PTC UNIT - V

Part-1: Functions: Designing Structured Programs, Functions in C, User Defined Functions, InterFunction Communication, Standard Functions, Passing Array to Functions, Passing Pointers to Functions, Recursion

Part-2: Files I/O - Text Input / Output: Files, Streams, Standard Library Input / Output Functions, Formatting Input / Output Functions, Character Input / Output Functions

Files I/O - Binary Input / Output: Text versus Binary Streams, Standard Library, Functions for Files, Converting File Type.

Functions in C

Designing Structured Programs in C

Structured programming is a **modularization** or a **parsing** technique that divides a larger program into smaller subprograms. The 3 methods of parsing or dividing are

1. **Top-down:**
   Breaks down a large problem into smaller "tasks" namely functions and sub-functions.
2. **Bottom-up:**
   Starts with basic building blocks (like Lego) and builds smaller code units. These lower functions work together and emerge to solve the bigger problem.
3. **Middle-out:**
   The best strategy is to design a program by combining and using both methods at different levels of parsing.

Designing Structured programming makes a program

- easy to understand,
- easy to implement,
- easy to test and debug,
- easy to improve quality, and
- code reusable.

Structured programming enables code reusability, which means writing code once and using it many times.

In C, **Top-Down structured programming** can be designed **using the concept of the functions**. Using the concept of the functions, we can divide the larger program into smaller function modules. These functions are executed individually.

A C program is written with one or more functions. The main() function is mandatory to execute a C program. The execution of the program always starts with **main()**, but it can also **call other functions to perform specific tasks**.

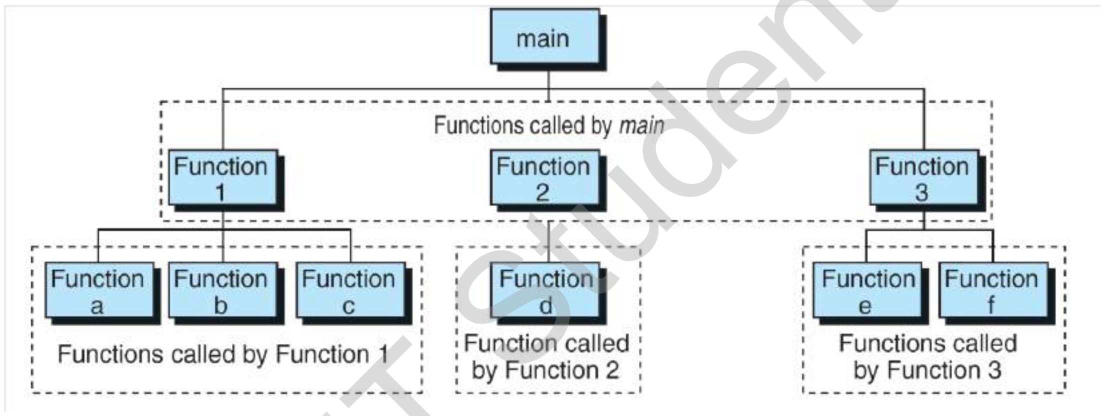
Definition of a Function:

A function is a set of statements used to perform a specific task and is executed individually. A function is a small part of a program.

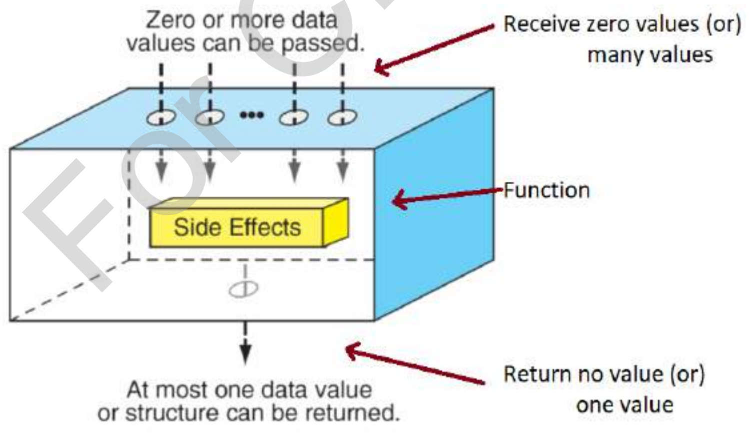
The **purpose** of a function is

- to receive zero or more data values,
- operate on those data values, and
- return at most one data value as a result.

Chart using Functions in C:



Function Concept:


Zero or more data values can be passed.
Receive zero values (or) many values
Side Effects
Function
At most one data value or structure can be returned.
Return no value (or) one value

Note: A function can have a return value, a side effect, or both. The side effect occurs before the value is returned from the function. (Ex: **return (x=y++);** //x value is returned, but y value also changed as a side effect).

The functions are of 2 types.

Types of Functions	
A. User-Defined Functions	B. Standard C Library Functions

A. User-Defined Functions:

Users can define their own functions in a C program. The functions that are created by users are called user-defined functions.

- A function is a block of code used to perform a specific task.
- A function runs only when it is called.
- We can pass data into functions as parameters.
- Functions can be reused; write the code once, and use it many times.

The user-defined functions in C consist of the following:

1. **Function Declaration** (or **Function Prototype) -** tells the compiler about a function's name, return type, and parameters.
2. **Function Definition -** provides the actual body of the function.
3. **Function Call -** can be done from main(), another function, or by itself to execute the called function.
4. A Function in C must be Declared and Defined before the Function Call.

