FUNCTIONS IN C 13

13.1 OVERVIEW

The idea of modular programming is the result of inspiration from the hardware manufacturing where replicable components of different items are available. If a component of an item gets out of order, it is replaced with a newer one. Many different components from different manufacturers can be combined together to form a hardware device such as computers, cars, and washing machines. Functions are the building blocks of C programs. They encapsulate pieces of code to perform specific operations. Functions allow us to accomplish the similar kinds of tasks over and over again without being forced to keep adding the same code into the program. Functions perform tasks that may need to be repeated many times.

In programs we have seen so far, the whole program logic was contained in a single *main* function. This style of writing programs is known as *unstructured programming*. Recall chapter 8 where we have discussed the difference between unstructured and structured programming. Here we shall discuss structured programming approach. It is a modular way of writing programs. The whole program logic is divided into number of smaller modules or functions. The *main* function calls these functions where they are needed. A *function* is a self-contained piece of code with a specific purpose.

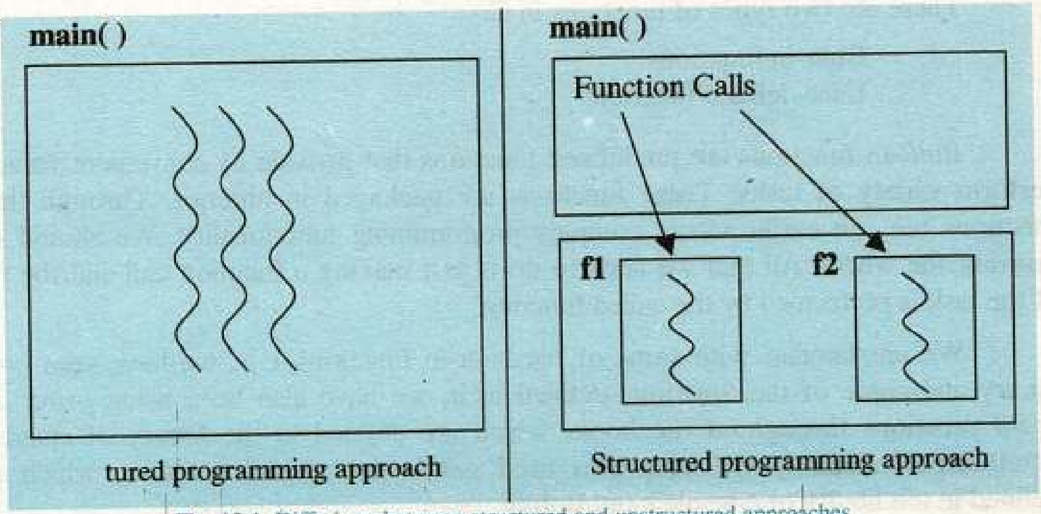

Fig. 13.1 Difference between structured and unstructured approaches

The above figure demonstrates the idea of structured and unstructured programming.

13.2 IMPORTANCE OF FUNCTIONS

A program may have repetition of a piece of code at various places. Without the ability to package a block of code into a single function, programs would end up being much larger. But the real reason to have functions is to break up a program into easily manageable chunks. The use of functions provides several benefits. Some of them are:

- They make programs significantly easier to understand and maintain. The main program can consist of a series of function calls rather than countless lines of code.
- Functions increase reusability of the code. Well written functions may be reused in multiple programs. The C standard library is an example of the reuse of functions.
- Different programmers working on one large project can divide the workload by writing different functions, hence ensuring the parallel development of the software.
- Functions can be executed as many times as necessary from different places in the program
- When an error arises, rather than examining the whole program, the infected function is debugged only.

13.3 TYPES OF FUNCTIONS

There are two types of functions in C:

(ii) Built-in functions
(iii) User-defined functions

Built-in functions are predefined functions that provide us convenient ways to perform variety of tasks. These functions are packaged in libraries. Through these functions we can easily access complex programming functionality. We should not reinvent the wheel. All that we need to do is just making a function call and the rest of the task is performed by the called function.

We are familiar with some of the built-in functions, e.g., we have seen *ctype* library and some of the functions defined in it, we have also been using *printf* and *scanf* functions throughout the book, which are defined in the library of standard input/output, similarly we have also used *getch* and *getche* functions which are defined in the library of console input/output.

