MASTER OF COMPUTER APPLICATIONS

SEMESTER 1

PROGRAMMING & PROBLEM-SOLVING USING C

Unit 5

Functions

Table of Contents

1. INTRODUCTION

In the previous unit, you studied about the control statements and its usage in C. You also studied how those control statements are helpful in making decisions when various types of conditions and options are available in the problem. In this unit, you will study about what a function is.

A function is a “black box'' that we've locked part of our program into. The idea behind a function is that it *compartmentalizes* part of the program, and in particular, that the code within the function has some useful properties:

It performs some well-defined task, which will be useful to other parts of the program. It might be useful to other programs as well; that is, we might be able to reuse it (and without having to rewrite it).

The rest of the program doesn't have to know the details of how the function is implemented. This can make the rest of the program easier to think about.

The function performs its task *well*. It may be written to do a little more than is required by the first program that calls it, with the anticipation that the calling program (or some other program) may later need the extra functionality or improved performance. (It's important that a finished function do its job well, otherwise there might be a reluctance to call it, and it therefore might not achieve the goal of reusability.)

By placing the code to perform the useful task into a function, and simply calling the function in the other parts of the program where the task must be performed, the rest of the program becomes clearer: rather than having some large, complicated, difficult-to-understand piece of code repeated wherever the task is being performed, we have a single simple function call, and the name of the function reminds us which task is being performed.

Since the rest of the program doesn't have to know the details of how the function is implemented, the rest of the program doesn't care if the function is re-implemented later, in some different way (as long as it continues to perform its same task, of course!). This means that one part of the program can be rewritten, to improve performance or add a new feature (or simply to fix a bug), without having to rewrite the rest of the program.

Functions are probably the most important weapon in our battle against software complexity. You'll want to learn when it's appropriate to break processing out into functions (and also when it's not), and *how* to set up function interfaces to best achieve the qualities mentioned above: reusability, information hiding, clarity, and maintainability.

1.1 Objectives

After studying this unit, you should be able to:

- *explain the importance of functions*
- *implement the concepts of formal arguments and actual arguments*
- *explain function declaration(function prototypes) and function definition*
- *use the concept of recursion*
- *explain how the concept of functions reduces software complexity*

2. FUNCTION BASICS

A function is a self-contained program segment that carries out some specific, well-defined task. Every C program contains one or more functions. One of these functions must be called **main**. Program execution will always begin by carrying out the instructions in main. Additional functions will be subordinate to main, and perhaps to one another.

So what defines a function? It has a *name* that you call it by, and a list of zero or more *arguments* or *parameters*. Parameters (also called formal parameters) or arguments are the special identifiers through which information can be passed to the function. A function has a *body* containing the actual instructions (statements) for carrying out the task the function is supposed to perform; and it may give you back a *return value*, of a particular type.

In general terms, the first line can be written as
data-type name(data-type parameter 1, data-type parameter 2, ..., data-type parameter n)

Example 5.1: Here is a very simple function, which accepts one argument, multiplies it by 4, and hands that value back.

```
int multbyfour(int x)
{
        int retval;
        retval = x * 4;
        return retval;

}
```

On the first line we see the return type of the function (int), the name of the function (multbyfour), and a list of the function's arguments, enclosed in parentheses. Each argument has both a name and a type; **multbyfour** accepts one argument, of type **int**, named x. The name x is arbitrary, and is used only within the definition of **multbyfour.** The caller of this function only needs to know that a single argument of type **int** is expected; the caller does not need to know what name the function will use internally to refer to that argument. (In particular, the caller does not have to pass the value of a variable named x.)

Next we see, surrounded by the familiar braces, the body of the function itself. This function consists of one declaration (of a local variable *retval*) and two statements. The first statement is a conventional expression statement, which computes and assigns a value to *retval*, and the second statement is a return statement, which causes the function to *return* to its caller, and also specifies the value which the function returns to its caller.

In general term, a *return* statement is written as

return expression

The *return* statement can return the value of any expression, so we don't really need the local *retval* variable; this function can also be written as

```
int multbyfour(int x)
{
        return x * 4;

}
```

How do we call a function? We've been doing so informally since day one, but now we have a chance to call one that we've written, in full detail. The arguments in the function call are referred to as *actual arguments or actual parameters,* in contrast to the formal arguments that appear in the first line of the function definition.

