CHAPTER TEN: C- Programming:- User Defined Functions

Chapter Objectives

By the end of this chapter the learner should be able to

By the end of the chapter the user should be able to;

- Describe and design a function
- Describe function integration and integrate a function into a program
- Describe how functions communicate

10.1. Introduction

C functions can be classified into two categories; ***library functions*** and ***user-defined functions***. Library functions are not required to be written by the programmers, while user-defined functions have to be developed by the user at the time of writing the C program. ***main function*** is an example of user-defined functions while ***scanf*** and ***printf functions*** are examples of library functions. Some user-defined functions can eventually become library functions.

10.2. Need for user-defined functions

Large programs are divided in functional parts (figure 10.1), and each of the functional part is code separately and independently but later combined into a single unit. The independently coded programs are called sub-programs, and in C they are referred to as **"functions"**. The functions can be called and used whenever needed. The division of large programs into sub-programs have the following advantages;

1. It facilitates top-down modular programming. The high level logic of the overall problem is solved first while the details of each lower-level functions are addressed later.
2. The length of a source program can be reduced by using functions at appropriate places. This factor is particularly critical with microcomputers where memory space is limited.
3. It is easy to locate and isolate a faulty function for further investigations.
4. A function may be used by many other programs. This means that a C programmer can build on whatever others have already done, instead of starting all over again from scratch.

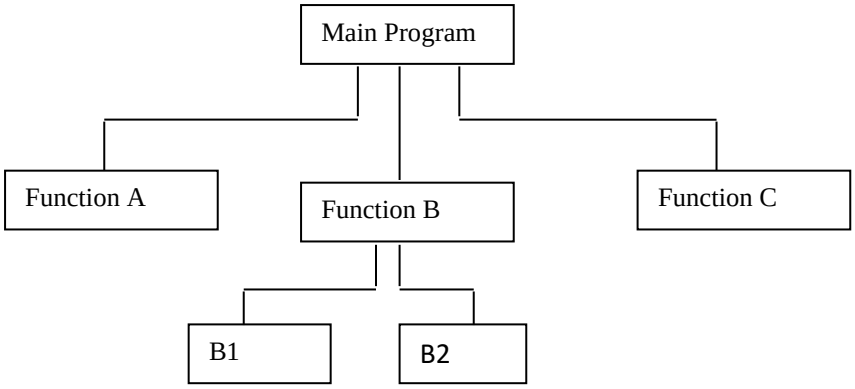


Figure 10.1 Top-down modular programming using functions.

10.3. Modular programming;

A strategy applied to the design and development of software systems. It is defined as organizing a large program into small, independent program segments called modules that are separately named and individually callable program units. These modules are carefully integrated to become a software system that satisfies the system requirements. It is basically a "divide – and –conquer" approach to problem solving. Modules are identified and designed such that they can be organized into a to-down hierarchical structure (similar to an organization chart).

Characteristics of modular programming;

1. Each module should do only one thing
2. Communication between modules is allowed only by a calling module.
3. A module can be called by one and only one higher module
4. No communication can take place directly between modules that do not have calling – called relationship.
5. All modules are designed as single-entry-exit systems using control structures.

10.4. A Multi-Function Program.

A function is a self-contained block of code that performs a particular task. A C program is designed using a collection of functions.

Program Example 10.1: Program to illustrate the use of Functions

```
/* documentation section*/
#include<string.h>
```

```
#include<stdio.h>
#include<conio.h>
Void printline(void);  /* declares a function*/
main()
{
printline();        /* first call to the printline function */
      printf("This illustrated the use of C functions");
printline();   /* second call to the printline function */
}
void printline(void)    /* printline function  */
{
int i;
for(i =1; i<40, i++)
printf("-");
printf("\n");
}
```

Out put

This Illustrates the use of C functions

The program contain two user-defined functions ***main()*** function and ***printline()*** function. In the above program the main function call the user-defined function printline twice and the library function printf() once. A called function can also call another function example printline function in the above example calls printf() function 39 times to draw a line.

10.5. Elements of User-Defined Functions

Functions in C are classified as one of the derived data types in C. Functions can be defined and used like any other variable in C programs.