To use a built-in function in C, we must have to include in wer program the header file containing its declaration. For example to user printf( ) and scanf( ), we have to include stdio.h file in wer program.

Built-in functions are not sufficient for solving every kind of problem. A programmer may need to write his/her own functions depending on the nature of problem being solved. Such functions are called *user-defined functions*.

13.4 WRITING FUNCTIONS IN C

We are familiar with the *main*( ) function, which is the mandatory part of every C program. In 8th chapter, we have introduced the structure of the *main* function. Every function in C has almost the same basic structure. A function in C consists of a *function header* which identifies the function followed by the body of the function between curly braces containing the executable code for the function. Every function in C is written according to the following general form:

```
returen_type FunctionName (parameter_list)
{
    Executable Statement(s)

    return expression;
}
```

13.4.1 Function Header

The first line of function definition is called the function header i.e.

```
return_type FunctionName (parameter_list)
```

It consists of three parts:

- The type of the return value
- The name of the function
- The parameters of the function enclosed in parentheses

The `return_type` can be any valid data type. If the function does not return a value, the return type is specified by the keyword **void**. A function that has no parameter specifies the keyword **void** as its parameter list. Hence, a function that has no parameter and does not return any value to the calling function will have the header:

```
void FunctionName (void)
```

However the keyword *void* is optional. The above function header for a function that has no argument can be re-written as follows:

```
void FunctionName()
```

13.4.2 The Function Body

Variables declaration and the program logic are implemented in the function body. Function body makes use of the arguments passed to the function. It is enclosed in curly braces. A function can be called in the body of another function.

13.4.3 The `return` Statement

The *return* statement is used to specify the value retuned by a function. The general form of *return* statement is:

```
return [expression];
```

When the *return* statement is executed, *expression* is evaluated and returned as the value of the function. Execution of the function stops when the *return* statement is executed, even if there are other statements still remaining in the function body. If the type of the return value has been specified as *void* in the function header then there is no need to use a return statement.

13.5 FUNCTION PROTOTYPE

The compiler must know functions used in the program. That's why we include corresponding header files in the source program before using built-in functions such as stdio.h and conio.h etc. A header file contains the prototypes of the functions provided by the library. The compiler actually needs enough information to be able to identify the function that we are using. A *function prototype* is a statement that provides the basic information that the compiler needs to check and use a function correctly. It specifies the parameters to be passed to the function, the function name, and the type of the return value. The general form of the function prototype is as follows:

```
return_type FunctionName (parameter_list);
```

We might be surprising at the above statement. It looks like the function header; yes it is, but with a semicolon at the end.

The prototype for a function which is called from another function must appear before the function call statement. Functions prototypes are usually placed at the beginning of the source file just before the function header of the *main* function.

13.6 CALLING A FUNCTION

Function call is a mechanism that is used to invoke a function to perform a specific task. A function call can be invoked at any point in the program. In C the function name, the arguments required and the statement terminator (;) are specified to invoke a function call.

When function call statement is executed, it transfers control to the function that is called. The memory is allocated to variables declared in the function and then the statements in the function body are executed. After the last statement in the function is executed, control returns to the calling function.

13.7 LOCAL VARIABLES AND THEIR SCOPE

When the program executes, all variables are created in memory for a limited time period. They come into existence from the place where they are declared and then they are destroyed. The duration in which a variable exists in memory is called *lifetime of the variable*.

Note: Operating system manages the allocation and de-allocation of memory for all variables in wer programs. So, by *destroying a variable* we mean returning the memory allocated to a variable back to the operating system for other programs.

The *scope of a variable* refers to the region of a program in which it is accessible. The name of a variable is only valid within its scope. So a variable can not be referred outside its scope. Any attempt to do so will cause a compiler error.

All variables that we have declared so far have been declared within a block – that is, within the extent of a pair of curly braces. These are called *local variables* and have *local scope*. The scope of a local variable is from the point in the program where it is declared until the end of the block containing its declaration.

