UNIT 6: FUNCTION AND STORAGE CLASS

UNIT STRUCTURE

6.0 OBJECTIVES

After completion of this unit you will be able to:

- Define and declare a function
- Calling a function and transferring data through parameter passing
- Write programs involving recursive functions, call by value and call by reference
- Concept of various storage class used in C programming

6.1 INTRODUCTION TO FUNCTIONS

C functions are basic building blocks in a program. All C programs are written using functions to improve re-usability, understandability and to keep track on them. A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

Any function can be called from any other function. Even **main( )** can be called from other functions.

For example,

```
main( )
{
        message( ) ;
}

message( )
{
        printf ( "\nCan't imagine life without C" ) ;
        main( ) ;
}
```

A function can be called any number of times. For example,

```
main( )
{
     message( ) ;
     message( ) ;
}
message( )
{
     printf ( "\nHello World!!" ) ;
}
```

A function can also be referred as a method or a sub-routine or a procedure, etc.

USES OF C FUNCTIONS:

- C functions are used to avoid rewriting same logic/code again and again in a program.
- There is no limit in calling C functions to make use of same functionality wherever required.
- We can call functions any number of times in a program and from any place in a program.
- A large C program can easily be tracked when it is divided into functions.
- The core concept of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality and to improve understandability of very large C programs.

6.2 Types of functions in C programming

Depending on whether a function is defined by the user or already included in C compilers, there are two types of functions in C programming

There are two types of functions in C programming:

- Standard library functions
- User defined functions

6.2.1 Standard library functions

The standard library functions are built-in functions in C programming to handle tasks such as mathematical computations, I/O processing, string handling etc.

These functions are defined in the header file. When you include the header file, these functions are available for use. For example:

The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in "stdio.h" header file.

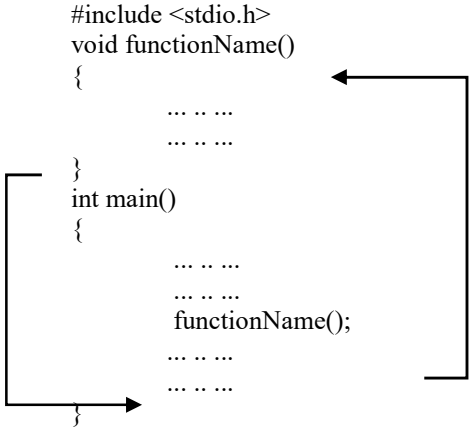
There are other numerous library functions defined under "stdio.h", such as scanf(), fprintf(), getchar() etc. Once you include "stdio.h" in your program, all these functions are available for use.

6.2.2 User-defined functions

As mentioned earlier, C allow programmers to define functions. Such functions created by the user are called user-defined functions.

Depending upon the complexity and requirement of the program, you can create as many user-defined functions as you want.

How user-defined function works?



The execution of a C program begins from the main() function.
When the compiler encounters *functionName()*; inside the main function, control of the program jumps to *void functionName().*And, the compiler starts executing the codes inside the user-defined function.
The control of the program jumps to statement next to functionName();once all the codes inside the function definition are executed.

6.3 Function Declaration, Function Call And Function Definition:

There are 3 aspects in each C function. They are,

- Function declaration or prototype – This informs compiler about the function name, function parameters and return value's data type.
- Function call – This calls the actual function
- Function definition – This contains all the statements to be executed.

Example 6.3 : User-defined function to add two integers

```
#include <stdio.h>
int addNumbers(int a, int b);         // function prototype
int main()
{
   int n1,n2,sum;
   printf("Enters two numbers: ");
   scanf("%d %d",&n1,&n2);
   sum = addNumbers(n1, n2);         // function call
   printf("sum = %d",sum);
```

```
    return 0;
}

int addNumbers(int a,int b)        // function definition
{
    int result;
    result = a+b;
    return result;                 // return statement
}
```

5.3.1 Function prototype

A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body.
A function prototype gives information to the compiler that the function may later be used in the program.