Here is a tiny skeletal program to call multbyfour:

```
#include <stdio.h>
extern int multbyfour(int);
int main()
{
        int i, j;
        i = 5;
        j = multbyfour(i);
        printf("%d\n", j);
        return 0;

}
```

This looks much like our other test programs, with the exception of the new line

```
extern int multbyfour(int);
```

This is an *external function prototype declaration*. It is an external declaration, in that it declares something which is defined somewhere else. (We've already seen the defining instance of the function **multbyfour**, but may be the compiler hasn't seen it yet.) The function prototype declaration contains the three pieces of information about the function that a caller needs to know: the function's name, return type, and argument type(s). Since we don't care what name the **multbyfour** function will use to refer to its first argument, we don't need to mention it. (On the other hand, if a function takes several arguments, giving them names in the prototype may make it easier to remember which is which, so names may optionally be used in function prototype declarations.) Finally, to remind us that this is an external declaration and not a defining instance, the prototype is preceded by the keyword **extern.**

The presence of the function prototype declaration lets the compiler know that we intend to call this function, **multbyfour.** The information in the prototype lets the compiler generate the correct code for calling the function, and also enables the compiler to check up on our code (by making sure, for example, that we pass the correct number of arguments to each function we call).

Down in the body of main, the action of the function call should be obvious:
the line

```
j = multbyfour(i);
```

calls B, passing it the value of i as its argument. When **multbyfour** returns, the return value is assigned to the variable j. (Notice that the value of main's local variable i will become the value of **multbyfour's** parameter x; this is absolutely not a problem, and is a normal sort of affair.)

This example is written out in “longhand,'' to make each step equivalent. The variable i isn't really needed, since we could just as well call

```
j = multbyfour(5);
```

And the variable j isn't really needed, either, since we could just as well call

```
printf("%d\n", multbyfour(5));
```

Here, the call to **multbyfour** is a subexpression which serves as the second argument to **printf.** The value returned by **multbyfour** is passed immediately to **printf.** (Here, as in general, we see the flexibility and generality of expressions in C. An argument passed to a function may be an arbitrarily complex subexpression, and a function call is itself an expression which may be embedded as a subexpression within arbitrarily complicated surrounding expressions.)

We should say a little more about the mechanism by which an argument is passed down from a caller into a function. Formally, C is *call by value*, which means that a function receives *copies* of the values of its arguments. We can illustrate this with an example. Suppose, in our implementation of **multbyfour,** we had gotten rid of the unnecessary **retval** variable like this:

```
int multbyfour(int x)
{
        x = x * 4;
        return x;

}
```

We might wonder, if we wrote it this way, what would happen to the value of the variable i when we called

```
j = multbyfour(i);
```

When our implementation of **multbyfour** changes the value of x, does that change the value of i up in the caller? The answer is no. x receives a copy of i's value, so when we change x we don't change i.

However, there is an exception to this rule. When the argument you pass to a function is not a single variable, but is rather an array, the function does not receive a copy of the array, and it therefore can modify the array in the caller. The reason is that it might be too expensive to copy the entire array, and furthermore, it can be useful for the function to write into the caller's array, as a way of handing back more data than would fit in the function's single return value. We will discuss more about passing arrays as arguments to a function later.

There may be several different calls to the same function from various places within a program. The actual arguments may differ from one function call to another. Within each function call, however, the actual arguments must correspond to the formal arguments in the function definition; i.e the actual arguments must match in number and type with the corresponding formal arguments.

Program 5.1: A program to find the largest of three integers

```
#include<stdio.h>
main()
{
        int x, y, z, w;
        /* read the integers */
        int max(int, int);
        printf("\nx= ");
        scanf("%d", &x);
        printf("\ny= ");
        scanf("%d", &y);
        printf("\nz= ");
        scanf("%d", &z);
        /* Calculate and display the maximum value */
        w= max(x,y);
        printf("\n\nmaximum=%d\n", max(z,w));
}
int max(int a, int b)
{
        int c;
        c=(a>=b)?a:b;
        return c;

}
```

Please execute this program and observe the result.

Function calls can span several levels within a program; function A can call function B, which can call function C and so on.