Optimized Layout of C Program using Functions: A function declaration should be before the main() and a function Definition should be after main() for an optimized C program. (A function may also be declared inside the main function or in any other function which is not optimized).

```
#include<stdio.h>
Function Declarations
```

```
int main()
{  Local declaratons
   Function calls
   Statements
}
```

```
Function Definitions
{
  Local declarations
  Statements
}
```

1. Function Declaration

The **function declaration** is also called a **function PROTOTYPE**. A function prototype consists of

- **name of the function**
- **data type of the return value and**
- **list of parameters.**

Declaration Syntax:

```
returnType   functionName(parameters list);
```

returnType specifies the data type of the value which is sent out as a return value from the function definition.
functionName is a user-defined name used to identify the function uniquely in the program.
parametersList is the data values that are sent to the function definition.

Ex: **int multiply(int x, int y);**

2. Function Definition

The function definition provides the actual code of that function. The function definition is also known as the **body of the function**. The function definition means the actual statements to be executed by a function. These statements of a function are written inside the curly braces "{ }". The function definition is executed only when it is called from other functions or by itself.

Function Definition Syntax:

```
return_type   function_name( parameter list )
{
  body of the function
}
```

Ex: function definition

```
int multiply(int x, int y)
{ int result;
  result = x * y;
  return result;
}
```

3. Function Call

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

A function is called either from the main function or from other functions or also by itself.

Function Call Syntax:

```
functionName(parameters);
```

Example: Program to multiply two integers using functions.

```
#include <stdio.h>
/* function declaration */
int multiply(int x, int y);

int main () {
   /* local variable definition */
   int a = 10;
   int b = 20;
   int result;

   /* calling a function to get multiplicaton value */
   result = multiply(a, b);

   printf( "Multiplied value is : %d\n", result);

   return 0;
}

/* function returning the multiplication of two numbers */
int multiply(int x, int y)
{
   /* local variable declaration */
   int mult;
   mult = x * y;
   return mult;
}
```

Output: Multiplied value is : 200

Parameter Passing in Functions

When the execution control is transferred from **calling function** to **called-function,** it may carry none or many data values. The data that are passed from the calling function to called function are called **actual parameters**.

Parameters are the data values that are passed from the calling function to called function.
* At a function call, the copy of actual parameter values are copied into formal parameters.

In C, there are two types of parameters.

- **Actual Parameters (or Arguments)**
- **Formal Parameters**

The **actual parameters** are the parameters that are specified in the **calling function**.
The **formal parameters** are the parameters that are declared in the **called function**.
When a function is called, a COPY of Actual parameter values is copied into Formal parameters.

Actual Parameters vs Formal Parameters:

Actual Parameters (or Arguments)	Formal Parameters
Actual parameters are passed from the CALLING function. These are also called **Arguments**.	**Formal parameters** are declared in the CALLED function definition. These are also called **Parameters**.
Actual parameters are SENDING parameters to called (receiving) function from the calling (sending) function. Ex: in main()	Formal parameters are declared in the function definition for RECEIVING values from Actual Parametters..
During the time of call, each argument is always assigned to the parameter in the function definition.	Parameters are local variables that are assigned values of the arguments when the function is called.
Must match ● exact data type ● number of parameters ● order of parameters	Must match ● exact data type ● number of parameters ● order of parameters
You may have the same or different names for Formal and Actual parameters. **Ex:** **int q; float p;** **cost(q, p); //function call**	You may have the same or different names for Formal and Actual parameters. **Ex:** **float cost(int qty, float price);**

Example: Program to add two integers using a function

```
#include<stdio.h>
int add(int a, int b);  //Declaration of Function

int main()
{int x=10, y = 20, total;
 total = add(x,y);   //x and y are ACTUAL parameters
 printf("Total of x and Y = %d", total);
 return 0;
}
//Definition of Function
int add(int a,int b) //a and b are FORMAL parameters
{ int sum;
  sum = a + b;
  return sum;       //Sends value back to source
}
```

In the concept of functions, the function call is known as **"Calling Function"** and the function definition is known as **"Called Function"**.

When we make a function call, the execution control jumps from the calling function to called function. After executing the called function, the execution control comes back to the calling function from the called function. When the control jumps from calling function to called function it may carry one or more data values called "**Parameters**" and while coming back it may carry a single value called "**return value**". That means the data values transferred from the calling function to called function are called **Parameters** and the data value transferred from called function to the calling function is called **Return value**.

Based on the data flow between the calling function and called function, the functions are classified into 4 categories as described below.

Advantages Of Functions

- We can implement modular programming using functions
- Larger programs can be divided into smaller modules using functions
- Functions make the program easy to read and understand
- Once a function is created it can be used many times (code re-usability)

Inter-Function Communication

The exchange of information using arguments between two functions is called an Inter-Function Communication.

A Function Call in the program transfers the execution control from Calling a Function to Called Function and executes the function definition. After the function is executed, the control finally comes back to the Calling Function.

In this process, both Calling and Called functions exchange information. The process of exchanging information between Calling and Called functions is called Inter-Function Communication.

What are the Categories of Functions?

The Inter-Function Communication in C is categorized into 4 categories as follows. The 4 categories of functions are divided based on **Parameter Passing** and **Return value.**

1. **Bi-directional Communication**
2. **Downward Communication**
3. **Upward Communication**
4. **No Exchange Communication**

Category	Pass Parameters	Return Value	Example
1. Bi-Directional	Yes	Yes	int add(int, int)
2. Downward	Yes	No	void add(int, int)
3. Upward	No	Yes	int add( )
4. No Transfer	No	No	void add( )

1. Bi-Directional Function: Pass Parameters - YES & Return Value - YES

When these types of functions are used,

- Calling-functions send parameters to called function, and
- Called function returns value back to the calling function.
- Execution control jumps from the calling function to called function along with parameters and executes called function, and
- Finally comes back to the calling function along with a return value.