Similarities between functions and variables

1. Both function name and variable names are considered identifiers and therefore they must adhere to the rules of identifiers.
2. Like variables, functions have types (such as int) associated with them.

3. Like variables, function names and their types must be declared and defined before they are used in a program.

A user-defined functions consists of thee elements;

1. **Function Definition:** an independent program module that is specially written to implement the requirements of the function.
2. **Function Call:** invoking a function at an appropriate place in the program. The function/ module that call a function is referred to as the ***calling function*** or ***calling program***
3. **Function declaration:** declaration of the function within the calling function that will be used later. Also referred to as ***function prototype.***

10.6. Definition of Functions

A function definition also known as function implementation has the following elements;

1. Function name
2. Function type
3. List of parameters.
4. Local variable declarations
5. Function statements; and
6. A return statement.

All the six elements are grouped together into two parts, namely; function header and function body. The general format for a function definition is as follows;

```
function_type  function_name(parameter list)       /* function header */
{                                                  /*  function body */
        local variable declaration;
        executable statement 1'
        executable statement 2;
        ------
        -------
         return statement;
}
```

10.6.1 Function header

The functional header consists of the first three (3) elements; function type (also called return type); function name and the formal parameter list. No semi colon is used at the end if the function header section.

Name and Type

Function type specifies the type of value (int, float, void) that the function is expected to return to the calling function. If not stated C assumes int (integer) data type. If the function is not returning anything we should specify return type as void.

Example printline(void); or printline();

Function Name is any valid C identifier and thus must follow the same rules of formulation as other variable names in C.

Formal Parameter List

Parameter list declares the variables that will receive data sent by the calling program. They serve as input data to the function to carry out the specified task. They are called **Formal parameter** list since they represent actual input values. The parameters can also be used to send data to the calling program where they are referred to as **Arguments.**

10.6.2. Function Body

Contain the declaration and statements necessary for performing the required task. It is enclosed on braces { } and contain three parts;

1. Local declarations that specify the variables needed by the function
2. Function statements that perform the task of the function
3. Return statement that returns the value evaluated by the function.

Examples of functions definition;

a). float mul (float x, float y)

```
{
        float results;          /* declare local variable*/
        results = x * y;        /*computes  the products*/
        return (results);       /*returns the results*/
}
```

b). void sum(int a, intb)

```
{
  printf("Sum = %s", a+b);        /* no local variables */
  return;                         /* Optional */
}
```

Note;

1. If a function is not returning any value one can omit the return statement or use the return(void) statement.
2. When a program reaches the return statement, the control is transferred back to the calling program. In the absence of the return statement the closing braces } acts as avoid return.
3. A local variable is a variable that is defined inside a function and used without having any role in the communication between functions.

10.7. Return values and their types

While it is possible to pass to the called function any number of values, the called function can only return ***one value*** per call at the most. The return statement can take either of the following forms;

return; or return(expression);

- return; does not return any value and only acts as the closing brackets for the function,
- return(expression); returns the value and once reached the program control is immediately returns to the calling function

A function may have more than one return statements, especially when the value returned is based on certain conditions. Example

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

All functions by default returns **int** type of data. We can force a function to return another type of data type by using the type specifier in the function header. When a value is returned, it is automatically cast to the function's type

10.8. Function call

A function may be called by simply using the name followed by a list of actual parameters (or arguments) if any enclosed in parenthesis. Example

```
main()
{
int y;
y = mul(10,5)     /* Function Call */
printf("%d\n", y);
}
        int mul (int x, int y)   /* Mul function */
        {
                int p;
                p = x*y;
                return (p);
          }
```

Function Call

A function call is a postfix expression. The operator (..) is at a very high level of precedence. Therefore, when a function call is used as a part of an expression, it will be evaluated first, unless parentheses are used to change the order of precedence.

In a function call, the function name is the operand and the parenthesis set (..) which contains the actual parameters is the operator. The actual parameters must match the function's formal parameters in type, order and number. Multiple actual parameters must be separated by commas.