Program 5.2: Program to check whether a given integer is a perfect square or not.

```
#include<stdio.h>
main()
{
        int psquare(int);
        int num;
        printf(" Enter the number:");
        scanf("%d", &num);
        if(psquare(num))     /* main() calls the function psquare() */
        {
                printf("%d is a perfect square\n");
        else
                printf("%d is not a perfect square\n");
        }
}
```

```
int psquare(int x)
{
int positive(int);
float sqr;
if(positive(x)) /* psquare() in turn calls the function
positive() */ {
        sqr=sqrt(x));
        if(sqr-int(sqr))==0)
                return 1;
        else
                return 0;
}
int positive(int m)
{
        if(m>0)
                return 1;
        else return 0;
}
```

Execute the above program and observe the result.

In the above program the **main** function calls the function psquare() and it in turn calls the function positive() to check whether the number to be checked for perfect square is a positive or not before checking.

The **return** statement can be absent altogether from a function definition, though this is generally regarded as a poor programming practice. If a function reaches end of the block without encountering a return statement, control simply reverts back to the calling portion of the program without returning any information. Using an empty return statement(without the accompanying expressions) is recommended.

Example 5.2: The following function accepts two integers and determines the larger one, which is then written out. The function doesn't return any information to the calling program.

```
void max(int x, int y)
{
        int m;
        m=(x>=y)?x:y;
        printf(" The larger integer is=%d\n", m);
        return;

}
```

SELF-ASSESSMENT QUESTIONS – 1

1. The function main() is optional in a C program. (True/False)
2. If the function is defined elsewhere (not in the same program where it is called), the function prototype must be preceded by the keyword _____.
3. The arguments that appear in function definition are called_____ arguments whereas the arguments that appear when the function is called are the _____ arguments.
4. The return data type, function name and the list of formal parameters enclosed in brackets are separated by _____.

3. FUNCTION PROTOTYPES

In modern C programming, it is considered good practice to use prototype declarations for all functions that you call. As we mentioned, these prototypes help to ensure that the compiler can generate correct code for calling the functions, as well as allowing the compiler to catch certain mistakes you might make.

In general terms, a function prototype can be written as *data-type name(type1, type2, …, type n*) Examples:
int sample(int, int) or int sample(int a, int b);
float fun(int, float) or float fun(int a, float b);

void demo(void); Here **void** indicates function neither return any value to the caller nor it has any arguments.

If you write the function definition after the definition of its caller function, then the prototype is required in the caller, but the prototype is optional if you write the definition of the function before the definition of the caller function. But it is good programming practice to include the function prototype wherever it is defined.

If prototypes are a good idea, and if we're going to get in the habit of writing function prototype declarations for functions we call that we've written (such as multbyfour), what happens for library functions such as **printf?** Where are their prototypes? The answer is in that boilerplate line

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

we've been including at the top of all of our programs. stdio.h is conceptually a file full of external declarations and other information pertaining to the "Standard I/O'' library functions, including **printf.** The #include directive arranges all of the declarations within **stdio.h** that are considered by the compiler, rather as if we'd typed them all in ourselves. Somewhere within these declarations is an external function prototype declaration for **printf,** which satisfies the rule that there should be a prototype for each function we call. (For other standard library functions we call, there will be other "header files'' to include.) Finally, one more thing about external function prototype declarations: we've said that the distinction between external declarations and defining instances of normal variables hinges

on the presence or absence of the keyword extern. The situation is a little bit different for functions. The “defining instance'' of a function is the function, including its body (that is, the brace-enclosed list of declarations and statements implementing the function). An external declaration of a function, even without the keyword **extern**, looks nothing like a function declaration. Therefore, the keyword **extern** is optional in function prototype declarations. If you wish, you can write

int multbyfour(int);

and this is just like an external function prototype declaration as

extern int multbyfour(int);

(In the first form, without the **extern**, as soon as the compiler sees the semicolon, it knows it's not going to see a function body, so the declaration can't be a definition.) You may want to stay in the habit of using **extern** in all external declarations, including function declarations, since “extern = external declaration'' is an easier rule to remember.

Program 5.3: Program to illustrate that the function prototype is optional in the caller function. The program is to convert a character from lowercase to uppercase.

```
#include<stdio.h>
char lower_to_upper(char ch) /* Function definition precedes main()*/
{
        char c;
        c=(ch>='a' && ch<='z') ? ('A'+ch-'a'):ch;
        return c;
}

main()
{
        char lower, upper;
   /* char lower_to_upper(char lower); */ /* Function prototype is
optional here*/
        printf("Please enter a lowercase character:");
        scanf("%c", &lower);
        upper=lower_to_upper(lower);
        printf("\nThe uppercase equivalent of %c is %c\n", lower, upper);
}
```

SELF-ASSESSMENT QUESTIONS – 2

5. Function prototype is also called _____.
6. The function prototypes are optional. (True/False)
7. The function prototypes of the library functions is in the corresponding _____ files.
8. The function prototype for the function fun() called in main() below is _____.

```
main()
    {
        double x, y, z;
        …
        z=fun(x, y);
        …
    }
```

4. RECURSION

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been met. The process is used for repetitive computations in which each action is stated in terms of a previous result. Many repetitive problems can be written in this form.

