FUNCTIONS

Introduction to Functions:

Designing Structured Programs in C

Structured programming is a programming technique in which a larger program is divided into smaller subprograms to make it easy to understand, easy to implement and makes the code reusable etc,. The structured programming enables code reusability. Code reusability is a method of writing code once and using it many times. Using structured programming technique, we write the code once and use it many times. Structured programming also makes the program easy to understand, improves the quality of the program, easy to implement and reduces time.

In C, the structured programming can be designed using functions concept. Using functions concept, we can divide larger program into smaller subprograms and these subprograms are implemented individually. Every subprogram or function in C is executed individually.

Introduction to Functions in C

When we write a program to solve a larger problem, we divide that larger problem into smaller subproblems and are solved individually to make the program easier. In C, this concept is implemented using functions.

Functions are used to divide a larger program into smaller subprograms such that program becomes easy to understand and easy to implement.

A function is defined as follows...

Definition

Function is a subpart of program used to perform specific task and is executed individually.

Every C program must contain atleast one function called main(). However a program may also contain other functions.

Every function in C has the following...

- Function Declaration (Function Prototype)
- Function Definition
- Function Call

Function Declaration

The function declaration tells the compiler about function name, datatype of the return value and parameters. The function declaration is also called as function prototype. The function

declaration is performed before main function or inside main function or inside any other function.

Function declaration syntax -

returnTypefunctionName(parametersList);

- In the above syntax, returnType specifies the datatype of the value which is sent as a return value from the function definiton.
- The functionName is a user defined name used to identify the function uniquely in the program.
- The parametersList is the data values that are sent to the function definition.

Function Definition

- The function definition provides **the actual code** of that function.
- The function definition is also known as **body of the function**.
- The actual task of the function is implemented in the function definition. That means the actual instructions to be performed by a function are written in function definition. The actual instructions of a function are written inside the braces "{ }". The function definition is performed before main function or after main function.

Function definition syntax -

returnTypefunctionName(parametersList)
{

Actual code...

}

Function Call
The function call tells the compiler when to execute the function definition. When a function call is executed, the execution control jumps to the function definition where the actual code gets executed and returns to the same functions call once the execution completes. The function call is performed inside main function or inside any other function or inside the function itself.

Function call syntax -

functionName(parameters);

Advantages of Functions

- Using funcions we can implement modular programming.
- Functions makes the program more readable and understandable.
- Using functions the program implementation becomes easy.
- Once a function is created it can be used many times (code re-usability).
- Using functions larger program can be divided into smaller modules.

Types of Functions in C

In C Programming Language, based on providing the function definition, functions are divided into two types. Those are as follows...

- **System Defined Functions**
- **User Defined Functions**

1.System Defined Functions

The C Programming Language provides pre-defined functions to make programming easy. These pre-defined functions are known as system defined functions.

The system defined function is defined as follows...

Definition:

The function whose definition is defined by the system is called as system defined function.

The system defined functions are also called as **Library Functions** or **Standard Functions** or **Pre-Defined Functions**.

The implementation of system defined functions is already defined by the system.

2.User Defined Functions:

In C programming language, users can also create their own functions. The functions that are created by users are called as user defined functions.

The user defined function is defined as follows...

Definition:
The function whose definition is defined by the user is called as user defined function.

In C every user defined function must be declared and implemented. Whenever we make function call the function definition gets executed.

For example, consider the following program in which we create a fucntion called addition with two paramenters and a return value.

Program:

```
#include <stdio.h>
#include<conio.h>
void main(){
int num1, num2, result ;
int addition(int,int) ; // function declaration
```

```
clrscr() ;
printf("Enter any two integer numbers : ") ;
scanf("%d%d", &num1, &num2);

result = addition(num1, num2) ; // function call

printf("SUM = %d", result);
getch() ;
}
int addition(int a, int b)  // function definition
{
returna+b ;
}
```

In the above example program,

the function declaration statement "int addition(int,int)" tells the compiler that there is a function with name addition which takes two integer values as parameters and returns an integer value. The function call statement takes the execution control to the additon() definition along with values of num1 and num2. Then function definition executes the code written inside it and comes back to the function call along with return value.