Example:

```
// Function category: Passing arguments into function & Returns value
from function
#include <stdio.h>
int add(int, int);
void main() {
  int a=10, b=20;
  printf("Addition of 2 numbers = %d\n", add(a, b));
}
int add(int x, int y)
{
  return (x+y);
}
```

Output: Addition of 2 numbers = 30

2. Downward: Pass Parameters - YES & Return Value - NO

When these type of functions are used,

- Calling-functions send parameters to called-function and
- Called function DOES NOT return value back to calling-function.
- Execution control jumps from calling function to called function along with parameters and executes called function, and
- Finally comes back to the calling function WITHOUT a return value.

Example:

```
// Function category: Passing arguments into function & NO Return
value from function
#include <stdio.h>
void add(int, int);   //Function Declaration
void main() {
  int a=10, b=20;
  add(a, b);     //Function Call
}
```

```
void add(int x, int y)    //Function Definition
{  printf("Addition of 2 numbers = %d\n", x+y);
   //NO return value from function
}
```

Output: Addition of 2 numbers = 30

3. Upward: Pass Parameters - NO & Return Value - YES

When these type of functions are used,

- Calling-functions DOES NOT send parameters to called-function and
- But the Called function returns value back to calling-function.
- Execution control jumps from calling function to called function with NO parameters and executes called function, and
- Finally comes back to the calling function with a return value.

Example:

```
// Function category: Passing NO arguments into function & YES Returns
value from function
#include <stdio.h>
int add();  //Function Declaration
void main()
{ int result;
  result = add();     //Function Call
  printf("Addition of 2 numbers = %d\n", result);
}
int add()   //Function Definition
{   int x=10, y=20;
    return (x+y);
}
```

Output: Addition of 2 numbers = 30

4. No Transfer: Parameters - NO & Return Value - NO

When these type of functions are used,

- Calling-functions DOES NOT send parameters to called-function and
- Called function also DOES NOT return value back to calling-function.
- Execution control jumps from calling function to called function with NO parameters and executes called function, and
- Finally comes back to the calling function with NO return value.

Example:

```
// Function category: Passing NO arguments into function & NO Return value
from function
#include <stdio.h>
void add();  //Function Declaration
void main() {
  add();     //Function Call
}
void add()   //Function Definition
{
   int x=10, y=20;
   printf("Addition of 2 numbers = %d\n", x+y);
}
```

```
Output: Addition of 2 numbers = 30
```

B. Standard C Library Functions

The library of C Programming Language comes with many pre-defined functions. These functions make programming easier. These pre-defined functions are known as Library Functions or Standard Functions or System Defined functions.

The functions that are defined within the C library are called standard or system-defined functions.

All the standard functions are defined inside the header files such as stdio.h, conio.h, math.h, string.h, etc. Ex: The functions **printf()** and **scanf()** are defined in the header file called **stdio.h**.

Whenever we use standard functions in the program, we must include the respective header file using #include statement. Ex: If we use a standard function **sqrt()** in the program, we must include the header file called **math.h** because the function **sqrt()** is defined in **math.h.**

Header file	Description	Standard C Functions (few)
stdio.h	This is standard input/output header file	**scanf(), printf(), fscanff(), fprintf(), fopen(), fclose(), fread(), fwrite(), fseek(), ftell(), rewind(), remove(),** rename(), feof(), ferror(), clearerr(), freopen()
conio.h	This is console input/output header file	**clrscr(), getch()**
string.h	All string related functions are defined in this header file	**strlen(), strcpy(), strupr(), strlwr()**
stdlib.h	This header file contains general functions used in C programs	**atof(s), atoi (s), atol(s)** • converts character to float,integer and long respectively
math.h	All maths-related functions are defined in this header file	**sqrt(), abs()**, sin(), cos(), exp(), log(), **pow()**, ceil(), floor()
time.h	This header file contains time and clock related functions	**clock(), time()**, localtime(),
ctype.h	All character handling functions are defined in this header file	**isalnum(c), isalpha(c), isdigit (c)**, iscntrl (c) , ispunct(c) punctuation , isspace(c) space, tab or new line isupper(c) 'A'-'Z'
stdarg.h	Variable argument functions are declared in this header file	va_start(), va_arg() and va_end()
locale.h	Sets or reads location dependent information	setlocale(), localeconv()

errno.h	Error handling functions are given in this file	extern int **errno**
graphics.h	Provides functions to draw graphics.	**circle(), rectangle()**
float.h	Provides constants related to floating point data values	
stddef.h	Defines various variable types	

Example: Program using standard C mathematical functions

```
#include<stdio.h>
#include <math.h>
#define PI 3.1415
int main(){
printf("\nCeiling = %f",ceil(3.6));
printf("\nFloor = %f",floor(3.2));
printf("\nSquare root = %f",sqrt(7));
printf("\nPower = %f",pow(3,3));
printf("\nAbsolute = %d",abs(-12));
printf("\nRemainder = %.0f",fmod(7,2));

double x, fractpart, intpart;
x = 7.54321;
//returns the fraction part and sets integer to the integer part
fractpart = modf(x, &intpart);
printf("\nInteger part = %lf", intpart);
printf("\nFraction Part = %lf", fractpart);

   double d, deg, val;
   d = 60.0;
   val = PI / 180.0;
   deg = cos( d*val );
   printf("\nThe cosine of %lf is %lf degrees", d, deg);
return 0;     }
```

Output:

```
Ceiling = 4.000000
Floor = 3.000000
Square root = 2.645751
Power = 27.000000
```

```
Absolute = 12
Remainder = 1
Integer part = 7.000000
Fraction Part = 0.543210
The cosine of 60.000000 is 0.500027 degrees
```

Methods of Function Call:

There are two methods to call a function by passing parameters from calling function to called function.