Example 1

```
#include <stdio.h>
void main()
{
          int nCount = 0;

          if (nCount = 0)
          {
                    int chk;

                    chk = 10;
          }
          printf("%d", chk);
}
```

We have used two variables in this program; these are *nCount*, and *chk*. Both of these are local variables. But, they have different scope. The scope of the variable *nCount* is the block of *main*( ) function i.e., from its point of declaration to the end of the *main*( ) function. Whereas the scope of the variable *chk* is the block of *if* statement i.e., from its point of declaration until the end of the block of *if* statement.

These variables can only be referenced within their respective scopes. Any reference made to them outside of their scopes would be illegal, thus the program causes the following *compiler error*.

'chk' : undeclared identifier

This is because in the last *printf( )* statement of the program, the variable chk is referenced outside of *if* block i.e., out of its scope, which is illegal. The lifetime of local variables is the duration in which the program control remains in the block in which they are declared. As soon as the control moves outside of their scope, these variables are destroyed.

13.8 GLOBAL VARIABLES AND THEIR SCOPES

The variables which are declared outside all blocks i.e. outside the main( ) and all other functions are called *global variables* and have *global scope*. They are accessible from the point where they are declared until end of the file containing them. It means they are visible throughout all the functions in the file, following their point of declaration. The lifetime of global variable is until the termination of the program. They exist in memory from the start to the end of the program.

Note: It is very important to understand that every time the block of statements containing a declaration for a local variable is executed, the variable is created anew, and if we specify an initial value for the local variable, it will be reinitialized each time it is created.

Example:

```
#include <stdio.h>
void main()
{
        int nCount = 1;

        clrscr();
        While(nCount <= 10)
        {
                int chk = 10;
                printf("%d\t", chk);

                chk = chk + 1;
                nCount++;
        }
}
```

In this program, each repetition of the loop prints the same value of the variable *chk*. The addition to the value of *chk* will have no effect, because at the end of execution of the body of the loop, the control moves outside the loop body (which is also the scope of *chk* variable) and returns to the *while* statement; this causes the the *chk* variable to be destroyed in each repetition. The *chk* variable is again created in the next repetition and gets destroyed at the end of the repetition. This process continues until the loop condition is true.

Output:

```
10    10    10    10    10    10    10    10    10    10
```

Example 2

```
#include <stdio.h>
void Counter(void);
int nCount = 0;

void main()
{
        for (int n = 0; n <= 10; n+=2)
              Counter();

        Printf("nCount = %d", nCount);
  }
  void Counter(void)
  {
        nCount++;
  }
```

This is a simple program which demonstrates the use of global variable. Here, we have declared a global variable i.e., *nCount* outside the *main* and the *Counter* functions. This is not contained in any block. The global variable *nCount*, the function *main*, and the function *Counter* all are defined in the same file. Because, the variable *nCount* is declared on top of the two functions, therefore it is visible within them. The function *Counter*, increments the value of *nCount* by one each time it is called. The main() executes a loop six times and call the function *Counter* to increment the value of nCount. The value of the variable nCount is printed as the final *output* of the program i.e.,

Output:

```
nCount = 6
```

Note: The point to be noted here is that the variable **nCount** is declared outside the functions main() and Counter(), but they manipulate it as if it was declared within them. The nCount is created in memory before the start of execution of the main() and exists until the execution of the program ends.

13.9 FUNCTIONS WITHOUT ARGUMENTS

The simplest type of function is one that returns no value and no arguments are passed to them. The return type of such functions is void and the Parameter_List may either be empty or containing the keyword void. Lets consider the following example.

Example 3 **Write a function named Print_Asterisks that will print asterisks (*) according to the pattern shown in the fig. 13.2. and invoke a function call from the function main to print the asterisks.**

```
*******
******
*****
****
***
**
*
```

Fig. 13.2 asterisks pattern

```
#include <stdio.h>

void Print_Asterisks(void); //function
                            // prototype
void main(void)
{
    // Function call
    Print_Asterisks();
}
void Print_Asterisks(void) //function header
{
    int inner;

    for(int outer=7; outer>=1; outer--)
    {
        inner = 1;
        while( inner <= outer)
        {
            printf("*");
            inner++;
        }
        printf("\n");
    }
}
```

We have discussed this program in the previous chapter, but here we have followed a different approach. The next line to the #include directive is the *prototype* for the function Print_Asterisks( ). It tells the compiler about the function, its return_type and number of parameters (void) in this case. Our *main* function consists of just one line of code i.e.,

```
Print_Asterisks();
```

It represents a *function call* to the function Print_Asterisks( ). We can think of a function as a worker who takes necessary steps to accomplish the task assigned to him.

