Unit : 6 Functions

Lesson Structure

6.0 Objective

After going through this unit, you will learn

- Use of functions in programming
- To define a function

- To declare a function
- To call a function
- Passing parameners and returnning statement
- Functions call by value mechanism
- Functions call by reference mechanism
- Recursive functions

6.1 Introduction

C enables us to break a program into segments commonly known as functions. Each part may be independently coded and later combined into a single unit. These smaller segments are called subprograms or functions. Functions are easier to understand, debug and test.

Functions are very important tool for modular programming, where we break large programs into small subprograms or modules. The use of functions reduces complexity and makes programming simple and easy to understand.

In this unit, we will discuss how functions are defind and how are they accessed from the main program? We will also discuss various types of functions and how to invoke them. And finally you will learn an interesting and important programming technique known as Recursion, in which a function calls within itself.

6.2 Definition of Function

A function is a self-contained block of executable code that can be called from any other function. In many programs, a set of statements are to be executed repeatedly at various places in the program and may with different sets of data, the idea of function comes in mind. You keep those repeating statement in a function and call them as and when required. When a function is called, the control transfers to the called function, which will be executed, and then transfer the control back to the calling function to the statement following the function call.

A function definition is also known as function implementation include the following statements.

1. Function Header
2. Function Body

The general syntax of a function definition is :

```
returntype functionname (parameterlist)
{
local variable declaration;
executable statements;
.............
...................
Return statement;
}
```

The first line

return_type function_name(parameter_list)

is known as the function header and the statements within the opening closing braces constitute the function body which is a combine statement.

The function body contains the declaration and statements necessary for performing the required task. The body enclosed in braces, contains local declaration, executable statements and return statement.

Example 1: Program to demonstrate function.

```
#include < stdio.h >
#include < conio.h>
void calling ( ); / * function declaration */
void main ()
{
calling () ; / * function call */
printf ("you are in main");
getch ();
}
void calling()
printf (" \n you are in calling");
}
```

O/P:

You are in calling

you are in main

6.3 Declaration of Function

Like variables all functions in C program must be declared, before they are invoked, A function declaration consists of four parts.

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

the general synax to declare a function is:

return_type function_name (parameter_list);

for example:

int square(int no);

float temperature(float c, float f);

6.4 Function Prototyping

Function prototyping require that every function which is to be accessed should be declared in the calling function. The function declaration, that will be discussed earlier, will be included for every function is its calling function or in global declaration section section.

Because if we do not use function prototyping we must define the called function before the calling function otherwise the compiler will give an error. Function prototyping solve the problem.

Example 2: program to calculate the square of a given integer.

```
#include <stdio.h>
#include < conio.h >
int square (int); /* function prototype */
void main ()
 {
int n, sq;
printf ( “enter a numbrt to find square” );
sq = square (n); / * function call * /
printf ( “\n square of the number is: %d” , sq);
getych ();
```

```
}
int square ( int no)
{
int n:
n = no *no;
return (n);
}
```

O/P:

Enter a number to find square: 6

square of the number is: 36

6.5 The Return Statement

The return statement is used to terminate the execution of a function and returns control to the calling function. when the return staement encountered, the program execution resumes in the calling function. A return statement may or may not return a value to the calling function. We can pass any number of arguments to a function but it always return only one value at a time.

The return statement can take one of the following forms.

```
return;
or
return (expression);
```

the first, the plain return does not return any value. It acts like the closing braces of the function. When the return encountered, the control is immediately passed back to the calling function. An example of simple return is as follows;

```
if (error)
        return;
```

The second form of return with expression returns the value of the epression. For example the function

```
int  mul (int x, int y )
{
int p;
p= x*y;
```

```
return (p );
}
```

Returns the value of p which is product of x and y. the last two statements can be combined into one statement as follows:

```
return (x * y);
```

A function may have more than one return statements. This situation arises when the value returned is based on conditions. for example :

```
If ( x <= 0)
        return 0;
 else
        return 1;
```

All functions by default return int type data. We can force a function to return a perticular type of data by using a type specifier in the function header.