1. **Call by Value**
2. **Call by Reference** (also called Passing Pointer to Function)

1. Call by Value

In this method, the actual parameter values are COPIED to formal parameters. These duplicated formal parameters are used in the called function. The changes made to the formal parameters do not affect the values of the actual parameters.

That means, after the execution control comes back to the calling function, the actual parameter values remain the same.

Example: Program for Function Call by Value - Swap Numbers

```
#include <stdio.h>
void swap( int p, int q);
int main( )
{
int a=35, b=45 ;
printf("Before swapping: %d, %d", a, b);

swap(a, b);     //calling function BY VALUE or copies of values
printf("\nAfter swapping: %d, %d", a, b);
}
void swap( int p, int q)    //called function
{   int tmp;
    tmp = p;
    p = q;
    q = tmp;
}
```

Output:

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

In the above example program, the variables a and b are called actual parameters and the variables p and q are called formal parameters. The value of a is copied into p and the value of b is copied into q. The changes made on variables p and q do not affect the values of a and b.

2. Call by Reference

In the call by reference method, the **memory addresses of the actual parameters are passed to called function and received by the formal parameters (pointers)**. This address is used to access the memory locations of the actual parameters in the called function. Here, the formal parameters must be pointer variables.

Whenever we use these formal parameters in the called function, they directly access the memory locations of actual parameters. So the changes made to the formal parameters affect the values of actual parameters.

Example: Program for Function Call by Reference - Swap Numbers

```
#include <stdio.h>
void swap( int *p, int *q);

int main( )
{
int a=35, b=45 ;
printf("Before swapping: %d, %d", a, b);

swap(&a, &b);    //calling function BY REFERENCE or sending addresses
printf("\nAfter swapping: %d, %d", a, b);
}

void swap( int *p, int *q)   //called function
{  int tmp;
   tmp = *p;
   *p = *q;
   *q = tmp;
}
```

Output:

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

In the above example program, the addresses of variables **a** and **b** are copied to pointer variables **p** and **q**. The changes made on the pointer variables **p** and **q** in called function affect the values of actual parameters **a** and **b** in the calling function.

Summary of Actual Parameters (Arguments) vs Formal Parameters:

Function Call by Value
- Actual Parameters are Copied to the Formal Parameters
- Changes to Formal Parameters in function don't affect Actual Parameters.

Function Call by Reference
- Address of Actual Parameters is Passed to the Formal Parameters (POINTERs)
- Changes to Formal Parameters affect the value of Actual Parameters.

Scope of C Variable:

A scope is an area of a program where a variable can be accessed after its declaration.

Every variable in C is defined in scope. Scope is the section or region of a program where a variable has its life, and that variable cannot be used or accessed outside that region.

A variable declared within a function is different from the variable declared outside of a function. The variable can be declared in 3 places. They are

Type	Position	Access
1. **Global Scope Variables**	Before main() function. Out of all functions. Accessible in all functions.	Accessible to all functions.
2. **Local Scope Variables** ● Local to a Function ● Local to a Block { }	Declaration of variables inside a function or inside a block **{ }**.	Accessible within a function or a block { }.
3. **Local Scope Formal Parameters** ● Local to a Function	In the header of the function definition as formal parameters.	Accessible within a function.

1. **Global Scope (Global Declaration Before the function definition)**

Variables declared outside the function block and accessed inside the function are called global variables.

Global Scope

- The global variables are defined outside a function or any specific block on top of the C program. These variables hold their values throughout the program
- These are accessible in all functions defined in your program.
- Any function can access variables defined within the global scope, i.e., its availability stays for the entire program after being declared.

2. **Local Variables**

The variables that are declared within a function block are used only within that function. Those variables are not accessible outside that function. These are called local variables.

Local Function Scope or Block Scope

- A local scope or block is a collective program statement placed and declared within a function or block within curly braces.
- Variables lying inside such blocks are called local variables to those blocks. All these locally scoped statements are written and enclosed within the curly braces { }.
- C also has a provision for nested blocks, which means that a block or function can occur within another block or function. That means, the variables declared within a parent block can be accessed in the nested block and all other internal blocks. However, these variables cannot be accessed outside the parent block.

Example: Proggram to show Global and Local Scope Variables

```
#include<stdio.h>
int num1, num2;  //Declaration of Global Variable
void add();
void sub();
void mult();
int main(){
   num1 = 10 ;  //Initialization of Global Variable
   num2 = 20 ;  //Initialization of Global Variable
   printf("num1 = %d, num2 = %d \n", num1, num2) ;
   add();
   sub();
   mult();
   return 0;    }
```

```
void add()
{
   int result ;   //Local to add()
   result = num1 + num2  ; //accessing Global variables
   printf("\nAddition = %d", result)  ;
}
void sub()
{
   int result ;   //Local to sub()
   result = num1 - num2  ;  //accessing Global variables
   printf("\nSubtraction = %d", result)  ;
}
void mult()
{
   int result ;   //Local to mult()
   result = num1 * num2  ; //accessing Global variables
   printf("\nMultiplication = %d", result)  ;
}
```

Output:

```
num1 = 10, num2 = 20
Addition = 30
Subtraction = -10
Multiplication = 200
```

3. Local Scope Formal Parameters in Function Definition:

The variables declared in the function definition as parameters have a local variable scope. These variables behave like local variables in the function. They can be accessed inside the function; they are not accessible outside the function.