Similarly, the function `Print_Asterisks()`is capable of printing asterisks in a specific order. When the function call statement is executed, the control is immediately transferred to the Print_Asterisks function. Memory is allocated to the variables *inner* and *outer*. Then comes the *for* and *while* loops, which print asterisks. When the task is completed, the control is transferred to the function *main* from the function Print_Asterisks, and the memory allocated to the variables *inner* and *outer* is returned to the operating system again. Then, the control is transferred to the next statement to the function call statement in the calling function i.e., main(). As there is no statement in the *main* function other than the function call, so the program will terminate.

13.10 FUNCTIONS THAT RETURN A VALUE AND ACCEPT ARGUMENTS

So far, we have discussed simple functions that return no value to the calling function. However, we may need a function that could return a value and arguments could be passed to it. In previous chapters, we have seen a number of such built-in functions e.g., sqrt(), toupper(), tolower() etc. Here, we shall learn to write these types of functions in C. Let's consider the general form of function header:

```
return_type FunctionName(parameter_list)
```

The *return_type* specifies the data type of the value that the function returns. *Parameter_list* is a coma separated list which specifies the data type and the name of each parameter in the list.

Example 4

```
#include <stdio.h>
int Add(int n1, int n2);
void main()
{
      int a, b;
      int sum;
      clrscr();        //clears   the   previous   output
                       from //the screen
      printf("Enter values for 'a' and 'b' >");
      scanf("%d %d", &a, &b);

      sum = Add(a, b);
```

```
        printf("%d + %d = %d", a, b, sum);
    }
    int Add(int n1, int n2)
    {
        return n1 + n2;
    }
```

Suppose the user enters 12 and 15 for *a* and *b* respectively then the *output* of the of the program will be:

```
12 + 15 = 27
```

The 7th line of code in the *main* function is a *function call* to the *Add( )* function. The Add( ) requires two parameters of type *int* to be passed to it. In the function call, we have passed two variables i.e., *a* and *b* of type *int* to the function. These arguments (i.e., variables *a* and *b*) are called *actual arguments* or *actual parameters* of the function. These are local variables and their scope is the body of *main* function. Whereas the parameters specified in the function header (i.e., *n1* and *n2*) are called *formal arguments* or *formal parameters* of the function and their scope is the body of *Add* function. These are also called dummy arguments.

When parameters are passed to a function, the value of actual parameters is copied in the formal parameters of the functions. The function uses its formal parameters for processing data passed to it. Any change made to the value of formal parameters does not affect the value of actual parameters. Here, the values of *a* and *b* are copied in *n1* and *n2* respectively. The function *Add* returns the sum of the two values to the *main* function which is then assigned to the variable *sum*.

Example 5

```
#include <stdio.h>
float Area_of_Triangle(int base, int altitude);
void main()
{
    int a, b;
    float area;

    printf("Enter value for altitude: ");
    scanf("%d", &a);
    printf("Enter value for base: ");
    scanf("%d", &b);
    area = Area_of_Triangle(a, b);
    printf("Area of triangle is %.2f", area);
}

float Area_of_Triangle(int base, int altitude)
{
    return (0.5*base*altitude);
}
```

Output

Suppose the user enters 25 and 45 for altitude and base respectively, then the output of the program will be:

```
Area of triangle is 562.50
```

Exercise 13c

1. Fill in the blanks:
   - (i) A __________ is a self contained piece of code.
   - (ii) Pre-defined functions are packaged in _____________.
   - (iii) A _____________ provide basic information about the function to the compiler.
   - (iv) The duration for which a variable exists in memory is called its _________.
   - (v) __________ of a variable refers to the region of the program where it can be referenced.
   - (vi) __________ variables are declared outside all blocks.
   - (vii) A function can not return more than ______ value(s) through **return** statement.
   - (viii) The parameters specified in the function header are called _________ parameter.
   - (ix) The parameters passed to a function in the function call are called ________ parameters.
   - (x) Functions help to achieve __________ programming.