6.6 Types of Function Calling (Invoking)

We categorize a function's calling depending on arguments or parameters and their returning a value. We can divide a function's calling into four categoy depending on parameters passed and return type.

Category 1 : function with no arguments and no return value.

Category 2 : functions with arguments and no return value.

Category 3 : functions with arguments and return value.

Category 4 : functions with no arguments but return a value.

Category 1: Functions with no arguments and no return value

These type of function has no arguments and does not return any value to the calling function.

These type of functions are confined to themselves i.e neither they receive any data from the calling function nor they transfer any data to the calling function. So there is no data communication between the calling and the called function.Only program control will be transfered.

Example 3 : Write a program to demonstrate function with no arguments and no return value.

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

```
#include<conio.h>
void called (); /* with no arguments and no return value */
void main ()
 {
        printf (“/n in calling function” );
        called ();
        printf ( “\n in calling again” );
         }
        void called ()
        {
        printf ( “control in called function”);
         }
```

O/P:

In calling function

Control in called function

In calling again

Category 2 : Functions with argument and no return value

These type of functions receive data from calling function but does not send value to the calling function. One way communication takes place between the calling and the called function.

Example 4 : Write a program to demonstrate function with argunmets but no return value

```
#include<stdio.h>
#include < conio.h>
void sum ( int a, int b, ); /* with argunment but no return value */
 {
int num1, num2;
printf (“enter two numbers”);
scanf (“%d%d”, &num1, &num2);
sum (num1, num2);
getch ();
```

```
}
void sum (int a, int b)
{
        int s1;
        s1 = a + b;
printf ("\n the sum of two numbers is %d", s1);
}
```

O/P:

Enter two numbers: 23 45

The sum of two numbers is: 68

Category 3 : Function with argument and return value

In this category two way communication takes place between the calling and called function i.e function returns a value and also arguments passed to it.

Example 5 : Write a program to demonstrate function with arguments and return value

```
#include <stdio.h>
#include < conio.h>
int sum (int a, int b, ); /* with argument and with return value */
void amin ()
{
int num1, num2, result;
printf ("enter two numbers");
scanf ("%d%d", &num1, num2);
result = sum (num1, num2);
printf ("the sum of two numbers is : %d", result);
getch ();
}
int sum (int a, int b)
{
        int s1 ;
        s1 = a+b ;
```

```
return s1 ; /* returning value to the calling function */
}
```

O/P:

Enter two numbers: 23 45

The sum of two numbers is : 68

Category 4 : Function with no arguments but with return value

This type of function does not send any value to the calling function but sends a value to the called function.

Example 6 : Write a program to demonstrate function with no argunments but return value

```
#include <stdio.h>
#include < conio.h>
int sum (); /* with return value but no argument */
void main ()
{
        int s1 ;
        s1 = sum ();
        printf (“sum is %d”, s1);
        getch ();
}
void sum ( )
{
int a, b, num;
printf (“Enter two numbers”);
scanf (“% d% d”, & a, & b);
num = a + b;
return num ; / * returning value to the calling function */
}
```

O/P;

Enter two numbers: 23 45

The sum of two numbers is : 68

6.7 Call By Value

In call by value mechanism the value of the variables are passed by the calling function to the called function. By default a functions is called by call by value mechanism. In this case the called function uses a copy of actual arguments, and if the called function modify the value of parameters passed to it, then the changes will reflect in the called function only. In the calling function's variable no change will made.

Example 7 : program to demonstrate call by value

```
#include < stdio.h >
#include < conio.h >
void awap ( int p, int q);
void main ()
 {
int a, b;
printf ("enter two values");
scanf ("%d%d" , &a, &b);
printf ("the value if a and a and b is %d and %d", a,b);
swap (a,b);
printf ("\n the value of  a and b in calling after swapping is %d and %d", a,b);
getch ();
 }
void main (int p, int q)
 {
int temp ;
printf ("the value of p and q before swapping is %d and %d", p,q);
temp = p;
p = q ;
q = temp ;
printf ("\n the value of p and q in called after swapping is %d and %d", p,q );
 {
```

O/P

The value of a and b is 10 and 15

The value of p and q before swapping is 10 and 15

The value of p and q in called after swapping is 15 and 10

The value of a and b in calling after swapping is 10 and 15

before swapping the value of variables in calling and called function.

