Unit # 5

FUNCTIONS

Students Learning Outcomes

After completing this unit students will be able to

- Explain the concept and types of functions
- Explain the advantages of using functions
- Explain the signature of function (Name, Arguments, Return type)
- Explain the following terms related to functions
  - Definition of a function
  - Use of a function

Unit Introduction

A good problem solving approach is to divide the problem into multiple smaller parts or sub-problems. Solution of the whole problem thus consists of solving the sub-problems one by one, and then integrating all the solutions. In this way, it becomes easier for us to focus on a single smaller problem at a time, instead of thinking about the whole problem all the time. This problem solving approach is called *divide and conquer*. C programming language provides us with *functions* that allow us to solve a programming problem using the divide and conquer approach. In this chapter, we will learn the concept of functions, their advantages, and how to work with them.

5.1 Functions

A function is a block of statements which performs a particular task, e.g. *printf* is a function that is used to display anything on computer screen, *scanf* is another function that is used to take input from the user. Each program has a *main* function which performs the tasks programmed by the user. Similarly, we can write other functions and use them multiple times.

5.1.1 Types of Functions

There are basically two types of functions:

1) Built-inFunctions
2) User Defined Functions

Built-in Functions

The functions which are available in C Standard Library are called built-in functions. These functions perform commonly used mathematical calculations, string operations, input/output operations etc. For example, *printf* and *scanf* are built-in functions.

User Defined Functions

The functions which are defined by a programmer are called user-defined functions. In this chapter we will learn how to write user defined functions.

5.1.2 Advantages of Functions

Functions provide us several advantages.

1) **Reusability:** Functions provide reusability of code. It means that whenever we need to use the functionality provided by the function, we just call the function. We do not need to write the same set of statements again and again.

2) **Separation of tasks:** Functions allow us to separate the code of one task from the code of other tasks. If we have a problem in one function, then we do not need to check the whole program for removing the problem. We just need to focus at one single function.

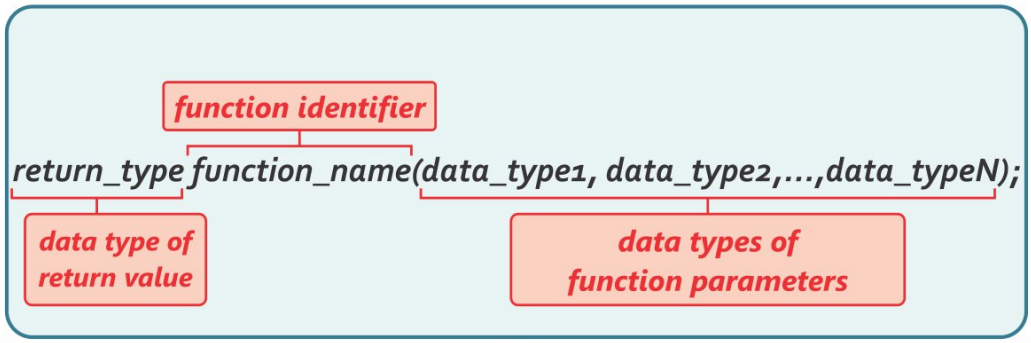
3) **Handling the complexity of the problem:** If we write the whole program as a single procedure, management of the program becomes difficult. Functions divide the program into smaller units, and thus reduce the complexity of the problem.

4) **Readability:** Dividing the program into multiple functions, improves the readability of the program.

5.1.3 Signature of a Function

A function is a block of statements that gets some inputs and provides some output. Inputs of a function are called **parameters** of the function, and output of the function is called its **return value**. A function can have multiple parameters, but it cannot return more than one values.

Function signature is used to define the inputs and output of a function. The general structure of a *function signature* is as follows:



Example Function Signatures:

Table 5.1 shows the descriptions of some functions and their signatures.

Function Description	Function Signature
A function that takes an integer as input and returns its square.	`int square (int);`
A function that takes length and width of a rectangle as input and returns the perimeter of the rectangle.	`float perimeter (float, float);`
A function that takes three integers as input and returns the largest value among them.	`int largest (int, int, int);`
A function that takes radius of a circle as input and returns the area of circle.	`float area (float);`
A function that takes a character as input and returns 1, if the character is a vowel, otherwise returns 0.	`int isVowel (char);`

Table 5.1: Some functions and their Signatures

5.1.4 Defining a Function

The function signature does not describe how the function performs the task assigned to it. Function definition does that. A function definition has the following general structure.

```
return_type function_name (data_type var1, data_type var2,.., data_type varN)
{
        Body of the function
}
```

Body of the function is the set of statements which are executed in the function to perform the specified task. Just after the function's signature, the set of statements enclosed inside { } form the body of the function.