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 calling function to called function. After executing the called function, the execution control comes back to calling function from called function. When the control jumps from calling function to called function it may carry one or more data values called "Paramenters" and while coming back it may carry a single value called "return value". That means the data values transferred from calling function to called function are called as Parameters and the data value transferred from called funcion to calling function is called Return value.
Based on the data flow between the calling function and called function, the functions are classified as follows...

- Function without Parameters and without Return value
- Function with Parameters and without Return value
- Function without Parameters and with Return value
- Function with Parameters and with Return value

Function without Parameters and without Return value

In this type of functions there is no data transfer between calling function and called function. Simply the execution control jumps from calling function to called function and executes called function, and finally comes back to the calling function.

For example, consider the following program...

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

```
void main(){
void addition() ; // function declaration
clrscr() ;

addition() ; // function call

getch() ;
}
void addition()  // function definition
{
int num1, num2 ;
printf("Enter any two integer numbers : ") ;
scanf("%d%d", &num1, &num2);
printf("Sum = %d", num1+num2 ) ;
}
```

Function with Parameters and without Return value

In this type of functions there is data transfer from calling function to called function (parameters) but there is no data transfer from called function to calling function (return value). The execution control jumps from calling function to called function along with the parameters and executes called function, and finally comes back to the calling function.

For example, consider the following program...

```
#include <stdio.h>
#include<conio.h>
void main(){
int num1, num2 ;
void addition(int, int) ; // function declaration
clrscr() ;
printf("Enter any two integer numbers : ") ;
scanf("%d%d", &num1, &num2);

addition(num1, num2) ; // function call

getch() ;
}
void addition(int a, int b)  // function definition
{
printf("Sum = %d", a+b ) ;
}
```

Function without Parameters and with Return value

In this type of functions there is no data transfer from calling function to called function (parameters) but there is data transfer from called function to calling function (return value).

The execution control jumps from calling function to called function and executes called function, and finally comes back to the calling function along with a return value.

For example, consider the following program...

```
#include <stdio.h>
#include<conio.h>
void main(){
int result ;
int addition() ; // function declaration
clrscr() ;

result = addition() ; // function call
printf("Sum = %d", result) ;
getch() ;
}
int addition()  // function definition
{
int num1, num2 ;
printf("Enter any two integer numbers : ") ;
scanf("%d%d", &num1, &num2);
return (num1+num2) ;
}
```

Function with Parameters and with Return value

In this type of functions there is data transfer from calling function to called function (parameters) and also from called function to calling function (return value). The execution control jumps from 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.

For example, consider the following program...

```
#include <stdio.h>
#include<conio.h>
void main(){
int num1, num2, result ;
int addition(int, int) ; // function declaration
clrscr() ;
printf("Enter any two integer numbers : ") ;
scanf("%d%d", &num1, &num2);

result = addition(num1, num2) ; // function call
printf("Sum = %d", result) ;
getch() ;
}
int addition(int a, int b)  // function definition
{
return (a+b) ;
```

```
}
```

Parameter Passing in C

When a function gets executed in the program, the execution control is transferred from calling function to called function and executes function definition, and finally comes back to the calling function. When the execution control is transferred from calling function to called function it may carry one or more number of data values. These data values are called as parameters.

Parameters are the data values that are passed from calling function to called function.

In C, there are two types of parameters and they are as follows...

- **Actual Parameters**
- **Formal Parameters**

The actual parameters are the parameters that are speficified in calling function. The formal parameters are the parameters that are declared at called function. When a function gets executed, the copy of actual parameter values are copied into formal parameters.

In C Programming Language, there are two methods to pass parameters from calling function to called function and they are as follows...

- **Call by Value**
- **Call by Reference**

Call by Value

In call by value parameter passing method, the copy of actual parameter values are copied to formal parameters and these formal parameters are used in called function. The changes made on the formal parameters does not effect the values of actual parameters. That means, after the execution control comes back to the calling function, the actual parameter values remains same. For example consider the following program...

