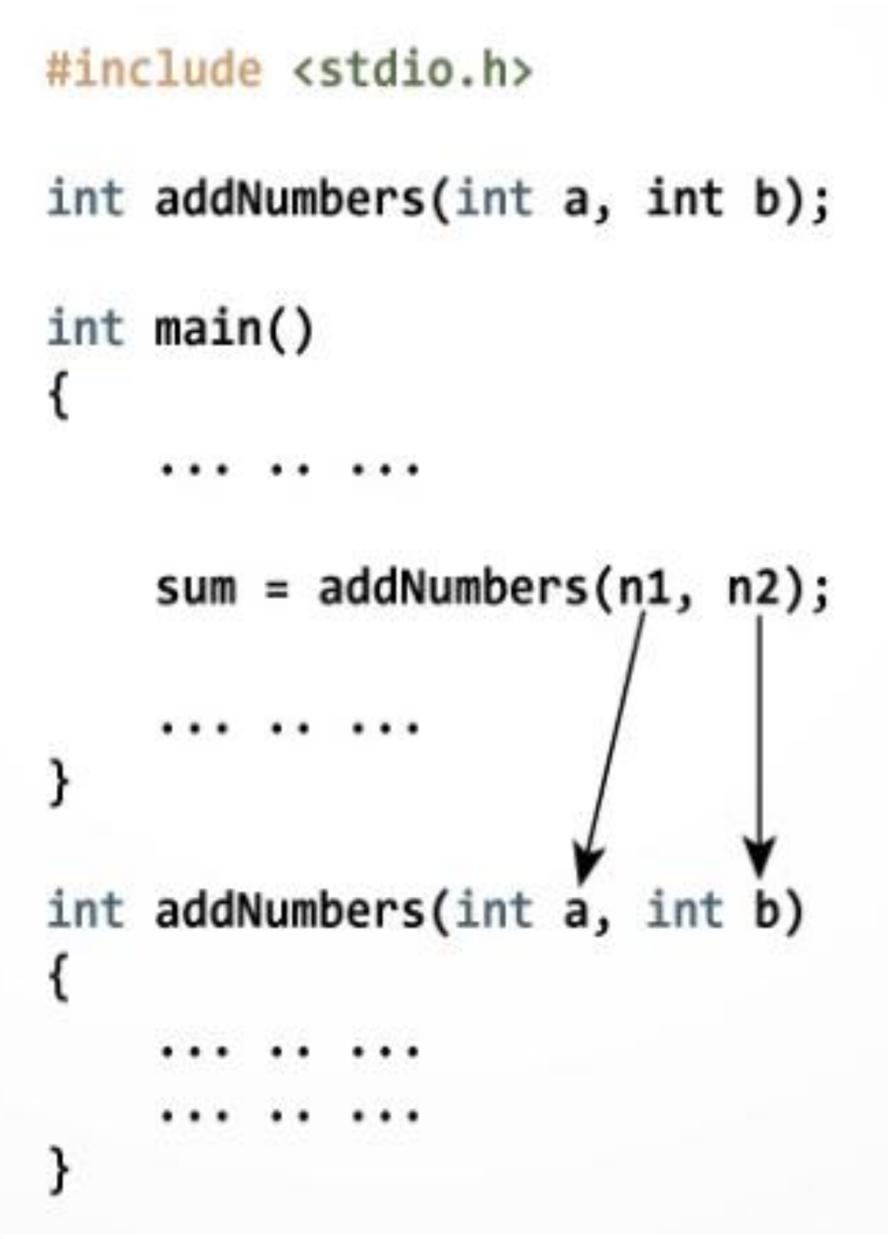
```
int max(int, int);
```

- Function declaration is required when you define a function in one source file and you call that function in another file.

How FUNCTION works in 'C'



```
#include <stdio.h>

void functionName()
{
    ... .. ...
    ... .. ...
}

int main()
{
    ... .. ...
    ... .. ...

    functionName();

    ... .. ...
    ... .. ...
}
```



```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    ... .. ...
}
```