Example: Program to show Local Scope Formal Parameter in Function

```
#include<stdio.h>
void addition(int, int)  ;

int main(){
   int num1, num2  ;
   num1 = 10  ;
   num2 = 20  ;
   addition(num1, num2)  ;
   return 0;     }
```

```
void addition(int a, int b)
{  //a and b are Local Formal Parameters
   int resultSum ;
   resultSum = a + b ;
   printf("\nAddition = %d", resultSum) ;
}
```

Output: Addition = 30

Note: Here, the variables **a** and **b** are declared in function definition as parameters. So, they can be used only inside the addition() function.

Passing Arrays to Function:

An array is a collection of similar data types which are stored in memory as a contiguous memory block.

We can pass the address of an array to a function argument. For example, we have a function to sort a list of numbers; it is more efficient to pass these numbers as an array to function instead of passing them as variables. Since the number of elements the user has is not fixed and passing numbers as an array will allow our function to work for any number of values.

Since arrays are a continuous block of values, we can pass the reference of the first memory block of our array to the formal parameter in the function. This receiving parameter can be an array variable or can also be a pointer variable. And then we can easily calculate the address of any element in the array using the formula:

address(arr[i]) = (start address of array) + i * (size of individual element)

(see example-2)

Note: Name of the array acts as a pointer because it points to address of the first element of the array.
Ex: int arr[10]; here, the name "**arr**" and "**&arr[0]**" both refer to the address of the first value in the array.
So, we can send an Actual parameter such as "**arr**" to a formal parameter of either "**a[ ]**" or "***a**" as both will receive the address.

Example 1 of 2: Program to Pass an Array to Function and Access the Array

```
//Pass arrays to function
#include<stdio.h>
void read(int[], int); //No need to specify parameter names in Declaration
void display(int[], int); //No need to specify parameter names in Declaration
void main() {
  int qty[10], n, j;
  printf("Enter number of products : ");
  scanf("%d", &n);
  read(qty, n);
  display(qty, n);
}
void read(int r[10], int n) {
  int i;
  printf("\nEnter Quantity Sold for %d Products : \n", n);
  for(i = 0; i < n; i++) {
    printf("Enter Quantity Sold for Product x[%d] : ", i);
    scanf("%d", &r[i]);
  }
}
void display(int d[10], int n) {
  int i;
  printf("\nThe Sold Quantity are : \n");
  for(i = 0; i < n; i++) {
    printf("Product d[%d] : %d\n", i, d[i]);
  }
}
```

Output:

```
Enter number of products : 3

Enter Quantity Sold for 3 Products :
Enter Quantity Sold for Product x[0] : 100
Enter Quantity Sold for Product x[1] : 200
Enter Quantity Sold for Product x[2] : 300

The Sold Quantity are :
Product y[0] : 100
Product y[1] : 200
Product y[2] : 300
```

Example 2 of 2: Program to Pass an Array to Function and Access with Pointer Arithmetic

```
//Pass arrays to function parameter pointer (with pointer arithmetic)
#include<stdio.h>
void readScore(int *s, int size);
void displayScore(int *s, int size);
int totalScore(int *s, int size);
int main()
{
  int n = 11;  //max number of players
  int scores[n], total;
  readScore(scores, n); //”scores” is starting address of the array
  displayScore(scores, n);
  total = totalScore(scores, n);
  printf("\n Total = %d",total);
return 0;
}
void readScore(int *s, int size)
{
  int i;
  printf("Enter 11 scores: \n");
  for(i = 0; i < size; i++) {
    printf("Player %d = ", i+1);
    scanf("%d", (s + i));
  }
}
void displayScore(int *s, int size)
{
  int i;
  for(i = 0; i < size; i++)
  {
    printf("\nPlayer %d = %d ",i+1, *(s + i));
  }
}

int totalScore(int *s, int size)
{
  int i, sum=0;
  for(i = 0; i < size; i++)
  {
    sum += *(s + i);
  }
return sum;
}
```

Output:

```
Player 1 = 10
Player 2 = 40
Player 3 = 50
Player 4 = 10
Player 5 = 30
Player 6 = 60
Player 7 = 25
Player 8 = 25
Player 9 = 25
Player 10 = 25
Player 11 = 50

 Total = 350
```

Passing Pointers to Function:

Passing the pointers to the function means the memory location of the variables is passed to the parameters in the function, and then the operations are performed. The function definition accepts these addresses using pointers.