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

void main()
{
int num1, num2 ;
void swap(int,int) ; // function declaration
clrscr() ;
  num1 = 10 ;
  num2 = 20 ;
```

```
printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;

swap(num1, num2) ; // calling function

printf("\nAfter swap: num1 = %d\nnum2 = %d", num1, num2);
getch() ;
}
void swap(int a, int b)  // called function
{
int temp ;
temp = a ;
  a = b ;
  b = temp ;
}
```

Output:

Before swap: num1 = 10, num2 = 20
After swap: num1 = 10, num2 = 20

In the above example program, the variables num1 and num2 are called actual parameters and the variables a and b are called formal parameters. The value of num1 is copied into a and the value of num2 is copied into b. The changes made on variables a and b does not effect the values of num1 and num2.

Call by Reference(or) pointers with Functions

In Call by Reference parameter passing method, the memory location address of the actual parameters is copied to formal parameters. This address is used to access the memory locations of the actual parameters in called function. In this method of parameter passing, the formal parameters must be pointer variables.

That means in call by reference parameter passing method, the address of the actual parameters is passed to the called function and is recieved by the formal parameters (pointers).

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

For example consider the following program...

```
#include <stdio.h>
#include<conio.h>
void main(){
int num1, num2 ;
void swap(int *,int *) ; // function declaration
clrscr() ;
  num1 = 10 ;
```

```
   num2 = 20 ;

printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;
swap(&num1, &num2) ; // calling function

printf("\nAfter swap: num1 = %d, num2 = %d", num1, num2);
getch() ;
}
void swap(int *a, int *b)  // called function
{
int temp ;
temp = *a ;
   *a = *b ;
   *b = temp ;
}
```

Output:
Before swap: num1 = 10, num2 = 20
After swap: num1 = 20, num2 = 10

In the above example program, the addresses of variables num1 and num2 are copied to pointer variables a and b. The changes made on the pointer variables a and b in called function effects the values of actual parameters num1 and num2 in calling function.

Inter Function Communication in C

When a function gets executed in the program, the execution control is transferred from calling function to called function and executes function definition, and finally comes back to the calling function. In this process, both calling and called functions have to communicate each other to exchange information. **The process of exchanging information between calling and called functions is called as inter function communication.**

In C, the inter function communication is classified as follows...

1. Downward Communication
2. Upward Communication
3. Bi-directional Communication

Downward Communication:

In this type of inter function communication, **the data is transferred from calling function to called function but not from called function to calling function**. The functions with parameters and without return value are considered under downward communication. In the case of downward communication, the execution control jumps from calling function to called function along with parameters and executes the function definition,and finally comes back to the calling function without any return value. For example consider the following program...

```
#include <stdio.h>
#include<conio.h>
void main(){
int num1, num2 ;
void addition(int, int) ; // function declaration
clrscr() ;
   num1 = 10 ;
   num2 = 20 ;

printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;
addition(num1, num2) ; // calling function

getch() ;
}
void addition(int a, int b)  // called function
{

printf("SUM = %d", a+b) ;

}
```

Output:

SUM = 30

Upward Communication:

In this type of inter function communication, **the data is transferred from called function to calling function but not from calling function to called function**. The functions without parameters and with return value are considered under upward communication. In the case of upward communication, the execution control jumps from calling function to called function without parameters and executes the function definition, and finally comes back to the calling function along with a return value. For example consider the following program...

```
#include <stdio.h>
#include<conio.h>
void main(){
int result ;
int addition() ; // function declaration
clrscr() ;

result = addition() ; // calling function

printf("SUM = %d", result) ;
getch() ;
}

int addition()  // called function
{
int num1, num2 ;
   num1 = 10;
   num2 = 20;
```

```
return (num1+num2) ;
}
```
Output:
SUM = 30

Bi - Directional Communication:

In this type of inter function communication, **the data is transferred from calling function to called function and also from called function to calling function**. The functions with parameters and with return value are considered under bi-directional communication. In the case of bi-drectional communication, the execution control jumps from calling function to called function along with parameters and executes the function definition, and finally comes back to the calling function along with a return value. For example consider the following program...

```
#include <stdio.h>
#include<conio.h>
void main(){
int num1, num2, result ;
int addition(int, int) ; // function declaration
clrscr() ;

   num1 = 10 ;
   num2 = 20 ;

result = addition(num1, num2) ; // calling function

printf("SUM = %d", result) ;
getch() ;
}
int addition(int a, int b)  // called function
{
return (a+b) ;
}
```
Output:
SUM = 30

Passing Arrays to Function:

Declaring Function with array as a parameter

There are two possible ways to do so, one by using call by value and other by using call by reference.

1.We can either have an array as a parameter.

int sum (int arr[]);

2. Or, we can have a pointer in the parameter list, to hold the base address of our array.

int sum (int* ptr);

Returning an Array from a function

We don't return an array from functions rather we return a pointer holding the base address of the array to be returned. But we must, make sure that the array exists after the function ends i.e. the array is not local to the function.