Following example defines a function *showPangram()* that does not take any input and does not return anything, but displays *A quick brown fox jumps over the lazy dog.* on computer screen.

</> EXAMPLE CODE 5.1

```
void showPangram()
{
      printf("\nA quick brown fox jumps over the lazy
      dog.\n");
}
```

function name

As the above function does not return anything thus return type of the function is *void*.

Let's take another example of a function that takes as input two integers and returns the sum of both integers.

</> EXAMPLE CODE 5.2

```
int add(int x, int y)
{
      int result;
      result = x + y;
      return result;
}
```

parameters of function

function name

return type

Inside the function, *return* is a keyword that is used to return a value to the calling function.

Important Note:

A function cannot return more than one values. e.g the following statement results in a compiler error.

```
return (4, 5);
```

Important Note:

There may be multiple *return* statements in a function but as soon as the first *return* statement is executed, the function call returns and further statements in the body of function are not executed.

Using a Function

We need to call a function, so that it performs the programmed task. Following is the general structure used to make a function call.

```
function_name(value1, value2,…, valueN);
```

For example, let's observe the following program.

</> EXAMPLE CODE 5.3

```
void main()
{
        printf("Hello from main()");
        showPangram();   <-- function call
        printf("Welcome back to main()");
}
```

Output:

```
Hello from main()
A quick brown fox jumps over the lazy dog.
Welcome back to main()
```


We can see that the program starts its execution from *main()* function. When it encounters a function call (inside the rectangle), it transfers the control to called function. After executing the statements of called function, the control is transferred back to the calling function, i.e. *main()* in the above example.

The following program inputs two numbers and displays their sum.

The statement inside the rectangle in the following code includes a call to the *add* function defined in previous section.

EXAMPLE CODE 5.4

```
void main ()
{
        int n1, n2, sum;
        scanf ("%d%d", &n1, &n2);
        sum = add (n1, n2);
        printf ("Sum is %d", sum);
}
```

function name

function call

function arguments

- In the function call *n1* and *n2* are arguments to the function *add()* discussed in **Example 5.2**.
- Variable *sum* is declared to store the result returned from the function *add()*.
- The variables passed as arguments are not altered by the function. The function makes a copy of the variables and all the modifications are made to that copy only.
- In the above example when *n1* and *n2* are passed, the function makes copies of these variables. The variable *x* is the copy of *n1* and the variable *y* is the copy of *n2*.

Important Note:

The values passed to the function are called ***arguments***, whereas variables in the function definition that receive these values are called ***parameters*** of the function.

In the above example, values of variables *n1* and *n2* are arguments to the function *add()*, whereas the variables x and y inside function *add()* are parameters of the function.

Important Note:

It is not necessary to pass the variables with same names to the function as the names of the parameters. However, we can also use same names. Here another important point is that even if we use same names, still the variables used in the function are a copy of the original variables. This is illustrated here through following example:

</> EXAMPLE CODE 5.5

```
#include<stdio.h>
void fun(int x, int y)
{
      x = 20;
      y = 10;
      printf("Values of x and y in fun(): %d %d", x, y);
}
void main()
{
      int x = 10, y = 20;
      fun(x, y);
      printf("Values of x and y in main(): %d %d", x, y);
}
```

Output:

```
Values of x and y in fun(): 20 10
Values of x and y in main(): 10 20
```

Important Note:

Following points must be kept in mind for the arrangement of functions in a program.

1- If the definition of called function appears before the definition of calling function, then function signature is not required.

2- If the definition of called function appears after the definition of calling function, then function signature of called function must be written before the definition of calling function.

Both the following code structures are valid.

a)
```
int add(int, int);
void main()
{
   printf("%d "add(4, 5));
}
int add(int a, int b)
{
   return a + b;
}
```

b)
```
int add(int a, int b)
{
   return a + b;
}
void main()
{
   printf("%d "add(4, 5)
}
```


PROGRAMMING TIME 5.1

Problem:

Write a function isPrime() that takes a number as input and returns 1 if the input number is prime, otherwise returns 0. Use this function in main().