Example: Program to Pass Pointer to a Function (Or You may also use the swap() program given above under the “call by reference” section as that too will pass a pointer to the function).

```
#include <stdio.h>
void increment(int* ptr) {
  (*ptr)++; // adding 1 to *ptr variable value
}
void decrement(int* ptr) {
  (*ptr)--; // subtracting 1 from *ptr variable value
}

int main()
{
  int* p, i = 10;
  p = &i;
  increment(p);
  printf("%d", *p); // prints 11

  decrement(p);
  printf("%d", *p); // prints 10
  return 0;
}
Output:    11    10
```

Recursive Functions in C

In C, function calls can be started,

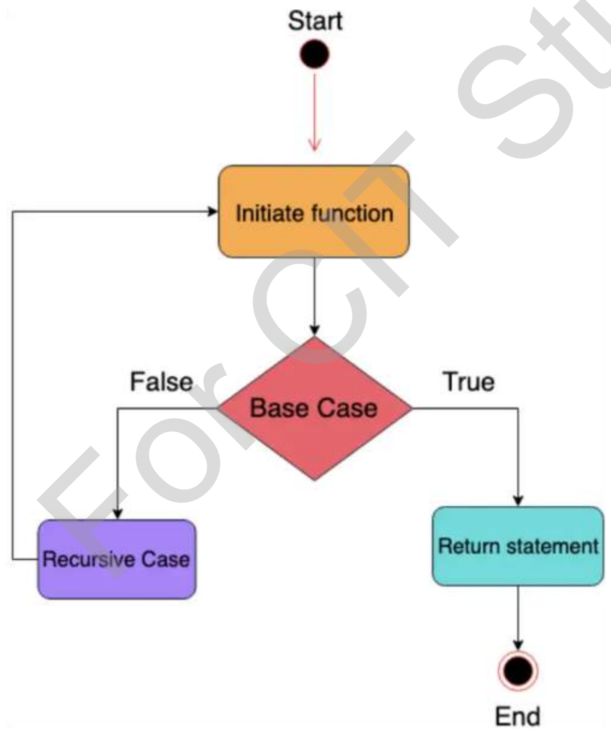
- from the main() function,
- from another function, or
- **from the same function itself.**

Definition of Recursive Function:

A function called by itself repetitively is called a recursive function. The function call is termed a recursive call.

The recursive function will call itself multiple times until a condition is satisfied. The recursive functions should be used very carefully because, when a function is called by itself it enters into the infinite loop. And when a function enters into the infinite loop, the function execution never gets completed. We should define the condition to exit from the function call so that the recursive function gets terminated.

Flowchart of Recursive Function:



The recursive function has two main parts in its body,

1. **the base case** (condition) and
2. **the recursive case (recursive function call).**

The flow of execution of Recursive Function:

- first, the program checks the base case condition.
- If it is TRUE, the function returns and quits;
- otherwise, the recursive case is executed by calling the function recursively.

The representation of recursion in a C program is as follows.

```
recursive_function()
{
    //base case condition
    if base_case = true;
       return;
    else
    //recursive case
       return recursive_function(argument); // recursive call
}
```

Note: When a function is called by itself, the first call remains under execution till the last call is invoked. Every time when a function call is invoked, the function returns the execution control to the previous function call.

Types of Recursion in C

There are two types of recursion in the C language.

1. **Direct Recursion**
2. **Indirect Recursion**

1. Direct Recursion

Direct recursion in C occurs when a function calls itself directly from inside. Such functions are also called direct recursive functions.

Structure of direct recursion:

```
function1()
{   //some code
    function1();
    //some code
}
```

In the direct recursion structure, the `function1()` executes, and from inside, it calls itself recursively.

Example 1 of 2: Program to find Factorial of an integer using Direct Recursion Function

```
#include<conio.h>
int  factorial( int ) ;
int main()
{
    int  fact, n ;
    printf("Enter a positive integer: ") ;
    scanf("%d", &n) ;
    fact = factorial( n ) ;
    printf("\nFactorial of %d is %d\n", n, fact) ;
    return 0;
}
int  factorial(int n)
{   int  temp ;
    if(n <= 1)
          return  1 ;
    else
          temp = n * factorial( n-1 ) ; // recursive function call
    return  temp ;
}
```

Output:

```
Enter a positive integer:  3
Factorial of 3 is 6
```

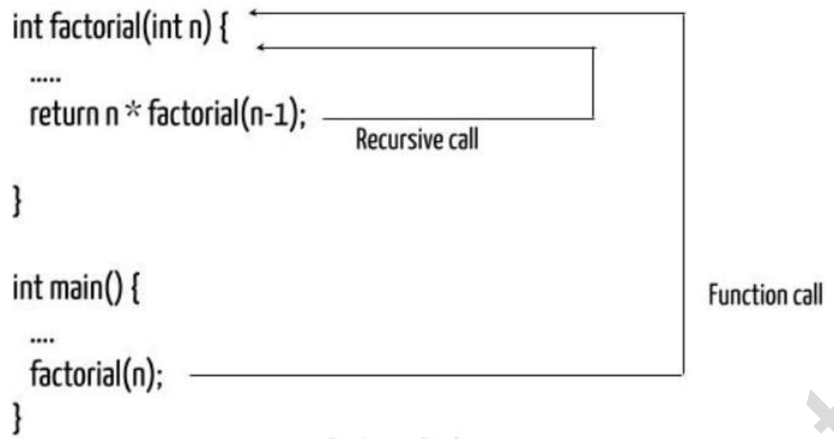
Explanation:

In the above example program, the factorial() function call is initiated from main() function with the value 3. Inside the factorial() function, the function calls factorial(2), factorial(1), and factorial(0) are called recursively.