FUNCTION – EXAMPLE 1

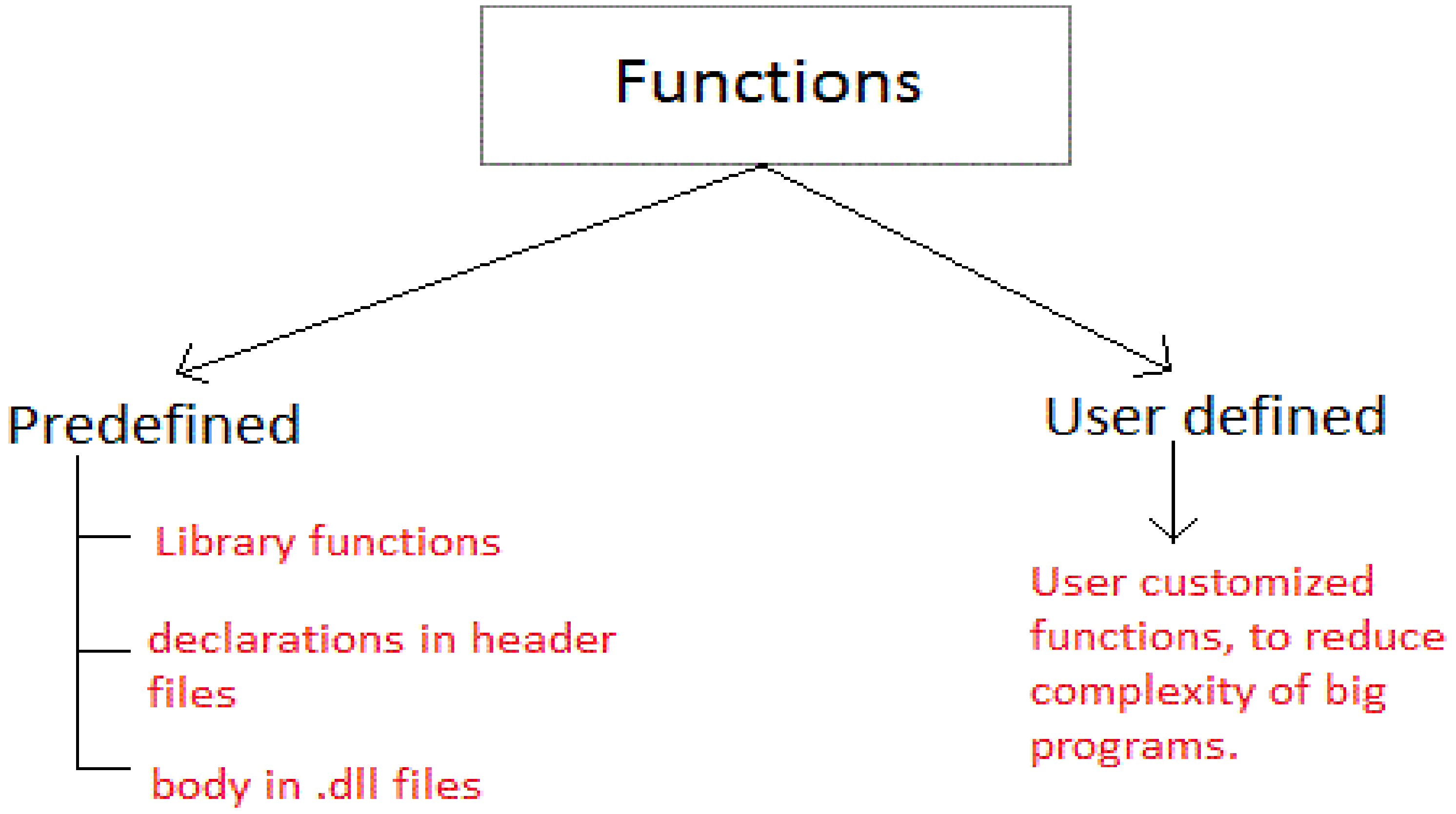
```
#include <stdio.h>   #include <conio.h>
int addNumbers(int , int );        // function declaration
int main( )
{
   int n1, n2, sum;
   printf("Enters two numbers: ");     scanf("%d %d", &n1, &n2);
   sum = addNumbers(n1,  n2);     // function call
   printf("sum = %d", sum);
   getch( );
}
int addNumbers(int a, int b)        // function definition
{  int result;
   result = a + b;
   return result;               // return statement
}
```

FUNCTION – EXAMPLE 2

```
#include <stdio.h>
int max(int , int );    /* function declaration */
 int main ( )
{   int a = 100;  int b = 200; /* local variable definition  */
    int ret;
    ret = max(a, b); /*calling a function to get max value */
    printf( "Max value is : %d\n", ret );
    return 0;
}
 int max(int num1, int num2) /* function returning the max of two numbers */
{   int result; /* local variable declaration */
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
```

TYPES OF A FUNCTION

- Functions can be classified into two categories:
  - **Pre-defined functions**
  - **User-defined functions**


Functions
Predefined
User defined
Library functions
declarations in header files
body in .dll files
User customized functions, to reduce complexity of big programs.

Types of a FUNCTION – (Contd.)

1. **Predefined standard library functions:** such as puts( ), gets( ), printf( ), scanf( ) etc. These are the functions which already have a definition in header files (.h files like stdio.h), so we just call them whenever there is a need to use them.

2. **User Defined functions:** The functions that we create in a program are known as user defined functions.