Note

1. If the actual parameters are more than the formal parameters, the extra actual arguments will be discarded.
2. On the other hand, if the actual are less than the formals, the unmatched formal arguments will be initialized to some garbage.
3. Any mismatch in data types may also result in some garbage values.

Program Example 10.2. : Program Using Function

```
/* Program using function          comment  */
#include <stdio.h>
#include<conio.h>
int mul (int a, int b);
int main()                    /* function body*/
{
 int a, b, c;
```

```
 a = 5;
 b = 10;
 c = mul (a,b);                          /* function call*/
 printf("Multiplication of %d and %d is %d", a,b,c);
getch();
}
                                  /* mul()       Sub-program  */
   int mul(int x,int y)
     {
      int p;
      p = x*y;
      return (p);
    }
```

10.9. Function Declaration

Like variables, all functions in a C Program must be declared, before they are invoked. A function declaration consists of four parts; Function type (return type), Function name, Parameter list and Terminating semicolon. The function declaration statement takes the following form

function_type function-name(parameter list);

Note

1. The parameter list must be separated by commas.
2. The parameter names do not need to be the same in the prototype declaration and the function definition.
3. The types must match the types of parameters in the function definition, in number and order
4. Use of parameter names in the declaration is optional
5. If the function has no formal parameters, the list is written as (void).
6. The return type is optional, when the function returns **int** type data.
7. The retype must be **void** if no value is returned.
8. When the declared types do not match with the types in the function definition, compiler will produce an error.

When a function does not take any parameters and does not return any value, its prototype is written as;

- ***void display(void);***

A prototype declaration may be placed in two places in a program.

1. Above all the functions (including main): referred to as ***Global prototype*** and is available to all functions in the program.
2. Inside a function definition: referred to as ***Local prototype*** and is available only to the local function.

The place of declaration of a function defines a region in a program in which the function may be used by other functions. This region is known as the scope of the function. Declare the prototypes in the global variables declaration section of a C program (above the main () function) to add flexibility, provide an excellent quick reference to the functions use in the program, and enhance documentation.

Parameters

Parameters (also known as arguments) are used in three places.

1. In declaration (prototypes)
2. In function call
3. In function definition

The parameters use in prototypes and function definitions are called formal parameters and those used in calling statements may be simple constants, variables or expressions.
The formal and actual parameters must match exactly in type, order and number. Their names, however, do not need match.

10.10. Categories of Functions

A function depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories

Category 1: Functions with no arguments and no return values

Category 2: Functions with arguments and no return values.

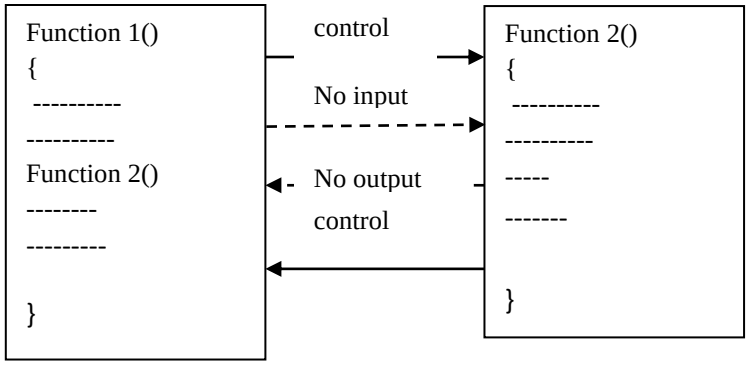
Category 3: Functions with arguments and one return value.

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

Category 5: Functions that return multiple values.

Category 1: Functions with no arguments and no return values

When a function has no arguments, it does not receive any data from the calling function. When it does not return an value the calling function does not receive any data from the called function. Thus there is no data transfer between the calling and called functions, there is only transfer of control.