Syntax of function prototype

returnType functionName(type1 argument1, type2 argument2,...);
In the above example, int addNumbers(int a, int b); is the function prototype which provides following information to the compiler:

1. name of the function is *addNumbers()*
2. return type of the function is *int*
3. two arguments of *type int* are passed to the function

Function Prototype requires that the function declaration must include the return type of function as well as the type and number of arguments or parameters passed. The variable names of arguments need not to be declared in prototype. The major reason to use this concept is that they enable the compiler to check if there is any mismatch between function declaration and function call. The function prototype is not needed if the user-defined function is defined before the main() function.

6.3.2 Calling a function

Control of the program is transferred to the user-defined function by calling it.

Syntax of function call

```
functionName(argument1, argument2, ...);
```

In the above example, function call is made using addNumbers(n1,n2); statement inside the main().

6.3.3 Function definition

Function definition contains the block of code to perform a specific task i.e. in this case, adding two numbers and returning it.

Syntax of function definition

```
returnType functionName(type1 argument1, type2 argument2, ...)
{
        //body of the function
}
```

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

6.4 Passing arguments to a function

In programming, argument refers to the variable passed to the function. In the example 4.6, two variables *n1* and *n2* are passed during function call.
The parameters *a* and *b* accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.
The type of arguments passed to a function and the formal parameters must match, otherwise the compiler throws error. If *n1* is of *char* type, *a* also should be of *char* type. If *n2* is of *float* type, variable *b* also should be of *float* type. Instead of using different variable names ***a*** and ***b***, we could have used the same variable names ***n1*** and ***n2.*** But the compiler would still treat them as different variables since they are in different functions.
A function can also be called without passing an argument.
If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.
Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
While calling a function, there are two ways in which arguments can be passed to a function –

6.4.1 Call By Value

This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
By default, C programming uses *call by value* to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

Example 6.4.1

```
/* function definition to swap the values */
void swap(int x, int y) {
   int temp;
   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put temp into y */
     return;
}

#include <stdio.h>
/* function declaration */
void swap(int x, int y);
 int main () {
   /* local variable definition */
   int a = 100;
   int b = 200;
   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );
   /* calling a function to swap the values */
   swap(a, b);
   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );
   return 0;
}
```

Output:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

In call by value, **original value is not modified**. In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().Different Memory is allocated for both actual and formal parameters. Because, value of actual parameter is copied to formal parameter.

By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

6.4.2 Call By Reference

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.
To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to, by their arguments.

Example 6.4.2

```
/* function definition to swap the values */
void swap(int *x, int *y) {

   int temp;
```

```
   temp = *x;    /* save the value at address x */
   *x = *y;      /* put y into x */
   *y = temp;    /* put temp into y */
   return;
}
```

Let us now call the function **swap()** by passing values by reference as in the following example −

```
#include <stdio.h>

/* function declaration */
void swap(int *x, int *y);

int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values.
      * &a indicates pointer to a ie. address of variable a and
      * &b indicates pointer to b ie. address of variable b.
   */
   swap(&a, &b);

   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );

   return 0;
}
```

Output:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

It shows that the change has reflected outside the function as well, unlike call by value where the changes do not reflect outside the function. In call by reference method, the address of the variable is passed to the function as parameter.
The value of the actual parameter can be modified by formal parameter.
Same memory is used for both actual and formal parameters since only address is used by both parameters.

6.5 Scope Rule of Functions

Look at the following program

```
main( )
{
        int i = 20 ;
        display ( i ) ;
}
display ( int j )
{
```

```
        int k = 35 ;
        printf ( "\n%d", j ) ;
        printf ( "\n%d", k ) ;
}
```

In this program is it necessary to pass the value of the variable **i** to the function **display( )**? Will it not become automatically available to the function **display( )**? No. Because by default the scope of a variable is local to the function in which it is defined. The presence of **i** is known only to the function **main( )** and not to any other function. Similarly, the variable **k** is local to the function **display( )** and hence it is not available to **main( )**. That is why to make the value of **i** available to **display( )** we have to explicitly pass it to **display( )**. Likewise, if we want **k** to be available to **main( )** we will have to return it to **main( )** using the **return** statement. In general we can say that the scope of a variable is local to the function in which it is defined.

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language −

- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.
- In the definition of function parameters which are called **formal** parameters.

Let us understand what are **local** and **global** variables, and **formal** parameters.