User – Defined Functions

(a) No Arguments & No Return Value

(b) No Arguments & a Return Value

(c) With Arguments & No Return Value

(d) With Arguments & a Return Value

(A) No Arguments & No Return Value

```
#include <stdio.h>
void checkPrimeNumber( );
int main( )
{   checkPrimeNumber( );     // argument is not passed
    return 0; }
 // return type of the function is void because function is not returning anything
  void checkPrimeNumber( )
{    int n, i, flag=0;
     printf("Enter a positive integer: ");        scanf("%d", &n);
     for(i=2; i <= n/2; ++i)
    {  if(n%i == 0)
       { flag = 1; }
    }
if (flag == 1)
   printf("%d is not a prime number.", n);
else
printf("%d is a prime number.", n);
}
```

(B) No Arguments But A Return Value

```
#include <stdio.h>
int getInteger( );
int main( )
{   int n, i, flag = 0;
    n = getInteger( ); // no argument is passed
    for(i=2; i<=n/2; ++i)
   {   if(n%i==0)
         { flag = 1; break; }
    }
    if (flag == 1)
        printf("%d is not a prime number.", n);
    else
       printf("%d is a prime number.", n);
     return 0; }
   int getInteger( ) // returns integer entered by the user
   { int n; printf("Enter a positive integer: "); scanf("%d", &n);
return n;
}
```

(C) WITH ARGUMENTS BUT NO RETURN VALUE

```
#include <stdio.h>
void checkPrimeAndDisplay(int n);
int main( )
{   int n;
    printf("Enter a positive integer: ");  scanf("%d",&n); // n is passed to the
    function checkPrimeAndDisplay(n);
 } // void indicates that no value is returned from the function void
checkPrimeAndDisplay(int n)
{    int i, flag = 0;
     for(i=2; i <= n/2; ++i)
     { if(n%i == 0)
        { flag = 1; break; }
      }
     if(flag == 1)
        printf("%d is not a prime number.",n);
    else
        printf("%d is a prime number.", n);
}
```

(D) With Arguments But A Return Value

```
#include <stdio.h>
int checkPrimeNumber(int n);
int main( )
{   int n, flag;
    printf("Enter a positive integer: "); scanf("%d",&n);
   flag = checkPrimeNumber(n);
   if(flag == 1)
       printf("%d is not a prime number",n);
   else
       printf("%d is a prime number",n);
    return 0;
} // integer is returned from the function
int checkPrimeNumber(int n)
{    int i;
     for(i=2; i <= n/2; ++i)
     {
      if(n%i == 0)      return 1;
     }   return 0; }
```

Parameter Passing to functions

- Two types of parameters: **Actual & Formal parameters.**

- The parameters that appear in function calls are called ***actual parameters***.

  For example: A = 10 and B = 20 are actual parameters.

- The parameters that appears in function declarations are called ***formal parameters***.

  For example: x and y are formal parameters.

- There are two ways to pass parameters:
  - **Pass by Value**
  - **Pass by Reference**

Parameter Passing to functions

- There are two most popular ways to pass parameters:

(a) Pass by Value: In this parameter passing method, copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

(b) Pass by Reference: In this parameter passing method, copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

PASS – BY – VALUE (EXAMPLE)

```
#include <stdio.h>
void swapnum( int , int );      // Function Declaration

int main( )
{   int num1 = 35, num2 = 45 ;
    printf("Before swapping: %d, %d", num1, num2);
    swapnum(num1, num2);      // Function Call
}
void swapnum( int var1, int var2)     //Function Definition
{   int tempnum ;

    tempnum = var1 ; /*Copying var1 value into temporary variable */
    var1 = var2 ;       /* Copying var2 value into var1*/
    var2 = tempnum ; /*Copying temporary variable value into var2 */
 printf("\n After swapping: %d, %d", var1, var2);
}
```

Output:

```
Before swapping: 35, 45
After swapping: 45, 35
```

PASS – BY – REFERENCE (EXAMPLE)

```
#include <stdio.h>
void swapnum( int * , int * )       // Function Declaration
int main( )
{    int num1 = 35, num2 = 45 ;
     printf("Before swapping:");
     printf("\n num1 value is %d", num1);
     printf("\n num2 value is %d", num2);
     swapnum( &num1, &num2 );      // Function Call
}
 void swapnum ( int *var1, int *var2)  // Function Definition
{   int tempnum ;
    tempnum = *var1 ;
   *var1 = *var2 ;
    *var2 = tempnum ;
    printf("\n After swapping:");
     printf("\n num1 value is %d", num1);
     printf("\n num2 value is %d", num2);
}
```

Output:

```
Before swapping: 35, 45
After swapping: 45, 35
```

Recursion

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.

- Using recursive algorithm, certain problems can be solved quite easily.
- Recursion is a powerful problem-solving technique that often produces very clean solutions to even complex problems.
- Recursive solutions can be easier to understand and to describe than iterative solutions.