No data communication between functions

Program Example 10.3. Program to show Functions with no arguments and no return values.

```
/* Program to calculate the value of  principal in a bank after a certain period of time */
#include <stdio.h>
#include<conio.h>
                                    /* functions declaration */
void printline (void);
void value(void);
main()
{
printline();
value();
printline();
}
                                    /* function  print line */
void printline(void)     /* contains no arguments */
{
 int i;
for (i =1; i<=35; i++)
printf("%c", '-');
printf("\n");
}
                                    /* function 2: value()    */
```

```
void value (void)   /* Contains no arguments */
{
int year, period;
float inrate, sum, principal;
printf("Principal Amount?");
scanf("%f", &principal);
printf("Interest rate?       ");
scanf("%f", &inrate);
printf("Period?      ");
scanf("%d", &period);

sum = principal;
year = 1;
while(year <= period)
{
sum = sum *(1+inrate);
year = year + 1;
}
printf ("\n%8.2f  %5.2f  %5d  %12.2f\n", principal, inrate, period, sum);
getch();

}
```

Test data

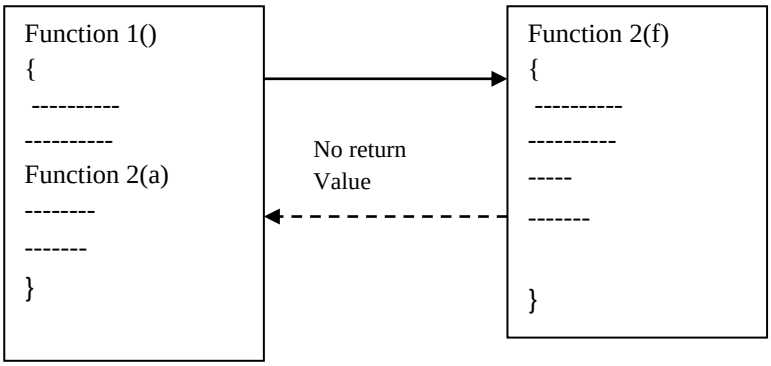
Principal = 50000
Interest = 0.12
Period = 5

Program Out put

```
--------------------------------------------------
Principal amount      5000
Interest rate?        0.12
Period                      5
5000.00               0.12          5        8811.71
--------------------------------------------------------------------------------
```

Category 2: Arguments but no Return Values

Values of
Arguments



One-way data communication

The calling function passes on a list or arguments to the called function. In the last example, the following changes will be done at the definitions of the functions;

void (printline(char ch)

void value (float p, float r, int n)

ch, p,r, &bn are called formal arguments. The calling function can thus send values to the arguments using the function call value (5000, 0.12, 5). The values 5000, 0.12, & 5 are called actual arguments

The formal and actual arguments should match in number, type and order

Program Example 10.4. Program to demonstrate functions with arguments and no return Value.

```
/* Program to calculate the value of  principal in a bank after a certain period of time */
#include <stdio.h>
#include<conio.h>

void printline (char c);
void value(float, float, int);
main()
{
float principal, inrate;
int period;
printf("Enter Principal amount:  ");
scanf("%f", &principal);
printf("Enter Interest rate?  ");
scanf("%f", &inrate);
printf("Enter Period?      ");
scanf("%d", &period);
```

```
printline('-');
value(principal, inrate, period);
printline('-');
}
void printline(char ch)
{
 int i;
for (i =1; i<=52; i++)
printf("%c", ch);
printf("\n");
}

void value (float p, float r, int n)     /* value function */
{
int year;
float sum;

sum = p;
year = 1;
while(year <= n)
{
sum = sum *(1+r);
year = year + 1;
}
printf ("\%8.2f\t  %5.2f\t  %d\t  %8.2f\n", p, r, n, sum);
getch();
}
```

Test data

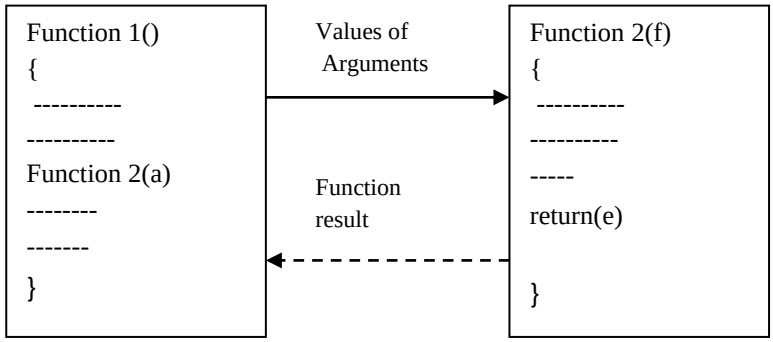
Principal = 50000
Interest = 0.12
Period = 5