In order to solve a problem recursively, two conditions must be satisfied. First, the problem must be written in a recursive form, and the second, the problem statement must include a stopping condition.

Example 5.3: Factorial of a number. Suppose we wish to calculate the factorial of a positive integer, n. We would normally express this problem as n!=1 x 2 x 3 x … x n.

This can also be written as n!=n x (n-1)!. This is the recursive statement of the problem in which the desired action(the calculation of n!) is expressed in terms of a previous result (the value of (n-1)! which is assumed to be known). Also, we know that 0!=1 by definition. This expression provides stopping condition for the recursion.

Thus the recursive definition for finding factorial of positive integer n can be written as:

fact(n)={ 1 if n=0

n x fact(n-1) otherwise}

Program 5.4: Program to find factorial of a given positive integer

```
#include<stdio.h>
main()
{
        int n;
        long int fact(int);
        /* Read in the integer quantity*/
        scanf("%d", &n);
        /*calaculate and display the factorial*/
        printf("n!=%ld\n", fact(n));
}
long int fact(int n)
{
        if(n==0)
                return(1);
```

```
        else
                return (n*fact(n-1));
}
```

Please execute this program and observe the result.

Example 5.4: The Towers of Hanoi. The Towers of Hanoi is a game played with three poles and a number of different sized disks. Each disk has a hole in the center, allowing it to be stacked around any of the poles. Initially, the disks are stacked on the leftmost pole in the order of decreasing size, i.e, the largest on the bottom, and the smallest on the top as illustrated in Figure 5.1.

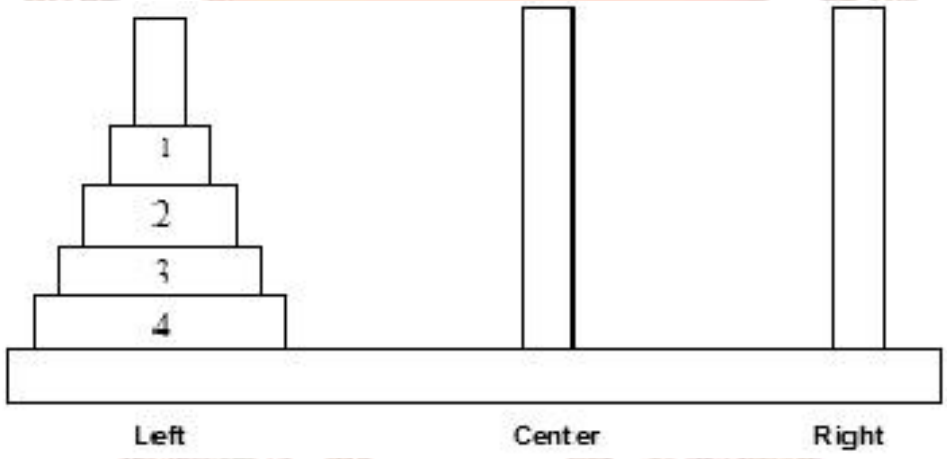

Fig 5.1: Illustration for Tower of Hanoi

The aim of the game is to transfer the disks from the leftmost pole to the rightmost pole, without ever placing a larger disk on top of a smaller disk. Only one disk may be moved at a time, and each disk must always be placed around one of the poles.

The general strategy is to consider one of the poles to be the origin, and another to be the destination. The third pole will be used for intermediate storage, thus allowing the disks to be moved without placing a larger disk over a smaller one. Assume there are n disks, numbered from smallest to largest as in Figure 5.1. If the disks are initially stacked on the left pole, the problem of moving all n disks to the right pole can be stated in the following recursive manner:

1. Move the top n-1 disks from the left pole to the center pole.
2. Move the n^{th} disk(the largest disk) to the right pole.
3. Move the n-1 disks on the center pole to the right pole.

The problem can be solved for any value of n greater than 0(n=0 represents a stopping condition).