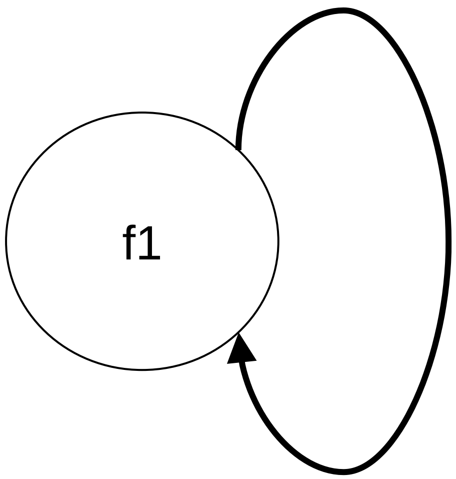
- Examples of such problems are:
    - Tower of Hanoi (ToH),
    - Inorder/Preorder/Postorder Tree Traversals,
    - DFS of Graph, etc.

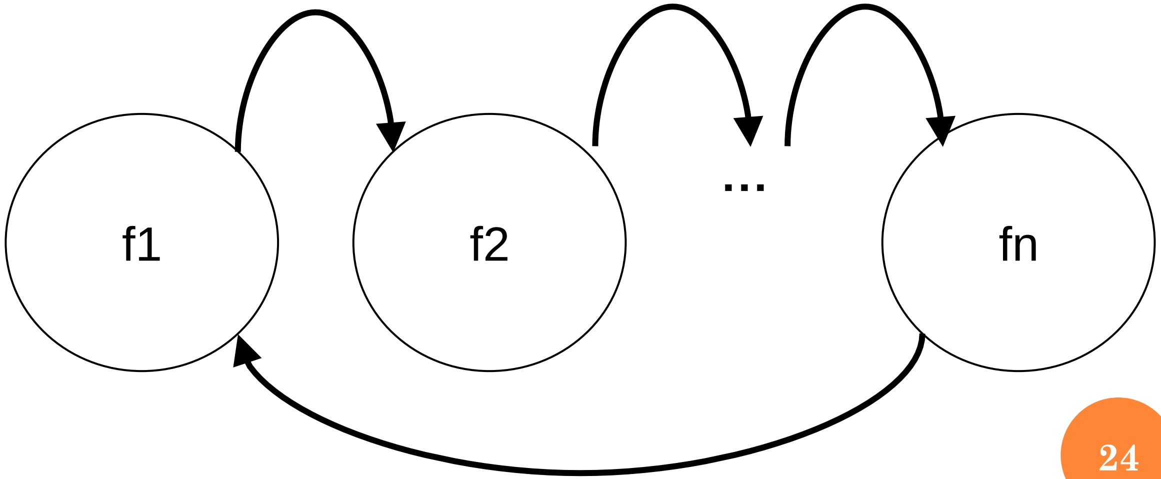
Iterative & Recursive Algorithms

Recursion:

- **Recursion means defining something, such as a function, in terms of itself.**
  - **For example, let *f(x) = x!***
  - **We can define *f(x)as:***

$$f(x) = \text{if } x < 2 \text{ then } 1 \text{ else } x * f(x-1)$$





Recursion Example

- **Sequences are functions from natural numbers to real:**

  f(i)= ai , where: i = 0, 1, 2, 3, ….. n

  a0, a1, a2, a3, …, an.