6.5.1 Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

Example 6.5.1

```
#include <stdio.h>
 int main () {
  /* local variable declaration */
  int a, b;
  int c;
   /* actual initialization */
  a = 10;
  b = 20;
  c = a + b;
   printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
   return 0;
}
```

6.5.2 Global Variables

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program show how global variables are used in a program.

Example 6.5.2

```
#include <stdio.h>
 /* global variable declaration */
int g;
 int main () {
  /* local variable declaration */
  int a, b;
  /* actual initialization */
  a = 10;
  b = 20;
  g = a + b;
  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
  return 0;
}
```

A program can have same name for local and global variables but the value of local variable inside a function will take preference. Here is an example −

```
#include <stdio.h>
 /* global variable declaration */
```

```
int g = 20;
 int main ()
{
  /* local variable declaration */
            int g = 10;
            printf ("value of g = %d\n",  g);
            return 0;
}
```

When the above code is compiled and executed, it produces the following result −
value of g = 10

6.5.3 Formal Parameters
Formal parameters, are treated as local variables with-in a function and they take precedence over global variables. Following is an example –
Example 6.5.3

```
#include <stdio.h>
 /* global variable declaration */
int a = 20;
 int main () {
  /* local variable declaration in main function */
  int a = 10;
  int b = 20;
  int c = 0;
  printf ("value of a in main() = %d\n",  a);
  c = sum( a, b);
  printf ("value of c in main() = %d\n",  c);
  return 0;
}

/* function to add two integers */
int sum(int a, int b) {

   printf ("value of a in sum() = %d\n",  a);
   printf ("value of b in sum() = %d\n",  b);

   return a + b;
}
```

When the above code is compiled and executed, it produces the following result −
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30

6.5.4 Initializing Local and Global Variables
When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows –

Data Type	**Initial Default Value**
int	0
char	'\0'
float	0
double	0
pointer	NULL

It is a good programming practice to initialize variables properly, otherwise your program may produce unexpected results, because uninitialized variables will take some garbage value already available at their memory location.

6.6 Return Statement

The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after return statement.
In the above example, the value of variable *result* is returned to the variable *sum* in the main() function.

Syntax of return statement

return (expression);

For example,

```
return a;
return (a+b);
```

The type of value returned from the function and the return type specified in function prototype and function definition must match.
You can pass any number of arguments to a function but can return only one value at a time.
If a function does not return anything, ***void*** specifier is used in the function declaration.
For example:

```
void square(int no)
{
        int sq;
        sq=no*no;
        printf("square is %d", sq);
}
```

All the function's return type is by default ***int,*** i.e., a function returns an integer value, if no type specifier is used in the function declaration.
A function can have many *return* statements. This thing happens when some condition based returns are required. For example:

```
/* Function to find greater of two numbers */
int greater (int x, int y)
{
        if (x>y)
                return (x);
        else
                return (y);
}
```

6.7 Types of Function Invoking

We categorize a function's invoking (calling) depending on arguments or parameters and their returning value. In simple words we can divide a function's invoking into four types depending on whether parameters are passed to a function or not and whether a function returns some value or not.
The various types of invoking functions are:

- Function with no arguments and no return value
- Function with no arguments and a return value
- Function with arguments and no return value
- Function with arguments and a return value

Let us discuss each category with some examples:

TYPE 1: Function with no arguments and no return value

As the name suggests, any function which has no arguments and does not return any values to the calling function, falls in this category. These type of functions are confined to themselves i.e., neither do they receive any data from the calling function nor do they transfer any data to the calling function. So there is no data communication between the calling and the called function but only program control will be transferred.

Example 6.7.1

```
/* Program for illustration of the function with no argument and no return value */
#include <stdio.h>
void checkPrimeNumber();

int main()
{
        checkPrimeNumber();    // no argument is passed to prime()
         return 0;
}

// return type of the function is void becuase no value is returned from the function
void checkPrimeNumber()
{
   int n, i, flag=0;
   printf("Enter a positive integer: ");
   scanf("%d",&n);
   for(i=2; i <= n/2; ++i)
{
        if(n%i == 0)
         {
            flag = 1;
           }
 }
if (flag == 1)
          printf("%d is not a prime number.", n);
else
        printf("%d is a prime number.", n);
}
```

TYPE 2: Function with no arguments and with return value

As the name suggests, any function which has no arguments but returns a value to the calling function, falls in this category.