In order to program this game, we first label the poles, so that the left pole is represented as L, the center pole as C and the right pole as R. Let us refer the individual poles with the char-type variables **from, to** and **temp** for the origin, destination and temporary storage respectively.

Program 5.5: Recursive Program to solve Towers of Hanoi problem.

```
#include<stdio.h>
main()
{
        void Recursive_Hanoi(int, char, char, char);
        int n;
        printf(" Towers of Hanoi\n\n");
        printf(" How many disks?");
        scanf("%d", &n);
        printf("\n");
        Recusrive_Hanoi(n, 'L', 'R', 'C');
}
void Recursive_Hanoi(int n, char from, char to, char temp)
{
/* Transfer n disks from one pole to another */
/*  n= number of disks
   from=origin
   to=destination
   temp=temporary storage */
{
        if(n>0){
                /* move n-1 disks from origin to temporary */
                Recursive_Hanoi(n-1, from, temp, to);

                /* move nth disk from origin to destination */
                printf(" Move disk %d from %c to %c\n", n, from, to);
```

```
                /* move n-1 disks from temporary to
                destination */ Recursive_Hanoi(n-1, temp, to,
                from);
        }
        return;
}
```

Please execute this program and observe the result

SELF-ASSESSMENT QUESTIONS – 3

9. _____ is a process by which a function calls itself repeatedly, until some specified condition is satisfied
10. A stopping condition must be there in a recursive definition. (True/False)
11. The output of the following program is _____.

```
#include<stdio.h>
main()
{
int n=5;
int fun(int n);
printf("%d\n", fun(n));
}
int fun(int n)
{
if(n==0)
return 0;
else
return (n+fun(n-1));
}
```

5. FUNCTION PHILOSOPHY

What makes a good function? The most important aspect of a good “building block'' is that have a single, well-defined task to perform. When you find that a program is hard to manage, it's often because it has not been designed and broken up into functions cleanly. Two obvious reasons for moving code down into a function are because:

1. It appeared in the main program several times, such that by making it a function, it can be written just once, and the several places where it used to appear can be replaced with calls to the new function.
2. The main program was getting too big, so it could be made (presumably) smaller and more manageable by lopping part of it off and making it a function.

   These two reasons are important, and they represent significant benefits of well-chosen functions, but they are not sufficient to automatically identify a good function. As we've been suggesting, a good function has at least these two additional attributes:
3. It does just one well-defined task, and does it well.
4. Its interface to the rest of the program is clean and narrow.

Attribute 3 is just a restatement of two things we said above. Attribute 4 says that you shouldn't have to keep track of too many things when calling a function. If you know what a function is supposed to do, and if its task is simple and well-defined, there should be just a few pieces of information you have to give it to act upon, and one or just a few pieces of information which it returns to you when it's done. If you find yourself having to pass lots and lots of information to a function, or remember details of its internal implementation to make sure that it will work properly this time, it's often a sign that the function is not sufficiently well-defined. (A poorly-defined function may be an arbitrary chunk of code that was ripped out of a main program that was getting too big, such that it essentially has to have access to all of that main function's local variables.)

The whole point of breaking a program up into functions is so that you don't have to think about the entire program at once; ideally, you can think about just one function at a time. We say that a good function is a “black box,'' which is supposed to suggest that the “container'' it's in is opaque – callers can't see inside it (and the function inside can't see out). When you call a function, you only have to know what it does, not how it does it. When you're writing a

function, you only have to know what it's supposed to do, and you don't have to know why or under what circumstances its caller will be calling it. (When designing a function, we should perhaps think about the callers just enough to ensure that the function we're designing will be easy to call, and that we aren't accidentally setting things up so that callers will have to think about any internal details.)

Some functions may be hard to write (if they have a hard job to do, or if it's hard to make them do it truly well), but that difficulty should be compartmentalized along with the function itself. Once you've written a “hard'' function, you should be able to sit back and relax and watch it do that hard work on call from the rest of your program. It should be pleasant to notice (in the ideal case) how much easier the rest of the program is to write, now that the hard work can be deferred to this workhorse function.

(In fact, if a difficult-to-write function's interface is well-defined, you may be able to get away with writing a quick-and-dirty version of the function first, so that you can begin testing the rest of the program, and then go back later and rewrite the function to do the hard parts. As long as the function's original interface anticipated the hard parts, you won't have to rewrite the rest of the program when you fix the function.)