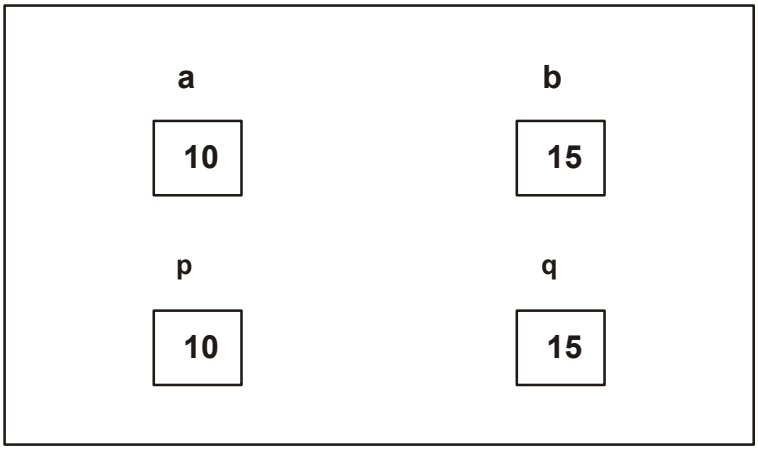

Figure 1

After swapping the value of variables in calling and called.

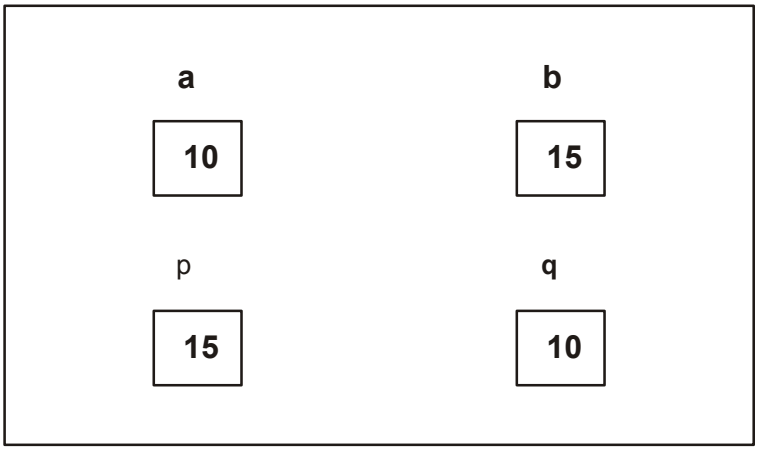

Figure 2

The value of a and b will remain unchanged even after swapping.

6.8 Call by Reference

When the callling function passes arguments to the called function using call by value method, the only way to return the modified value of the argunments to the caller is explicitly using return statement. The better option when a function wants to modify the value of the argunments is pass arguments using call by refrence technique. In call by refrence machanism we pass the address of the variables as value. Therefore if any change made in the called function it also reflects in the calling function.

Example 8 : program to demonstrate call by reference.

```
#include < stdio.h >
#include < conio.h >
void awap ( int* p, int *q);
void main ()
 {
int a, b;
printf ("enter two values");
scanf ("%d%d" , &a, &b);
printf ("the value if a and a and b is %d and %d", a,b);
swap (&a,&b);
printf ("\n the value of  a and b in calling after swapping is %d and %d", a,b);
getch ();
 }
void main (int * p, int *q)
 {
int temp ;
printf ("the value of p and q before swapping is %d and %d", p,q);
temp = *p;
 *p = *q ;
*q = temp ;
printf ("\n the value of p and q in called after swapping is %d and %d",* p,*q );
 {
O/P
The value of a and b is 10 and 15
```