Example 6.7.2

```
/* Program for illustration of the function with no argument and with return value */
#include <stdio.h>
int getInteger();
int main()
{
          int n, i, flag = 0;
           n = getInteger();            // no argument is passed to the function and
                                        //the value returned from the function is assigned to n
          for(i=2; i<=n/2; ++i)
         {
                  if(n%i==0)
                  {
                  flag = 1;
                  break;
                  }
         }
        if (flag == 1)
                  printf("%d is not a prime number.", n);
          else
                  printf("%d is a prime number.", n);

         return 0;
}
// getInteger() function returns integer entered by the user
```

```
int getInteger()
{
        int n;
        printf("Enter a positive integer: ");
        scanf("%d",&n);

        return n;

}
```

TYPE 3: Function with arguments and have no return value

If a function includes arguments but does not return any value to the calling function, falls in this category.

Example 6.7.3

```
/* Program for illustration of the function with argument and no return value */
#include <stdio.h>
void checkPrimeAndDisplay(int n);

int main()
{
   int n;
   printf("Enter a positive integer: ");
   scanf("%d",&n);
   checkPrimeAndDisplay(n);   // n is passed to the function
   return 0;
}

// void indicates that no value is returned from the function
void checkPrimeAndDisplay(int n)
{
   int i, flag = 0;
   for(i=2; i <= n/2; ++i)
   {
      if(n%i == 0){
         flag = 1;
         break;
      }
   }
   if(flag == 1)
      printf("%d is not a prime number.",n);
   else
      printf("%d is a prime number.", n);
}
```

TYPE 4: Function with arguments and return value

If a function includes arguments and also returns a value to the calling function, falls in this category. In this category two way communication takes place between the calling and the called function.

Example 6.7.4

```
/* Program for illustration of the function with argument and return value */
#include <stdio.h>
int checkPrimeNumber(int n);
int main()
```

```
{
   int n, flag;
   printf("Enter a positive integer: ");
   scanf("%d",&n);
   flag = checkPrimeNumber(n);      // n is passed to the checkPrimeNumber() function
                                                // the value returned from the function is assigned to flag variable
   if(flag==1)
         printf("%d is not a prime number",n);
   else
        printf("%d is a prime number",n);
   return 0;
}

int checkPrimeNumber(int n)
{
   /* Integer value is returned from function checkPrimeNumber() */
   int i;
   for(i=2; i <= n/2; ++i)
   {
     if(n%i == 0)
          return 1;
   }
   return 0;
}
```

6.8 STORAGE CLASS

Every variable in C programming has two properties: type and storage class.
Type refers to the data type of a variable. And, storage class determines the scope and lifetime of a variable.
There are 4 types of storage class:

1. automatic
2. external
3. static
4. register

6.8.1 Local Variable

The variables declared inside the function are automatic or local variables. The local variables exist only inside the function in which it is declared. When the function exits, the local variables are destroyed.

```
int main( )
{
   int n; // n is a local variable to main() function
   ... .. ...
}

void func() {
  int n1; // n1 is local to func() function
}
```

In the above code, n1 is destroyed when func() exits. Likewise, n gets destroyed when main() exits.

6.8.2 Global Variable

Variables that are declared outside of all functions are known as external variables. External or global variables are accessible to any function.

Example 6.8.2

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

```
void display();

int n = 5;  // global variable

int main()
{
   ++n;     // variable n is not declared in the main() function
   display();
   return 0;
}

void display()
{
   ++n;     // variable n is not declared in the display() function
   printf("n = %d", n);
}
```

Output

n = 7

Suppose, a global variable is declared in file1. If you try to use that variable in a different file file2, the compiler will complain. To solve this problem, keyword **extern** is used in file2 to indicate that the external variable is declared in another file.

6.8.3 Register Variable

The register keyword is used to declare register variables. Register variables were supposed to be faster than local variables. However, modern compilers are very good at code optimization and there is a rare chance that using register variables will make your program faster.

Unless you are working on embedded system where you know how to optimize code for the given application, there is no use of register variables.