In this program execution process,

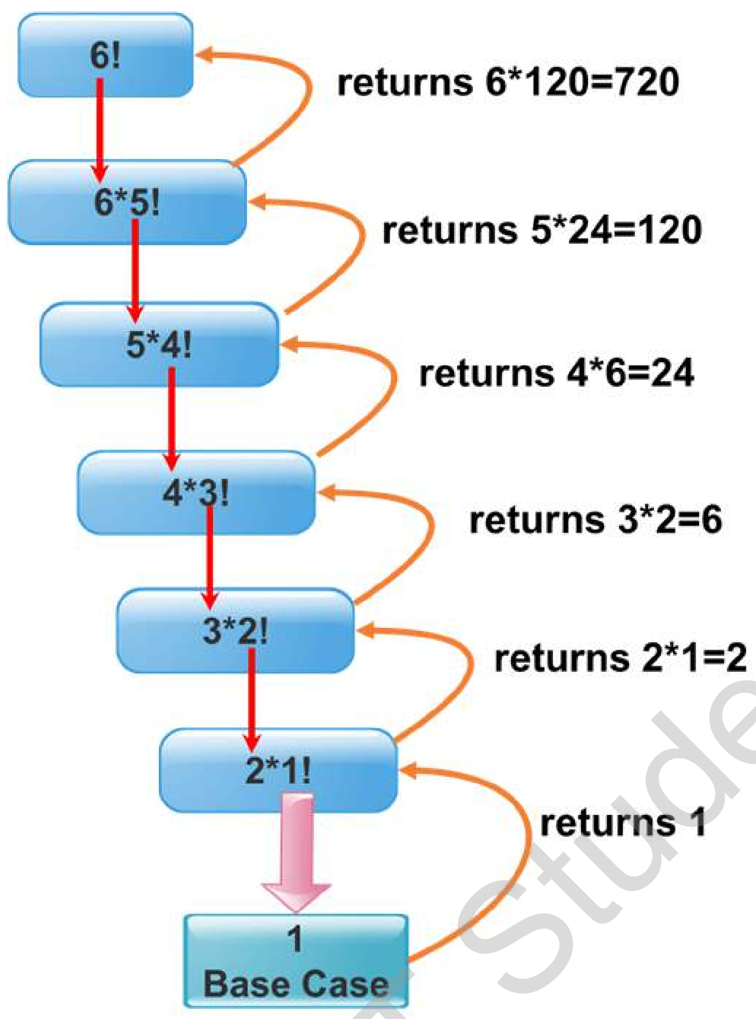
- the function call factorial(3) remains under execution till the execution of function calls factorial(2), factorial(1), and factorial(0) gets completed.
- Similarly the function call factorial(2) remains under execution till the execution of function calls factorial(1) and factorial(0) gets completed.
- In the same way the function call factorial(1) remains under execution till the execution of function call factorial(0) gets completed.

The following figure shows the complete execution process of the above program.

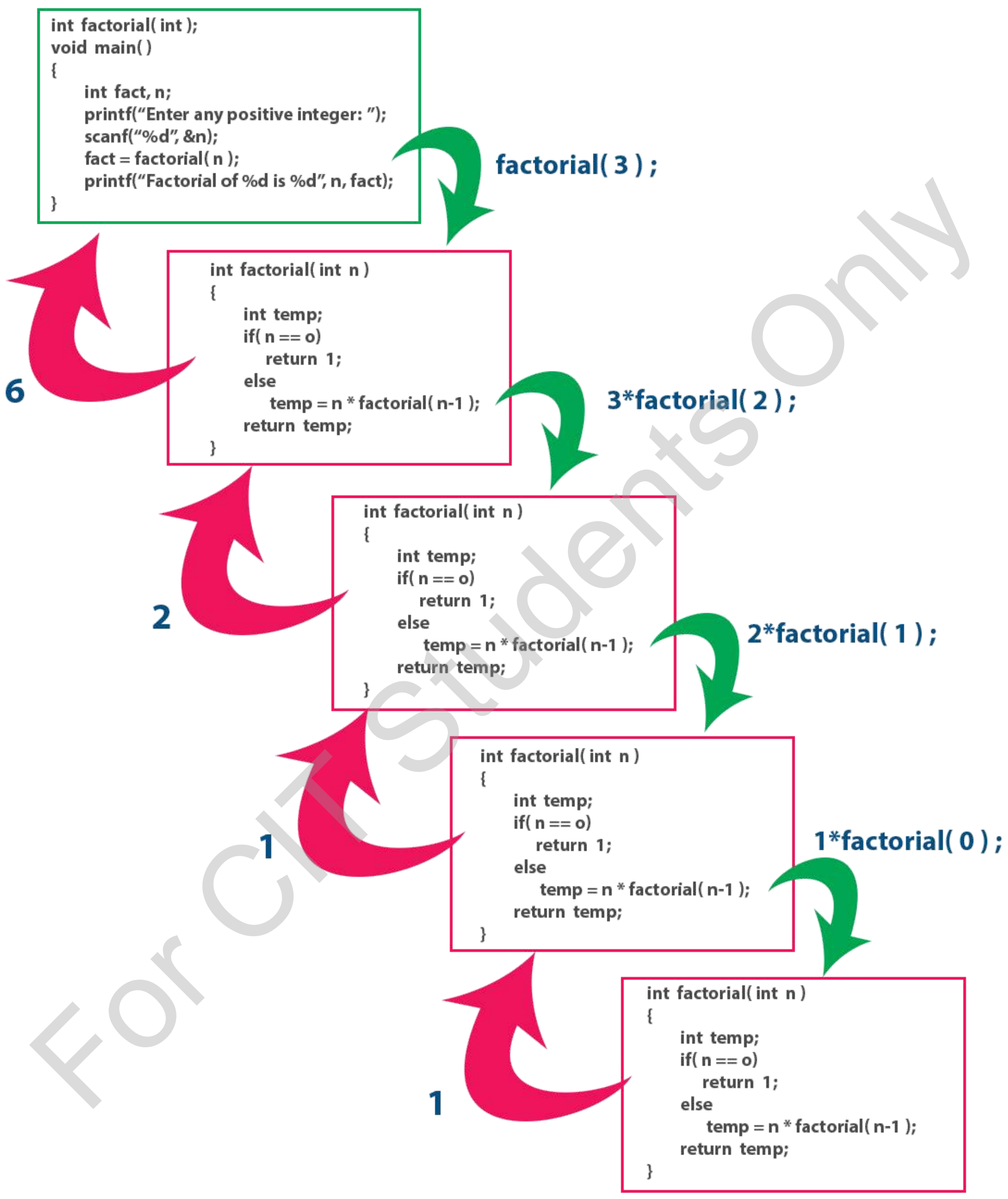


```
int factorial(int n) {
  .....
  return n * factorial(n-1);

}

int main() {
  ....
  factorial(n);
}
```