- **Example: Find *f(1), f(2), f(3), and f(4).***

  where, *f(0) = 1, and*

  f(n+1) = f(n)2+ f(n) + 1

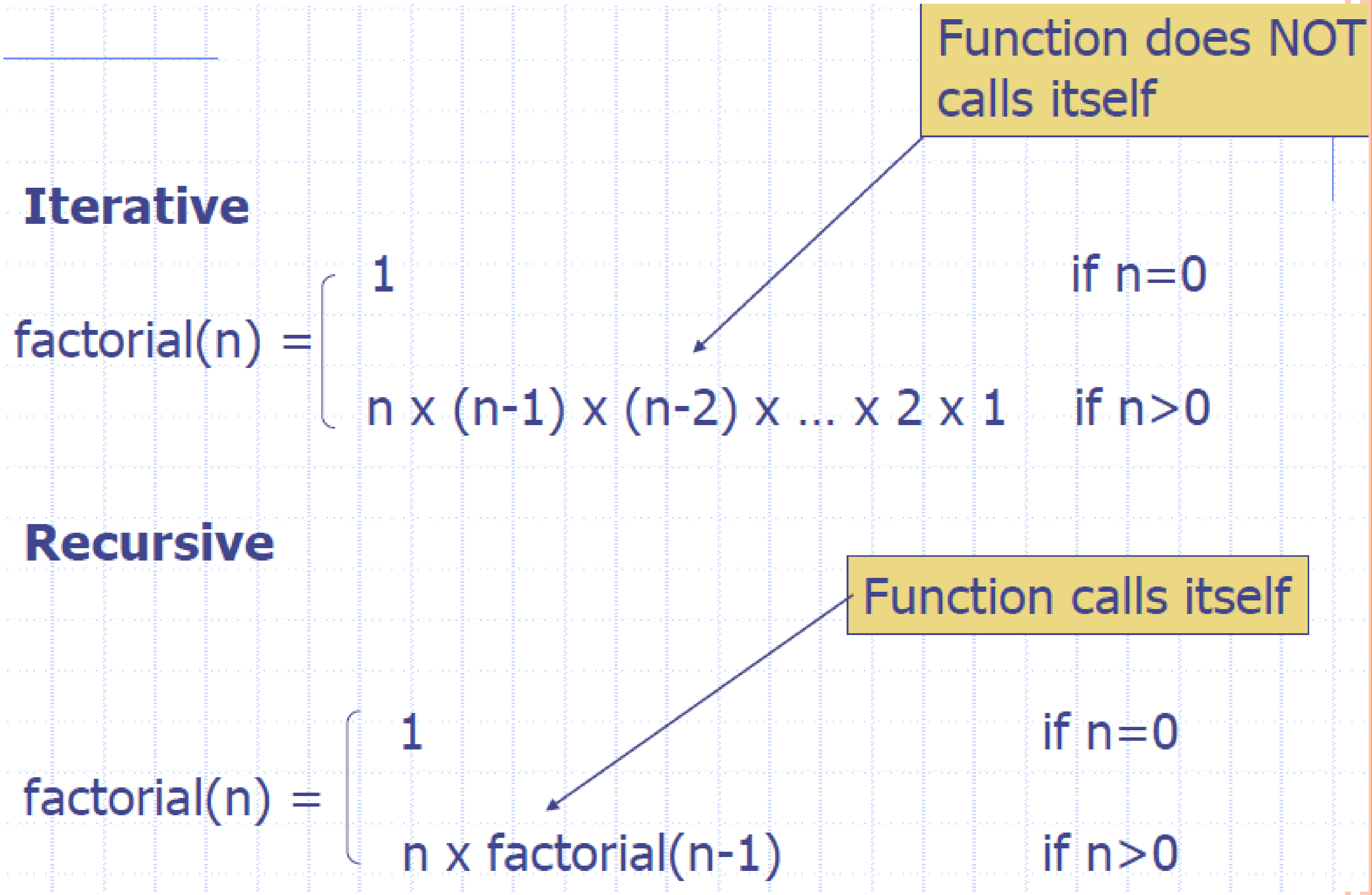
  f(1) = f(0)2+ f(0) + 1 = 12+ 1 + 1 = 3

  f(2) = f(1)2+ f(1) + 1 = 32+ 3 + 1 = 13

  f(3) = f(2)2+ f(2) + 1 = 132+ 13 + 1 = 183

  f(4) = f(3)2+ f(3) + 1 = 1832+ 183 + 1 = 33673

Iterative & Recursive Algorithms



Function does NOT calls itself

Iterative

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n=0 \\ n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 & \text{if } n>0 \end{cases}$$

Recursive

Function calls itself

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n=0 \\ n \times \text{factorial}(n-1) & \text{if } n>0 \end{cases}$$

Iterative Algorithm

```
factorial(n)
 {
  i= 1
  factN= 1
  while (i <= n)
      factN= factN* i
      i= i + 1
  return factN
 }
```

- The iterative solution is very straightforward.
- We simply loop through all the integers between 1 and n and multiply them together.
- In general, iterative solution is computed from small to big.

Recursive Algorithm

```
factorial(n)
 {
  if (n == 0)
     return 1
  else
    return n*factorial(n-1)
  end if
 }
```

Recursive Call

Note how much simpler the code for the recursive version of the algorithm is as compared with the iterative version.

We have eliminated the loop and implemented the algorithm with one 'if' statement.

Tracing Recursive Functions

- **Executing recursive algorithms goes through two phases:**
  - **Expansion in which the recursive step is applied until hitting the base step**
  - **"Substitution" in which the solution is constructed backwards starting with the base step**

```
factorial(4)  = 4 * factorial (3)
              = 4 * (3 * factorial (2))
              = 4 * (3 * (2 * factorial (1)))
              = 4 * (3 * (2 * (1 * factorial (0))))
```

Expansion phase

```
              = 4 * (3 * (2 * (1 * 1)))
              = 4 * (3 * (2 * 1))
              = 4 * (3 * 2)
              = 4 * 6
              = 24
```

Substitution phase

Iterative & Recursive for Factorial Program

Iterative

```
int factorial (int n)
{
int i, fact = 1;

if (n == 0)
   return (result);
else
{
    for (i=1; i<=n; i++)
        fact = fact * i;
}
return (fact);
}
```

Recursive

```
int factorial (int n)
{
int fact;


if (n == 0)
   return (1);
else
   fact = n * factorial (n-1);
return (fact);
}
```