Program:

```
#include <stdio.h>
int prime (int n)
{
      for (int i = 2; i < n; i++)
            if(n % i == 0)
                  return 0;
      return 1;
}
```

Continued

```
void main()
{
      int x;
      printf ("Please enter a number: ");
      scanf ("%d", &x);
      if(prime(x))
            printf ("%d is a Prime Number",x);
      else
            printf ("%d is not a Prime Number",x);
}
```

PROGRAMMING TIME 5.2

Problem:

Write a function which takes a positive number as input and returns the sum of numbers from 0 to that number.

Program:

```
int digitsSum(int n)
{
      int sum = 0;
      for(int i = 0; i <= n; i++)
      {
            sum = sum + i;
      }
      return sum;
}
void main()
{
      int number;
      printf("Please enter a positive number: ");
      scanf("%d", &number);
```

Continued

```
    if(number >= 0)
    {
        int sum = digitsSum(number);
        printf("The sum of numbers upto given number is
%d",sum);
    }
    else
        printf("You entered a negative number.");
}
```

SUMMARY

- A ***function*** is a block of statements that performs a particular task.
- The functions which are available in C Standard Library are called ***built-in functions***.
- The functions which are defined by a programmer are called ***user-defined functions***.
- Some advantages of using functions are: reusability of code, separation of tasks, reduction in the complexity of problem, and readability of code.
- **Function signature** describes the name, inputs and output of the function.
- We can define a function as follows

```
return_type name (Parameters)
{
   Body of the Function
}
```

- The ***return type*** of the function is the data type of the value returned by function.
- The ***name*** of the function should be related to its task.
- ***Parameters*** are variables of different data types, that are used to receive the values passed to the function as input.
- ***Body*** of the function is the set of statements which are executed in the function to fulfil the specified task.
- ***Calling a function*** means to transfer the control to that particular function.
- During the function call, the values passed to the function are called ***arguments***.
- We can call a user-defined function from another user defined function, same as we call other functions in main function.

Exercise

Q1 Multiple Choice Questions

1) Functions could be built-in or _____.

a) admin defined **b)** server defined **c)** user defined **d)** Both a and c

2) The functions which are available in C Standard Library are called _____.

a) user-defined **b)** built-in **c)** recursive **d)** repetitive

3) The values passed to a function are called _____.

a) bodies **b)** return types **c)** arrays **d)** arguments

4) char cd() { return 'a'}. In this function "char" is _____.

a) body **b)** return type **c)** array **d)** arguments

5) The advantages of using functions are _____

a) readability **b)** reusability **c)** easy debugging **d)** all

6) If there are three return statements in the function body, _____ of them will be executed.

a) one **b)** two **c)** three **d)** first and last

7) Readability helps to _____ the code.

a) understand **b)** modify **c)** debug **d)** all

8) _____ means to transfer the control to another function.

a) calling **b)** defining **c)** re-writing **d)** including

Q2 Define the following.

1) Functions **2)** Built-in functions **3)** Functions Parameters

4) Reusability **5)** Calling a function

Q3 Briefly answer the following questions.

1) What is the difference between arguments and parameters? Give an example.

2) Enlist the parts of a function definition.

3) Is it necessary to use compatible data types in function definition and function call? Justify your answer with an example.

4) Describe the advantages of using functions.

5) What do you know about the *return* keyword?

Q4 Identify the errors in the following code segments.

a)
```
void sum (int a, int b)
{
      return a + b;
}
```

b)
```
void message ();
{
      printf ("Hope you are fine :)");
      return 23;
}
```

c)
```
int max (int a; int b)
{
      if (a > b)
            return a;
      return b;
}
```

d)
```
int product (int n1, int n2)
      return n1*n2;
```

e)
```
int totalDigits (int x)
{
      int count = 0;
      for (int i = x; i >= 1, i = i/10)
            count++;
      return count
};
```

Q5 Write down output of the following code segments.

a)
```
int xyz (int n)
{
    return n + n;
}
int main()
{
    int p = xyx(5);
    p = xyz(p);
    printf ("%d ",p);
}
```

b)
```
void abc (int a, int b, int c)
{
    int sum = a + b + c;
}
int main()
{
    int x = 4, y = 7, z = 23, sum1 = 0;
    abc (x, y, z);
    printf ("%d %d %d" x, y ,z);
}
```

c)
```
int aa (int x)
{
    int p = x / 10;
    x++;
    p = p + (p * x);
    return p;
}
int main()
{
    printf ("We got %d ", aa(aa(23)));
}
```

d)

```
float f3(int n1, int n2)
{
      n1 = n1 + n2;
      n2 = n2 - n1;
      return 0;
}
int main()
{
      printf ("%f\n", f3(3, 2));
      printf ("%f\n", f3(10, 6));
}
```