Ex: The recursive function call execution **to find factorial of 6**. The function calls are stored in the stack until the control reaches the base case condition; then the calculation will occur. **The execution will occur in reverse order; the Last call is executed 1st then 2nd, & 3rd so on**



Ex: The recursive function call execution **to find factorial of 3**.



```
int factorial( int );
void main( )
{
     int fact, n;
     printf("Enter any positive integer: ");
     scanf("%d", &n);
     fact = factorial( n );
     printf("Factorial of %d is %d", n, fact);
}
```

```
int factorial( int n )
{
     int temp;
     if( n == o)
         return 1;
     else
         temp = n * factorial( n-1 );
     return temp;
}
```

```
int factorial( int n )
{
     int temp;
     if( n == o)
         return 1;
     else
         temp = n * factorial( n-1 );
     return temp;
}
```

```
int factorial( int n )
{
     int temp;
     if( n == o)
         return 1;
     else
         temp = n * factorial( n-1 );
     return temp;
}
```

```
int factorial( int n )
{
     int temp;
     if( n == o)
         return 1;
     else
         temp = n * factorial( n-1 );
     return temp;
}
```

Example 2 of 2: Program to find Fibonacci Series of a given number of terms using Direct Recursion Function

```
#include<stdio.h>
int fibonacci(int i);
int main()
{
  int i, n;
  printf("Enter terms for fibonacci series: ");
  scanf("%d", & n);

  for (i = 0; i < n; i++) {
      printf(" %d ", fibonacci(i));
  }
  return 0;
}

int fibonacci(int i)
{
  if (i == 0) {
    return 0;
  }
  if (i == 1) {
    return 1;
  }
  return fibonacci(i - 1) + fibonacci(i - 2);
}
```

Explanation:
In the C program above, we have declared a function named fibonacci(). It takes an integer i as input and returns the ith element of the Fibonacci series. At first, the main() function will be executed where we have taken two variables i and n. We will take input from the user that will be stored in n, and the for loop will execute till n iteration where with each iteration, it will pass the parameter to fibonacci() function where the logic for the Fibonacci series is written. Now inside fibonacci() function, we have nested if-else. If input = 0, it will return 0, and if the input = 1, it will return 1. These are the base cases for the Fibonacci function. If the value of i is greater than 1, then fibonacci(i) will return fibonacci (i - 1) + fibonacci (i -2) recursively, and this recursion will be computed till the base condition.

2. Indirect Recursion

Indirect recursion in C occurs when the first function calls the second function and the second function calls the first function again. Such functions are also called indirect recursive functions.

Following is the structure of indirect recursion.

```
function1()
{    //some code
     function2();
}


function2()
{   //some code
    function1();
}
```

Example: Program with Indirect Recursion Function

This program prints numbers from 1 to 10 in such a manner that when an odd no is encountered, we will print that number plus 1. When an even number is encountered, we would print that number minus 1 and will increment the current number at every step.

```
//Prints 1-10; odd+1 and even-1 using Indirect Recursion Function
#include<stdio.h>
void odd();
void even();
int n=1;

void odd()
{
    if(n <= 10)
    {
        printf("%d ", n+1);
        n++;
        even();
    }
    return;
}

void even()
{
    if(n <= 10)
    {
        printf("%d ", n-1);
        n++;
        odd();
    }
    return;
}

int main()
{
    odd();
}
```

Output:

```
2 1 4 3 6 5 8 7 10 9
```

Advantages of Recursion:

1. The code becomes shorter and reduces the unnecessary calling to functions.
2. Useful for solving formula-based problems and complex algorithms.
3. Useful in Graph and Tree traversal as they are inherently recursive.
4. Recursion helps to divide the problem into sub-problems and then solve them, essentially divide and conquer.

Disadvantages of Recursion:

1. The code becomes hard to understand and analyze.
2. A lot of memory is used to hold the copies of recursive functions in the memory.
3. Time and Space complexity is increased.
4. Recursion is generally slower than iteration.

Summary of Recursion:

- There are two types of recursion in the C language. The first is Direct recursion and Indirect recursion.
- The Direct recursion in C occurs when a function calls itself directly from inside.
- Indirect recursion occurs when a function calls another function, and then that function calls the first function again.
- The function call to itself is a recursive call, and the function will become a recursive function.
- The stack is maintained in the memory to store the recursive calls and all the variables with the value passed in them.