The value of p and q before swapping is 10 and 15

The value of p and q in called after swapping is 15 and 10

The value of a and b in calling after swapping is 15 and 10

Programs based on function

Example 9 : Write a program to find greater among three numbers.

```
#include < stdio.h >
#include < conio.h >
int greater (int a, int b, int c):
void main ()
 {
int num1, num2, num3, large ;
printf (“enter three values:”);
scanf( “ %d %d” %d”, & num1, & num2, & num3 );
large = greater (a, b, c );
printf (“the largest number = %d”, large );
getch ();
 }
int greater ( int a, int b, int c,)
 {
if (a > b && a>c)
return a ;
if ( b > a && b>c)
return b ;
else
return c;
 }
```

O/P:

Enter three numbers : 12 15 8

Largest number = 15

Example 10 : Write a program usins function to check whether a entered number is prime or not.

```
#include < stdio.h >
#include < conio.h >
void prime ();
void main ()
{
prime ();
getch ();
 }
void main () {
int num , i , flag = 0 ;
printf ( “Enter a number to check: \n”);
scanf ( “%d”, &num );
for (i = 2 ; i< = num/2 ; ++i)
 {
if (num % i = = 0)
 {
flag  = 1;
break ;
 }
}
if (flag = = 1)
printf (“ %d is not prime”, num );
else
printf(“%d is prime”, num);
 }
O/P:
Enter a number to check:5
 5 is prime
```

Example 11: Write a program to find factorial of a number using function.

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

int factorial (int);
void main ()
 {
int i , fact , num ;

printf (“Enter a number:”);
scanf(“%d”, &num);

fact = factorial (num);
printf ( “ Factorial of %d is: %d”, num, fact ,);
 }
        int factorial (int num )
 {
        int i,f = 1 ;
        for (i = 1; i <=num ; i++)
        f = f * i ;
return f ;
 }

O/P :
Enter a number : 5
Factorial of 5 is : 120
```

Example 12 : Write a program to calculate p(n/r)

```
#include < stdio.h >
#include < conio.h >
int fact (int);
void main ()
 {
```

```
int n, r ;
float result;
clrscr ();
printf ( " \n enter the value of n :");
scanf ("%d, &n);
printf ("n Enter the value of r: ");
scanf ( "%d", &r);
result = (float) fact(n) / fact(r);
printf ( "\n (n/r) : p (%d) = %f", n, r, result);
getch ();
}
int fact (int num)
{
int f = 1,i ;
for (i = num ; i>=1 ; i -- )
f = f * i ;
return f ;
}
```

O/P:

Enter the value of n : 4

Enter the value of r : 2

P (n /r) : p (4) / (2) = 12.00

Example 13 : Write a program to sum the series 1/1! + 1/2 ! + 1/3! +.....+ 1/n!

```
#include < stdio.h >
#include < conio.h >
int fact (int);
void main ()
{
```

```
int n, f, i ;
float result = 0.0;
clrscr ();
printf ( “  enter the value of n :”);
scanf (“%d, &n);
        for ( i = 1 ; i <= n ; i ++)
 {
f = fact (i);
result += 1 / (float) f;
 }
printf ( “\n the sum of the series 1/1! + 1/2! +..... = %f “,result );
getch();
 }
int fact (int num )
 {
        int f = 1, i ;
        for ( I = num ; i >= 1 ; i--)
        f = f * i;
return f;
 }
```

O/P;

enter the value of n : 2

The sum of series 1/1! + 1/2!..... = 1.5

6.9 Types of Variables and Storage Classes

In C, all constants and variables have a defined scope. Scope means the accessibility and visibility of the variables at different points in the program.

The storage class of a variable defines the scope(visibility) and lifetime of variables and/or functions declared within a C program. In addition the storage class gives the following information about the variables or the function.

1. The storage class of a function or variable determines the part of memory where storage space will be allocated for that variable or function (whether the variable / function will be stored in RAM or in the resister.)
2. It specifies how long the storage allocation will continue to exist for that function or variable.
3. It specifies that scope of the variable or function. i.e, the storage class indicates the part of the C program in which the variable name is visible or the part in which it is acceeptable.
4. It specifies whether the variable or function has internal, external or no linkage.
5. It specifies whether the variable will be automatically initialized to zero or to any indeterminate value.

C supports four storage classes :

1. Automatic or local variables
2. Expernal or global variables
3. Static variables
4. Register variables

The general syntax to specifying the storage class of a variables is as follows :

Storage_class_specifier data_type variable_name

6.9.1 Automatic Variables

Automatic variables are declared inside a function in which they are to be utilized. They are created when the function is called and destroyed automatically when the function is exited. Automatic variables are private to the function in which they are declared. It is the default.

Important things to remember about the variables declared automatic :

- All local variables declared within a function belong to automatic storage class by default.
- They should be declared at the start of the program block.
- Memory for the variable is automatically allocated upon entry to a block and freed automatically upon exit from a block.
- The scope of the variable is local to the block in which it is declared. These variables may be declared within a nested block.
- The auto variables are stored in the primary memory of the computer.
- If automatic variables are not initialized at the time of declaration. then a garbage value is assinged to it.

Example 14: Write a program to demonstrate automatic variable.