```
int* sum (int x[])
{
   // statements
   return x ;
}
```

We will discuss about this when we will study pointers with arrays.

Passing arrays as parameter to function

Now let's see a few examples where we will pass a single array element as argument to a function, a one dimensional array to a function and a multidimensional array to a function.

Passing a single array element to a function
Let's write a very simple program, where we will declare and define an array of integers in our main() function and pass one of the array element to a function, which will just print the value of the element.

```
#include<stdio.h>

void giveMeArray(int a);

int main()
{
int myArray[] = { 2, 3, 4 };
giveMeArray(myArray[2]);        //Passing array element myArray[2] only.
return 0;
}

void giveMeArray(int a)
{
printf("%d", a);
}
```

Passing a complete One-dimensional array to a function

To understand how this is done, let's write a function to find out average of all the elements of the array and print it.

We will only send in the name of the array as argument, which is nothing but the address of the starting element of the array, or we can say the starting memory address.

```
#include<stdio.h>

float findAverage(int marks[]);

int main()
{
   float avg;
   int marks[] = {99, 90, 96, 93, 95};
   avg = findAverage(marks);       // name of the array is passed as argument.
   printf("Average marks = %.1f", avg);
   return 0;
}

float findAverage(int marks[])
{
   int i, sum = 0;
   float avg;
   for (i = 0; i <= 4; i++) {
      sum += marks[i];
   }
   avg = (sum / 5);
   return avg;
}
```

94.6

Passing a Multi-dimensional array to a function

Here again, we will only pass the name of the array as argument.

```
#include<stdio.h>

void displayArray(int arr[3][3]);
```

```
int main()
{
   int arr[3][3], i, j;
   printf("Please enter 9 numbers for the array: \n");
   for (i = 0; i < 3; ++i)
   {
      for (j = 0; j < 3; ++j)
      {
         scanf("%d", &arr[i][j]);
      }
   }
   // passing the array as argument
   displayArray(arr);
   return 0;
}

void displayArray(int arr[3][3])
{
   int i, j;
   printf("The complete array is: \n");
   for (i = 0; i < 3; ++i)
   {
      // getting cursor to new line
      printf("\n");
      for (j = 0; j < 3; ++j)
      {
         // \t is used to provide tab space
         printf("%d\t", arr[i][j]);
      }
   }
}
```

UNIT II

Please enter 9 numbers for the array:
1
2
3
4
5
6
7
8
9

The complete array is:
1 2 3
4 5 6
7 8 9

P.VAMSHEEDHAR REDDY
(Asst.Prof,CSE DEPT)

(If any data needed to add into this material please write to : pvamsheedharreddy@gmail.com)

UNIT II

Scope of Variable in C

When we declare a variable in a program, it can not be accessed against the scope rules. Variables can be accessed based on their scope. Scope of a variable decides the portion of a program in which the variable can be accessed. Scope of the variable is defined as follows...

Definition:
Scope of a variable is the portion of the program where a defined variable can be accessed.

The variable scope defines the visibility of variable in the program. Scope of a variable depends on the position of variable declaration.

In C programming language, a variable can be declared in three different positions and they are as follows...

1. **Before the function definition (Global Declaration)**
2. **Inside the function or block (Local Declaration)**
3. **In the function definition parameters (Formal Parameters)**

Before the function definition (Global Declaration):

Declaring a variable before the function definition (outside the function definition) is called global declaration. The variable declared using global declaration is called global variable. Tha global variable can be accessed by all the functions that are defined after the global declaration. That means the global variable can be accessed any where in the program after its declaration. The global variable scope is said to be file scope.