2. Choose the correct option:
   - (i) Function prototypes for built-in functions are specified in:
     - a) source files
     - b) header files
     - c) object files
     - d) image files
   - (ii) Global variables are created in:
     - a) RAM
     - b) ROM
     - c) hard disk
     - d) cache
   - (iii) Which of the following is true about a function call?
     - a) Stops the execution of the program
     - b) Transfers control to the called function
     - c) Transfers control to the *main* function
     - d) Resumes the execution of the program
   - (iv) Which of the following looks for the prototypes of functions used in a program?
     - a) linker
     - b) loader
     - c) compiler
     - d) parser

(v) Memory is allocated to a local variable at the time of its:

a)	declaration	b)	destruction
c)	definition	d)	first reference

(vi) The name of actual and formal parameters:

a)	may or may not be same	b)	must be same
c)	must be different	d)	must be in lowercase

(vii) Formal arguments are also called:

a)	actual arguments	b)	dummy arguments
c)	original arguments	d)	referenced arguments

(viii) printf() is a:

a)	built-in function	b)	user-defined function
c)	local function	d)	keyword

(ix) A built-in function:

a)	can not be redefined	b)	can be redefined
c)	can not return a value	d)	should be redefined

(x) In a C program, two functions can have:

a) same name
b) same parameters
c) same name and same parameters
d) same name but different parameters

3. Write T for true and F for false statement.

(i) In C, arguments can be passed to a function only by value.
(ii) There can be multiple *main* functions in a C program.
(iii) A function can be called anywhere in the program.
(iv) In C, every function must return a value.
(v) A user-defined function can not be called in another user-defined function.
(vi) A function can be called only once in a program.
(vii) Scope of a local variable is the block in which it is defined.
(viii) Global variables exist in memory till the execution of the program.
(ix) An unstructured program is more difficult to debug than a structured program.
(x) Function body is an optional part of the function.

4. What is a function? How many types of functions are used in C? Discuss the difference between them.

5. Differentiate the following:
   (i) Function Definition and Function Declaration
   (ii) Global and Local variables
   (iii) Scope and Lifetime of a variable
   (iv) Function prototype and Function header
   (v) Formal parameters and Actual parameters of a function

6. How is a function call made in a C program? Discuss briefly.

7. Answer the following:
   (i) What is the purpose of a function argument?
   (ii) How many (maximum) values can a function return using **return** statement?
   (iii) When is a function executed, and where should a function prototype and function definition appear in a source program?
   (iv) Write three advantages of functions.

8. Write a program that call two functions Draw_Horizontal and Draw_Vertical to construct a rectangle. Also write functions Draw_Horizontal to draw two parallel horizontal lines, and the function Draw_Vertical to draw two parallel vertical lines.

9. Write a program that prompts the user for the Cartesian coordinates of two points (x_1, y_1) and (x_2, y_2) and displays the distance between them. To compute the distance, write a function named Distance( ) with four input parameters. The function Distance( ) uses the following distance formula to compute the distance and return the result to the calling function:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

10. Write a program that prompts the user to enter a number and then reverse it. Write a function Reverse to reverse the number. For example, if the user enters 2765, the function should reverse it so that it becomes 5672. The function should accept the number as an input parameter and return the reversed number.

11. Write a function named *Draw_Asterisks* that will print asterisks (*) according to the pattern shown in the following and make a function call from the function *main* to print the asterisks pattern.

```
*********
*******
*****
***
*
```

12. Write a function Is_Prime that has an input parameter i.e num, and returns a value of 1 if *num* is prime, otherwise returns a value of 0.

13. Write a complete C program that inputs two integers and then prompts the user to enter his/her choice. If the user enters 1 the numbers are added, for the choice of 2 the numbers are divided, for the choice of 3 the numbers are multiplied, for the choice of 4 the numbers are divide (divide the larger number by the smaller number, if the denominator is zero display an error message), and for the choice of 5 the program should exit. Write four functions Add(), Subtract(), Multiply() and Divide() to complete the task.

14. Write a program that prompts the user to enter a number and calls a function Factorial() to compute its factorial. Write the function Factorial() that has one input parameter and returns the factorial of the number passed to it.

15. Write a function GCD that has two input parameters and returns the greatest common divisor of the two numbers passed to it. Write a complete C program that inputs two numbers and call the function GCD to compute the greatest common divisor of the numbers entered.