```
#include < stdio.h >
#include < conio.h >
void func 1
 {
        int a = 10; / * local variable */
        printf ( " \n a = %d ", a);
 }
void func 2()
 {
int a =20 ;
printf ( " \n = %d", a);
 }
void main ()
        {
int a = 30 ; / * local variable * /
 func 1();
func 2();
printf ( " \n a = %d", a);
getch ();
 }
```

O/P;

a = 10

a = 20

a = 30

6.9.2 External or Global variables

Variables that are both alive and active throughout the entire program, are known as external variables. They are also known as global variables. Unlike local variables, global variables can be accessed by any function in the program. External variables are declared outside a function.

Points to remenber

- These are global and can be accessed by any function within its scope. Therefore value may be assinged in one can be written in another.
- There is difference in external variable definition and declaration.
- Initial values can be assigned.
- The external specifier is not required in external variable definition.
- A declaration is required if the external variable definition comes after the function definition.
- A declaration begins with an external specifier.
- Only when external variable is defined is the storage space allocated.
- External variables can be assigned initial values as a part of variable definitions, but the values must be constants rather than the expressions.
- If external variable is not initialized then it is automatically assigned by zero.

Example 15 : Write a program to demonstrate global varialble.

```
#include < stdio.h >
#include < conio.h >
int func1 func1 ();
int func1 func2 ();
int func1 func3 ();
in x;/* global variable */
void main ()
{
x = 10;
printf (“x = %d\n”, x);
printf (“x = %d\n”, func1 ( ));
printf (“x = %d\n”, func2 ( ));
printf (“x = %d\n”, func3 ( ));
getch 0;
 }
int func1 ()
```

```
{
x = x + 10;
}
int func2 ()
{
int x ;
x = 1 ;
return ( x) ;
}
int func3 ()
{
x = x + 10 ;
}
O/P ;
x = 10
x = 20
x = 1;
x = 30
```

in func1 and func3 uses the global variable x so, we do not need to pass explicitly the value of x and also we do need to return the value of x. Since the value of x is directly available everywhere in the program.

6.9.3 Static Variable

In case of single programs static variables are defined within functions and individually have the same scope as automatic variables. But static variables retain their values throughout the execution of program within their previous values.

Points to remember :

- The specifier preceeds the declaration. Static and the value cannot be accessed outside of their defining function.
- The static variables may have same name as that of external variables but the local variables take precedence in the function. Therefore external variables maintain their independence with locally defined auto and static variables.

- Initial value is expressed as the constant and not expression.
- Zero are assigned to all variables whose declaration do not include explicit initial values. Hence they always have assigned values.
- Initialization is done only is the first execution. Let us study this sample program to print value of a static variable.

Example 16 : Program to demonstrate static variable :

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

O/P :

x = 1

x = 2

x = 3

6.9.4 Register Variables

Besides three storage class specifications namely, Automatic, External and Static, there is a register storage class.Registers are special storage areas within a computer's CPU. All the arithmetic and logical operations are carried out with these registers. For the

same program, the execution time can be reduced if certain values can be stored in register rather than memory. These programs are smaller in size (as few instructions are required) and few data transfer are required.The reduction is there in machine code and not in source code. They are declared by the proceeding declaration by register reserved word as follows: register int m;

Points to remember :

- These variables are stored in registers of computers. If the register are not available they are put in memory.
- Usually 2 or 3 register variables are there in the program.
- Scope is same as automatic variable, local to a function in which they are declared.
- Address operator '&' cannot be applied to a register variable.
- If the register is not available the variable the variable is though to be like the automatic variable.
- Usually associated integer variable but with other types it is allowed having same size (short or unsigned).
- Can be formal arguments in functions.
- Pointers to register variables are not allowed.

Example 17 : Program to demonstrate register variable:

```
#include < stdio.h >
#include < conio.h >
int exp (int a, int b);
void main ()
 {
int a = 3 , b = 5 , res ;
res = exp (a, b);
printf( " \n to the power of  %d = %d", a , b , res );
getch ();
 }
int exp (int a, int b)
 {
```

```
register int res = 1 ;
int i ;
for ( i = 1 ; i <=b ; i++)
res res * b ;
return res ;
 }
```

O/P :

‘3 to power of 5 = 243

6.10 Recursion

Recursion is a special case where a function calls itself. A function that contains a function call to itself or a function call to a second function which eventually calls that first function is known as recursive function. Recursive functions are those in which there is atleast one function call to itself.

Two important conditions which must be satisfied by any recursive function are :

1. Each time a function calls itself, it must be nearer to a solution.
2. There must be a decision criterion for stopping the proces or computation.

Example 18 : Write a program using recursion to find factorial of a given number.

```
#include < stdio.h >
#include < conio.h >
void main ()
 {
long int n, fact ;
factorial (int n) ; / * function declaration * /
printf ( “\n enter the number to find factorial :”);
scanf (“ %d”, &n) ;
printf ( “ \n factorial of % is :”) ;
fact = factorial ( n ) ; / * Function Call * /
printf ( “%d”, fact);
```

```
getch ();
 }
long factorial ( int num )
 {
if ( num = = 0 ) / * terminating condition * /
return 1 ;
else
return num * fact ( num - 1 ); / * recursive call * /
 }
```

O /P :

Enter the number to find factorial : 5

The factorial of 5 is : 120

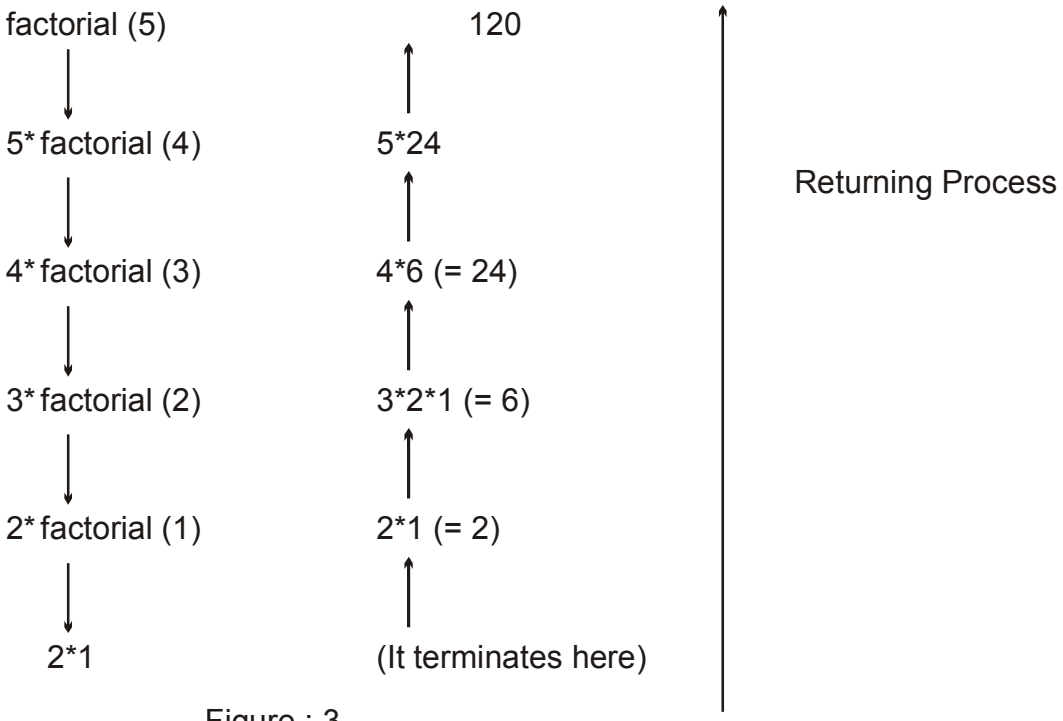

Figure : 3

Example 19 : Write a program using recursion to find sum of digits of a given number.

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