```
#include <stdio.h>
#include<conio.h>
int num1, num2 ;
void main(){
void addition() ;
void subtraction() ;
void multiplication() ;
clrscr() ;
  num1 = 10 ;
  num2 = 20 ;
printf("num1 = %d, num2 = %d", num1, num2) ;
addition() ;
subtraction() ;
multiplication() ;
getch() ;
}
```

P.VAMSHEEDHAR REDDY
(Asst.Prof,CSE DEPT)

(If any data needed to add into this material please write to : pvamsheedharreddy@gmail.com)

UNIT II

```
void addition()
{
int result ;
result = num1 + num2 ;
printf("\naddition = %d", result) ;
}
void subtraction()
{
int result ;
result = num1 - num2 ;
printf("\nsubtraction = %d", result) ;
}
void multiplication()
{
int result ;
result = num1 * num2 ;
printf("\nmultiplication = %d", result) ;
}
```

Output:
num1 = 10, num2 = 20
addition = 30
subtraction = -10
multiplication = 200

In the above example program, the variables num1 and num2 are declared as global variables. They are declared before the main() function. So, they can be accessed by function main() and other functions that are defined after main(). In the above example, the functions main(), addition(), subtraction() and multiplication() can access the variables num1 and num2.

Inside the function or block (Local Declaration):

Declaring a variable inside the function or block is called local declaration. The variable declared using local declaration is called local variable. The local variable can be accessed only by the function or block in which it is declared. That means the local variable can be accessed only inside the function or block in which it is declared.

```
#include <stdio.h>
#include<conio.h>
void main(){
void addition() ;
int num1, num2 ;
clrscr() ;
  num1 = 10 ;
```

UNIT II

```
   num2 = 20 ;
printf("num1 = %d, num2 = %d", num1, num2) ;
addition() ;
getch() ;
}
void addition()
{
intsumResult ;
sumResult = num1 + num2 ;
printf("\naddition = %d", sumResult) ;
}
```

Output:
ERROR

The above example program shows an error because, the variables num1 and num2 are declared inside the function main(). So, they can be used only inside main() function and not in addition() function.

In the function definition parameters (Formal Parameters):

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

```
#include <stdio.h>
#include<conio.h>
void main(){
void addition(int, int) ;
int num1, num2 ;
clrscr() ;
   num1 = 10 ;
   num2 = 20 ;
addition(num1, num2) ;
getch() ;
}
void addition(int a, int b)
{
intsumResult ;
sumResult = a + b ;
printf("\naddition = %d", sumResult) ;
}
```

Output:
addition = 30

UNIT II

Recursive Functions in C

In C programming language, function calling can be made from main() function, other functions or from same function itself. The recursive function is definedd as follows...

Definition:
A function called by itself is called recursive function.

The recursive functions should be used very carefully because, when a function called by itself it enters into 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.

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

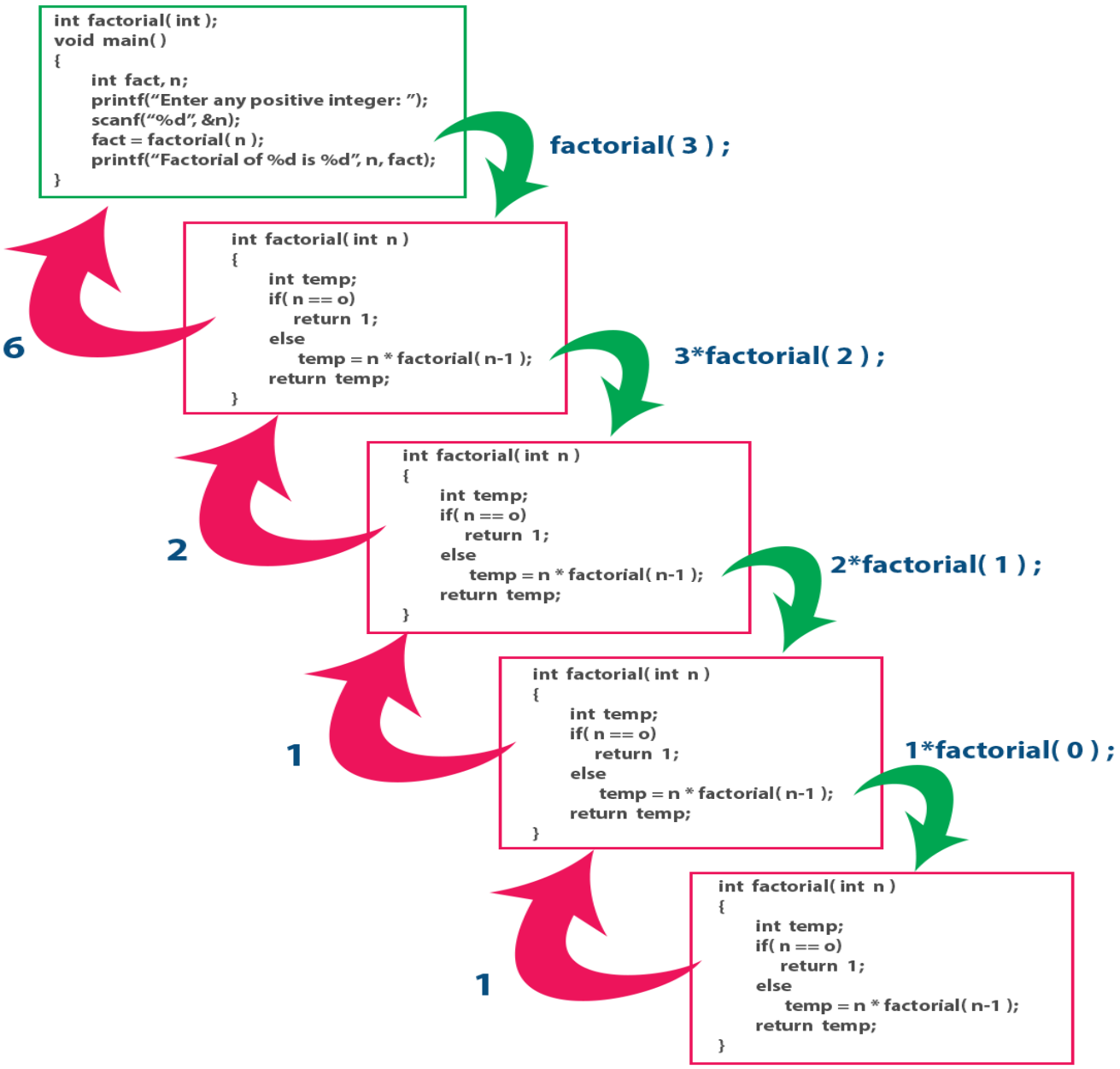
```
#include <stdio.h>
#include<conio.h>
int  factorial( int ) ;
void  main( )
{
        int  fact, n ;
        printf("Enter any positive integer: ") ;
        scanf("%d", &n) ;
        fact = factorial( n ) ;
        printf("Factorial of %d is %d", n, fact) ;
}
int  factorial( int  n )
{
        int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