Recursive Methods/Algorithms - TRY

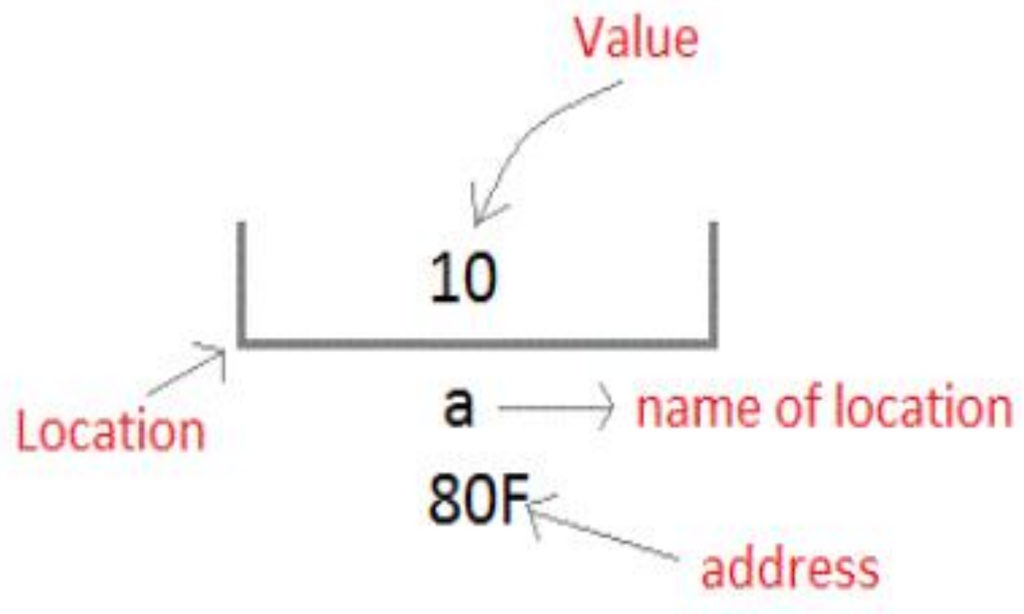
- **Factorial**
- **Fibonacci Series**
- **GCD**
- **Sum of Series**
- **Linear Search**
- **nCr**

Pointers

- **Pointer is a variable which holds the address of another variable of same data type.**

Let us assume that system has allocated memory location `80F` for a variable `a`.

```
int a = 10;
```



BENEFITS OF USING POINTERS

- Pointers are more efficient in handling Arrays and Structures.

- Pointers allow references to function and thereby helps in passing of function as arguments to other functions.

- It reduces length of the program and its execution time as well.

- It allows C language to support Dynamic Memory management.

Declaration of pointer variable

- **General syntax:**

```
datatype *pointer_name;
```

- **Pointer works with all data types:**

```
int *ip;          //pointer to integer variable
float *fp;        //pointer to float variable
double*dp;        //pointer to double variable
char *cp;         //pointer to char variable
```

INITIALIZATION OF POINTER VARIABLE

- **Pointer Initialization** is the process of assigning address of a variable to a **pointer** variable.

- **Example:**

```
#include <stdio.h>

void main( )
{
int a = 10;
int *ptr;             //Pointer declaration
ptr = &a;            // Pointer initialization
}
```

NULL POINTER

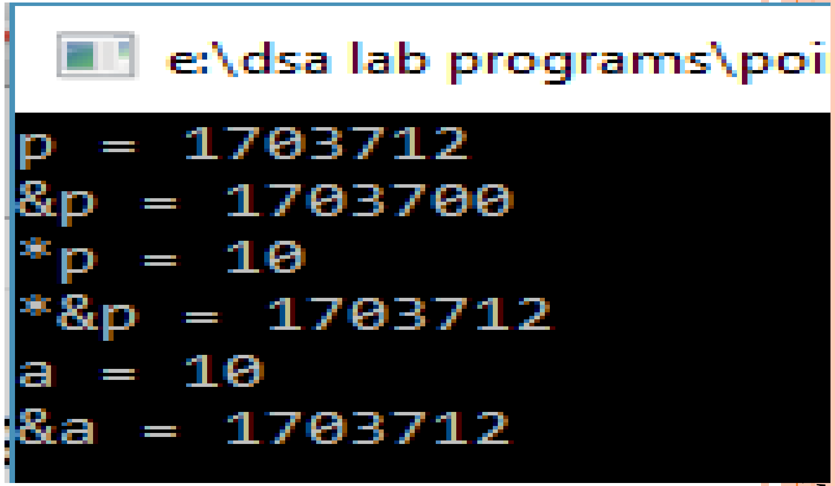
- If you are not sure about which variable's address to assign to a pointer variable while declaration, it is recommended to assign a NULL value to your pointer variable. A pointer which is assigned a NULL value is called a NULL pointer.