```
#include < conio.h >
void main ()
{
int num, sum1 ;
int sum (int);
printf ( “enter a number \n”);
scanf (“%d”, &num );
sum1 = sum (num) ;
printf ( “ sum of digits is %d”, sum);
getch ();
 }
 int sum (int num)
{
n = num % 10 ;
sum = n + sum (num /10);
 }
return sum ;
}
```

O/P:

Enter a number : 123

Sum of digits is 6

Example 20 : C program to check if a number is palindrome using recursion.

```
#include < stdio.h >
#include < conio.h >
int palindrome (int)
void main ()
 {
        int num, sum ;
```

```
        printf ( “Enter a number :”);
        scanf ( “%d “ , &num ) ;
sum = Palindrome ( num ) ;
if( num = = sum )
 printf ( “%d is a palindrome”, num ) ;
else
 printf ( “ %d is not a palindrome”, num ) ;
return 0 ;
 }
int palindrome ( int num) {
        static int sum=0 , r ;
if (num ! = 0 ) {
        r = num % 10 ;
sum = sum * 10 + r ;
palindrome ( num /10 );
 }
return sum ;
 }
```

O/P :

Enter a number: 121

121 is a palindrome

6.11 Summary

In this unit we learn about functions. Definition of functions, declaration of function, function call, prototyping, storage class, calling of function, call by value and call by refrence and recursive functions.

Every function in a program is supposed to perform a well defind task.The momemt the compiler encounters a function call, the control jumps to the statements that are a part of the called function. After the called function is executed, the control is returned back to the calling program.

The void return type in a function means the function does not return any value to the calling function. A function can accept any number of values but it always return one and only one cvalue. A return statement is required type is anything other than void.

Any variable declared in function are local to it and are created with function call and destroyed with function return. The actual and formal arguments should match in type, order and number. A recursive function should have a terminating condition i.e function should return a value instead of a repetitive function call.

6.12 Questions for Exercise

1. Define function? Why they are needed ?
2. How many types of storage class in c? Explain.
3. What is recursion? Explain.
4. Write a program using functions to check whether a given number is perfect or not.
5. Write a program to concatenate two strings using functions.
6. Write a program to read an integer number. Print the reverse of this number using recursion.
7. Write a program to find HCF of two numbers using recursion.
8. Write a program to calculate the area of a triangle.

6.13 Suggested Readings

1. Programming with ANSI and TURBO C, Ashok N. Kamthane, Pearson Education, 2002.
2. C, The complete Reference, Fourth Edition, Herbert Schildt, TMH, 2002.
3. The C programming Language, Brain W. Kernighan, Dennis M. Ritchie, PHI.
4. Computer Programming in C, Raja Raman. V , 2002, PHI.
5. Programming in ANSI C, E. balagurusamy third edition.

Reference Link

1. www.programiz.com/c-programimg/c-functions-examples.
2. www.tutroialspoint.com/cprogramming/c_functions.htm
3. www.cprogramming.com
4. en.wikipedia.org/wiki/c_ (programming_language)

●●●