Output:
Enter any positive integer: 3
Factorial of 3 is 6

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 complete execution process of the above program is shown in the following figure...



P.VAMSHEEDHAR REDDY
(Asst.Prof,CSE DEPT)

(If any data needed to add into this material please write to : pvamsheedharreddy@gmail.com)

UNIT II

Type Qualifiers in C

In C programming language, type qualifiers are the keywords used to modify the properties of variables. Using type qualifiers, we can change the properties of variables.

C programming language provides two type qualifiers and they are as follows...

1. const
2. volatile(Variable)

const type qualifier in C

- The const type qualifier is used to create constant variables.
- When a variable is created with const keyword, the value of that variable can't be changed once it is defined.
- That means once a value is assigned to a constant variable, that value is fixed and cannot be changed throughout the program.

- The keyword **const** is used at the time of variable declaration.

We use the following syntax to create constant variable using const keyword.

constdatatypevariableName ;

When a variable is created with const keyword it becomes a constant variable. The value of the constant variable can't be changed once it is defined. The following program generates error message because we try to change the value of constant variable x.

```
#include <stdio.h>
#include<conio.h>
void main()
{
inti = 9 ;
constint x = 10 ;
clrscr() ;

i = 15 ;
   x = 100 ; // creates an error

printf("i = %d\nx = %d", i, x ) ;

}
```

Output:
Compiler error, we cannot modify const variable

UNIT II

volatile type qualifier in C

The volatile type qualifier is used to create variables whose values can't be changed in the program explicitly but can be changed by any external device or hardware.

For example, the variable which is used to store system clock is defined as volatile variable. The value of this variable is not changed explicitly in the program but is changed by the clock routine of the operating system.

P.VAMSHEEDHAR REDDY
(Asst.Prof,CSE DEPT)

(If any data needed to add into this material please write to : pvamsheedharreddy@gmail.com)