Program Out put

```
--------------------------------------------------
Principal amount      5000
Interest rate?        0.12
Period                      5

5000.00               0.12          5       8811.71
--------------------------------------------------------------------------------
```

Category 3: Arguments with Return Values



Two- way data communication between functions

Program Example 10.5. Program to demonstrate functions with Arguments and Return Values.

```
/* Program to calculate the value of  principal in a bank after a certain period of time */
#include <stdio.h>
#include<conio.h>

int value(float, float, int);
void printline (char c, int len);

main()
{
 float principal, inrate;
 int period, amount;
```

```
 printf("Enter Principal amount:  ");
 scanf("%f", &principal);
 printf("Enter Interest rate?  ");
 scanf("%f", &inrate);
 printf("Enter Period?      ");
 scanf("%d", &period);

 printline('*', 52);
 amount = value(principal, inrate, period);
 printf ("\%8.2f\t  %5.2f\t  %d\t %d\n\n", principal, inrate, period,amount);

 printline('=', 52);
 getch();
}
void printline(char ch, int len)
{
 int i;
 for (i =1; i<=len; i++)
 printf("%c", ch);
 printf("\n");
}

int value (float p, float r, int n)     /* value function  */
{
int year;
float sum;
sum = p;
year = 1;
while(year <= n)
{
sum = sum *(1+r);
year = year + 1;
}
return(sum);
```

```
}
```

Test data

Principal = 50000
Interest = 0.12
Period = 5

Out put

```
--------------------------------------------------
Principal amount      5000
Interest rate?        0.12
Period                      5

5000.00               0.12            5       8811.71
--------------------------------------------------------------------------------
```

10.11. Nesting of Functions

C permits nesting of functions freely, ie. main() can call function1(), which calls function2() which calls function3()

Program Example 10.6: Program to demonstrate Nesting of Functions

/* Program to demonstrate Nesting of Functions calculation of ratios*/

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

float ratio (int x, int y, int z);
int difference (int x, int y);

main()
{
        int  a, b, c;
        printf("Enter values for a, b,& c\n");
        scanf("%d  %d %d", &a, &b, &c);
        printf("%f\n", ratio(a,b,c));
        getch();
}

        float ratio (int x, int y, int z)
        {
```

```
        if(difference(y,z))  /* difference function checks if    b-c = 0 */
        return (x/(y-z));
else
        return(0.0);
}
int difference (int p, int q)
{
  if (p != q)
        return (1);
else
        return(0);
}
```

- The program calculate the ratio **a/(b-c)**
- It has the following functions: **main(), ratio() and difference().**
- main() reads the values a, b, c and calls the function ratio to calculate the value a/(b-c). the ration function cannot be evaluated if (b-c) = 0, thus the ration() function calls another function difference to test whether the difference (b-c) is zero or not; difference returns 1 if b is not equal to c; otherwise returns zero to the function ratio.
- ratio function calculates the value a/(b-c) if it received 1 and returns the result in float, or sends zero if (b-c) = zero

10.12. Recursion

When a called function in turn calls another function a process of "chaining" occurs. Recursion is a special case of this process, where a function calls itself. The program runs endlessly until terminated by force. Recursive functions can be effectively used to solve problems where solution is expressed in terms of successively applying the same solution to subsets of the problem.

Program Example 10.7: Program to demonstrate Recursion in functions

```
main()
{
printf("This is an example of recursion\n");
        main();
        getch();
```

}

10.13. Passing Arrays to functions

One-Dimensional Arrays.

It is possible to pass the values of an array to a function. To pass a one-dimensional array to a called function, it is sufficient to list the name of the array without any subscripts and the size of the array as arguments. Example

largest(a,n),

will pass an array a to the called function. The called function should be appropriately defined to receive the array,

Program Example 10.8. Program to find the largest value in an array of elements