- Example:

```
#include <stdio.h>

int main( )
{
int *ptr = NULL;
return 0;
}
```

Pointers - Example program

```
#include <stdio.h>

void main()
{
int a = 10;
int *p;        //Pointer declaration
p = &a;       // Pointer initialization

printf("p = %d\n", p);           // prints the address of 'a'
printf("&p = %d\n", &p);       // prints the address of 'p'
printf("*p = %d\n", *p);        // prints value of that address
printf("*&p = %d\n", *&p); // prints value of that address
printf("a = %d\n", a);           // prints value of a
printf("&a = %d\n", &a);       // prints address of a
return 0;
}
```


e:\dsa lab programs\poi

```
p  =  1703712
&p  =  1703700
*p  =  10
*&p  =  1703712
a  =  10
&a  =  1703712
```

POINTS TO REMEMBER WHILE USING POINTERS

- While declaring/initializing the pointer variable, * indicates that the variable is a pointer.
- The address of any variable is given by preceding the variable name with symbol Ampersand ’&’.
- The pointer variable stores the address of a variable. The declaration int *a doesn't mean that ’a’ is going to contain an integer value. It means that a is going to contain the address of a variable storing integer value.
- To access the value of a certain address stored by a pointer variable, * is used. Here, the * can be read as 'value at‘.

Different Ways of assigning Pointers

(a) Pointer to Pointer (Double pointer)

(b) Pointer to Array

(c) Pointer to Functions

(A) Pointer to pointer (double pointer)

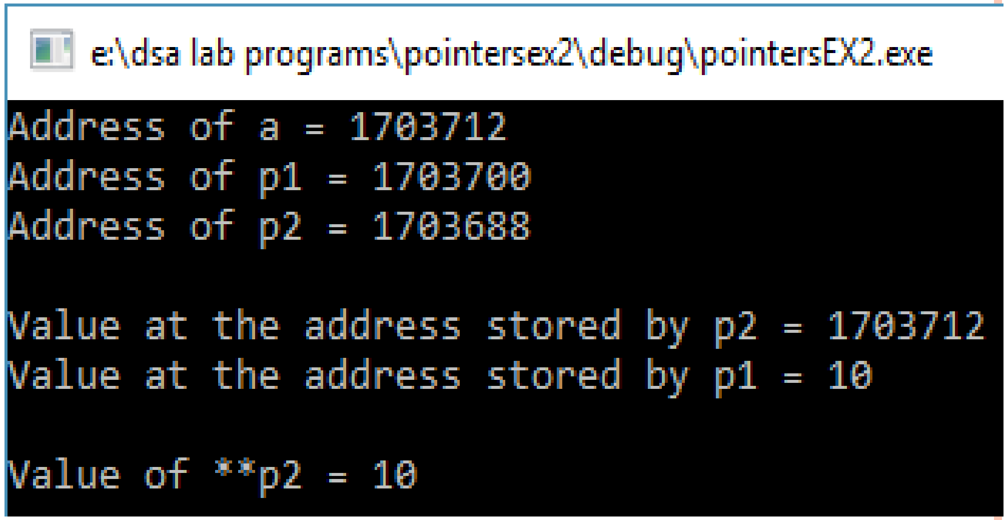
- When one pointer variable stores the address of another pointer variable, it is known as **Pointer to Pointer** variable or **Double Pointer**.

- **Syntax:**

```
Data_type **pointer_name;

int **p1;
```

(A) Pointer to pointer (Example program)

```
#include <stdio.h>
int main()
{
   int  a = 10;
   int  *p1;       //this can store the address of variable a
   int  **p2;     // this can store the address of pointer variable p1 only.

   p1 = &a;
   p2 = &p1;

   printf("Address of a = %d\n", &a);
   printf("Address of p1 = %d\n", &p1);
   printf("Address of p2 = %d\n\n", &p2);

 // below print statement will give the address of 'a'
   printf("Value at the address stored by p2 = %d\n", *p2);
   printf("Value at the address stored by p1 = %d\n\n", *p1);
   printf("Value of **p2 = %d\n", **p2); //read this *(*p2)

   return 0;
}
```


e:\dsa lab programs\pointersex2\debug\pointersEX2.exe

```
Address of a = 1703712
Address of p1 = 1703700
Address of p2 = 1703688

Value at the address stored by p2 = 1703712
Value at the address stored by p1 = 10

Value of **p2 = 10
```

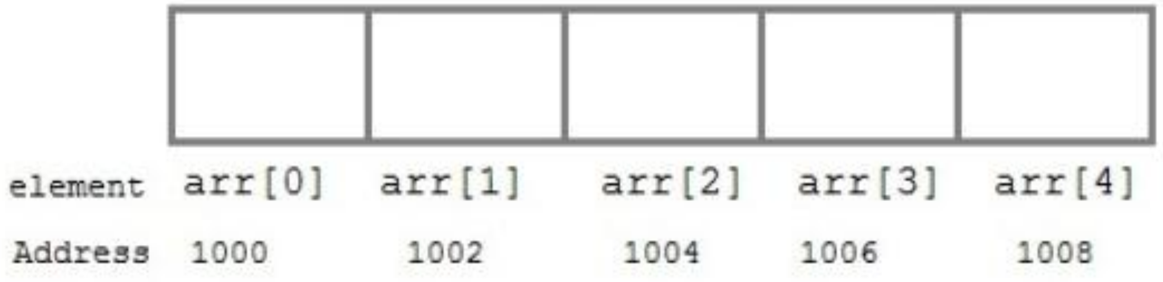
(B) POINTER TO ARRAY

- We can use a pointer to point to an array, and then we can use that pointer to access the array elements.