The functions are important for far more important reasons than just saving typing. Sometimes, we'll write a function which we only call once, just because breaking it out into a function makes things clearer and easier.

If you find that difficulties pervade a program, that the hard parts can't be buried inside black-box functions and then forgotten about; if you find that there are hard parts which involve complicated interactions among multiple functions, then the program probably needs redesigning.

For the purposes of explanation, we've been seeming to talk so far only about “main programs'' and the functions they call and the rationale behind moving some piece of code down out of a “main program'' into a function. But in reality, there's obviously no need to restrict ourselves to a two-tier scheme. Any function we find ourselves writing will often be appropriately written in terms of sub-functions, sub-sub-functions, etc.

Program 5.6: Program to create a function that types 65 asterisks in a row

```
/* letterhead1.c */
#include <stdio.h>
#define NAME "MEGATHINK, INC."
#define ADDRESS "10 Megabuck Plaza"
#define PLACE "Megapolis, CA 94904"
#define LIMIT 65
void starbar(void);  /* prototype the function */
int main(void)
{
starbar();
printf("%s\n", NAME);
printf("%s\n", ADDRESS);
printf("%s\n", PLACE);
starbar();       /* use the function */
return 0;
}
void starbar(void)
{
/* define the function */
int count;
for (count = 1; count <= LIMIT; count++)
putchar('*');
putchar('\n');
}
```

SELF-ASSESSMENT QUESTIONS – 4

12. By modularizing the problem into different sub problems. Each sub problem can be implemented as a _____.
13. The main purpose of function is to save typing time. (True/False)

6. SUMMARY

A function is a self-contained program segment that carries out some specific, well-defined task. When you find that a program is hard to manage, it's often because it has not been designed and broken up into functions cleanly. A function is a “black box'' that we've locked part of our program into. The idea behind a function is that it compartmentalizes part of the program. The function main() is must in every C program. The function prototype is nothing but the function declaration. Recursion is a process by which a function calls itself repeatedly, until some specified condition has been met.

7. TERMINAL QUESTIONS

1. What is the significance of the keyword ‘void’?
2. What is the difference between function declaration and function definition?
3. Write a recursive function to find sum of even numbers from 2 to 10.
4. Write a recursive definition to find gcd of two numbers.
5. Write a recursive definition to find nth fibonacci number. The Fibonacci series forms a sequence of numbers in which each number is equal to the sum of the previous two numbers. In other words, $F_i=F_{i-1} + F_{i-2}$

   where Fi refers to the i^{th} Fibonacci number. The first two Fibonacci numbers are 0 and 1, i.e, $F_1=0$, $F_2=1$;

6. What is the output of the following program?

```
#include<stdio.h>
main()
{
int m, count;
int fun(int count);
for(count=1;count<=10;count++) {
m=fun(count);
printf(“%d”, m);
}
}
```

```
int fun(int n)
{
int x;
x= n*n;
return x;
}
```

8. ANSWERS

Self-Assessment Questions

1. False
2. extern.
3. formal, actual
4. comma.
5. Function declaration
6. False
7. header
8. double fun(double, double);
9. Recursion.
10. True
11. 15
12. function
13. False

Terminal Questions

1. 'void' is the keyword used to specify that the function doesn't return any value. It can also be used to specify the absence of arguments.
2. Function declaration is a direction to the compiler that what type of data is returned by the function, the function name and about the arguments where as the function definition is actually writing the body of the function along with the function header.
3. #include<stdio.h>
   main()
   {
   int n=10;
   int fun(int n); printf("%d", fun(n));
   }
   int fun(int n)
   {
   if(n>0) return (n+fun(n-2));
   }
4. gcd(m,n)= { m or n if m=n
   GCD(m, m-n) if m>n
   GCD(n,m) if m<n }
5. fib(i)= { 0 if i=1 1 if i=2
   fib(i-1)+fib(i-2) otherwise}
6. Square of the integers from 1 to 10 is displayed.

9. EXERCISES

1. Suppose function F1 calls function F2 within a C program. Does the order of function definitions make any difference? Explain.
2. When a program containing recursive function calls is executed, how are the local variables within the recursive function interpreted?
3. Express the following algebraic formula in a recursive form: $Y = (x_1+x_2+...+x_n)$
4. Write a function that will allow a floating point number to be raised to an integer power.
5. Write a function to swap two numbers using pass by value technique. What is the drawback of the function?