```
#include <stdio.h>
#include<conio.h>
main()
{
float largest(float a[ ], int n);
float value[4] = {2.5, -4.75, 1.2, 4.67};
printf("%f\n", largest(value,4));
}
float largest (float a[], int n);
{
int i;
float max;
max = a[0];
for (i = 1; i< n; i++)
if(max < a[1])
max = a[1];
return (max);
getch();
}
```

Rules to Pass and Array to a Function

- The function must be called by passing only the name of the array.

- In the function definition, the formal parameter must be an array type; the size of the array does not need to be specified.
- The function prototype must show that the argument is an array.

Program Example 10.9: Program that uses a function to sort an array of integers

```
#include <stdio.h>
#include<conio.h>
void sort(int m, int x[]);
main()
{
            int i;
           int marks[5] = {40, 90, 73, 81, 35};
            printf("Marks before sorting \n");
             for(i = 0; i <5; i++)
               {
                        printf("%4d", marks[i]);
                          printf("\n");

                       sort(5, marks);
                       printf("Marks after sorting \n");
                       for ( i= 0; i <5; i++)
                        printf("%4d", marks[i]);
                       printf("\n");
             }
getch();
}
void sort(int m, int x[])
{
  int i, j, t;
  for (i = 0; i <= m-1; i++)
            for (j =1; j <= m-1;j++)
            if(x[j-1] >= x[j])
           {
```

```
            t = x[j-1];
      x[j-1] = x[j];
      x[j] = t;
}
}
```

10.14. Scope, Visibility and Lifetime of Variables

In C all variables not only have a ***data type***, but they also have a ***storage class***. The variables can be broadly categorized depending on the place of their declaration, as internal (local) and External (Global). Internal variables are declared within a function, while external are declared outside of any function

The following variable storage classes are most relevant to functions;

1. Automatic Variables: declared inside a function in which they are utilized, they are created when the function is called and destroyed automatically when the function is exited. They are private/ local/ internal to the function in which they are declared in. A variable declared inside a function without storage class specification is by default an automatic variable. One important feature of automatic variables is that their value cannot be changed accidentally by what happens in some other functions in the program.

```
main()
{
 auto int number;
 ----
  -----
}
```

2. External Variables .Variables that are **both alive and active** throughout the program. They can be accessed by any function in the program and are declared outside a function. Example

```
 int count;
main()
{
   count = 10;
   …….
```

```
   ……
}
function(1)
{
  int count = 0;
 ……
……
count = count + 1;
}

function2()
{
  int count;
……..
count = count + 2;
}
```

Note the variable count is available for use to all the functions (main(), function1() & function2())

3. Static Variables: Variables whose value persists until the end of the program. It can either be internal or external and is initialized only once when the program is compiled.
4. Register Variables: Variables kept in a machine register instead of in memory.

- The **scope** of a variable determines over what region of the program a variable is actually available for use ('Active')
- **Longevity** refers to the period during which a variable retains a given value during execution of a program ('alive'). Longevity has a direct effect on the utility of a given variable
- **Visibility** refers to the accessibility of a variable from the memory.

Scope Rules

1. The scope of global variable is the entire program life
2. The scope of a local variable begins at point of declaration and ends at the end of the block or function in which it is declared.
3. The scope of a formal function argument is its own function.
4. The lifetime (or longevity) of an auto variable declared in **main()** is the entire program execution time, although its scope is only the **main** function
5. The life of an **auto** variable declared in a function ends when the function is exited.
6. A static local variable, although its scope is limited to its functions, its lifetime extends till the end of program execution.

7. All variables have visibility in their scope, provided they are not declared again.
8. If a variable is re-declared within its scope again, it loses its visibility in the scope of the re-declared variable.

Chapter Review Questions

1. Describe any two ways of passing parameters to functions
2. Distinguish between the following;
    a) Scope and visibility
    b) Global and local
    c) Actual and formal arguments
3. Write a program that invokes a function called find() to perform the following tasks;
    a) Receive a character array and a single character
    b) Return 1 if the specified character is found in the, 0 otherwise.
4. Write a function that receives a floating point value x and returns it as a value rounded to the nearest decimal places. Example 123.4567 to be 123.46