- Example:

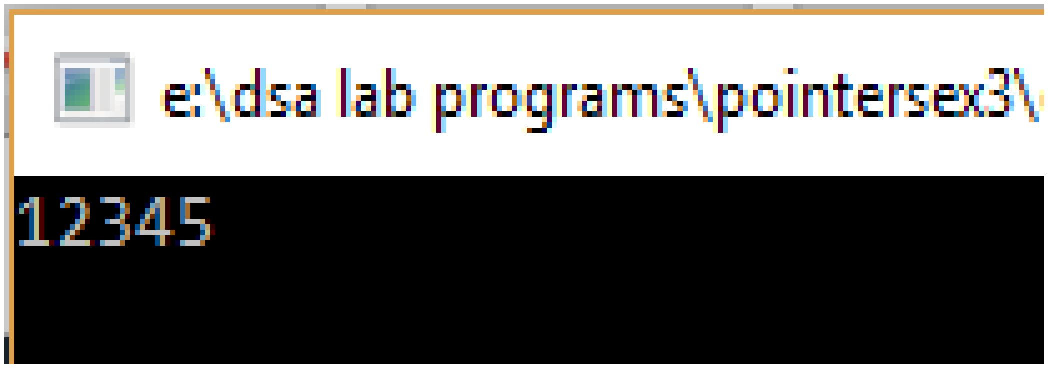
```
int *p;
p = arr;     (or) p = &arr[0]
```

- Assuming that the base address of arr is 1000 and each integer requires two bytes, the five elements will be stored as follows:



(B) POINTER TO ARRAY (EXAMPLE PROGRAM)

```
#include <stdio.h>

int main()
{
    int i;
    int a[5] = {1, 2, 3, 4, 5};
    int *p = a;     // same as int*p = &a[0]
    for (i = 0; i < 5; i++)
    {
        printf("%d\t", *p);
        p++;
    }

    return 0;
}
```



e:\dsa lab programs\pointersex3\

```
12345
```

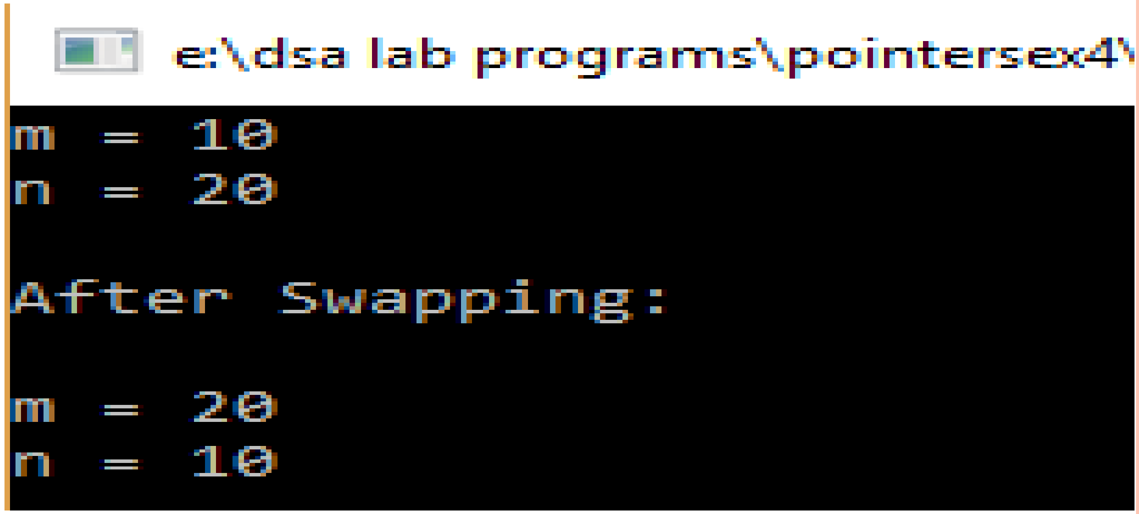
(D) POINTER TO FUNCTIONS

- Pointer as a function parameter is used to hold addresses of arguments passed during function call. This is also known as **call by reference**.

- When a function is called by reference any change made to the reference variable will effect the original variable.

(D) POINTER TO FUNCTIONS(EXAMPLE PROGRAM)

```
#include <stdio.h>
void swap(int *a, int *b);
int main()
{   int m = 10, n = 20;
    printf("m = %d\n", m);
    printf("n = %d\n\n", n);
    swap(&m, &n);    //passing address of m and n to the swap function
    printf("After Swapping:\n\n");
    printf("m = %d\n", m);
    printf("n = %d", n);
    return 0;
}
void swap(int *a, int *b)
{   int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```



e:\dsa lab programs\pointersex4\

```
m = 10
n = 20

After Swapping:

m = 20
n = 10
```