6.8.4 Static Variable

A static variable is declared by using keyword static. For example;

```
static int i;
```

The value of a static variable persists until the end of the program.

Example 6.8.4: Static Variable

```
#include <stdio.h>
void display();

int main()
{
   display();
   display();
   display();
}
void display()
{
   static int c = 0;
   printf("%d  ",c);
   c += 5;
}
```

Output

0 5 10

During the first function call, the value of c is equal to 0. Then, it's value is increased by 5.

During the second function call, variable c is not initialized to 0 again. It's because c is a static variable. So, 5 is displayed on the screen.

6.9 RECURSION

In C, it is possible for the functions to call themselves. A function is called 'recursive' if a statement within the body of a function calls the same function. Sometimes called 'circular definition', recursion is thus the process of defining something in terms of itself.

Let us now see a simple example of recursion. Suppose we want to calculate the factorial value of an integer. As we know, the factorial of a number is the product of all the integers between 1 and that number. For example, 4 factorial is 4 * 3 * 2 * 1. Thus factorial of a number can be expressed in the form of itself. Hence this can be programmed using recursion.

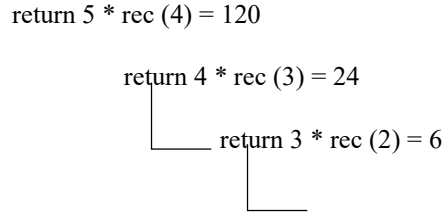
Example 6.9

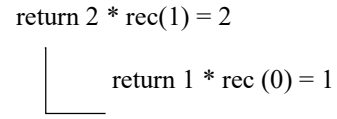
```
/* Following is the recursive version of the function to calculate the factorial value. */
main( )
{
      int rec(int);

      int a, fact ;
      printf ( "\nEnter any number " ) ;
      scanf ( "%d", &a ) ;
      fact = rec ( a ) ;
      printf ( "Factorial value = %d", fact ) ;
}

int rec ( int x )
{
       int f ;
       if ( x == 1 )
             return ( 1 ) ;
      else
             f = x * rec ( x - 1 ) ;       //rec() being called from inside itself
      return ( f ) ;
}
```

And here is the output of the program

```
Enter any number 5
Factorial value = 120
```

Let us understand this recursive factorial function thoroughly. In case the value of a is 5, main( ) would call rec( ) with 5 as its actual argument, and rec( ) will send back the computed value. But before sending the computed value, rec( ) calls rec( ) and waits for a value to be returned. It is possible for the rec( ) that has just been called to call yet another rec( ), the argument x being decreased in value by 1 for each of these recursive calls. We speak of this series of calls to rec( ) as being different invocations of rec( ). These successive invocations of the same function are possible because the C compiler keeps track of which invocation calls which. These recursive invocations end finally when the last invocation gets an argument value of 1, which the preceding invocation of rec( ) now uses to calculate its own f value and so on up the ladder. So we might say what happens is,





rec (5) returns (5 times rec (4),
which returns (4 times rec (3),
which returns (3 times rec (2),
which returns (2 times rec (1),
which returns (1)))))

6.9.1 ITERATION Vs RECURSION

To understand the **difference between Recursion and Iteration** implemented through loops, let's first consider a simple program,

```
/* down_recur.c -- program counts down ways */
#include <stdio.h>
void countdown(int);

int main(void)
{
   int num = 100;
   countdown(num);
    return 0;
}

void countdown(int count)
{
   if (count >= 1) {
      printf("%d\n", count);
      countdown(count--);
   }
}
```

Observe the output below, for simplicity, some portion of output is displayed, dots … indicate numbers in sequence,

```
100
99
98
97
96
95
.
.
.
6
5
4
3
2
1
```

Now, we try to perform the same job using loops, for example,

```
/*
 * diff_recur_and_loops.c -- program displays integers 100 through 1 using
 * loops
 */
#include <stdio.h>
int main(void)
{
```

```
    int num = 100;

    for (; num >= 1; num--) {
        printf("%d\n", num);
    }
    return 0;
}
```

Output of the above program is as follows,

```
100
99
98
97
96
95
.
.
.
6
5
4
3
2
1
```

Both outputs are same. Where's the difference in two approaches? let's unravel this, now.

Notice that each time 'countdown()' is called recursively, formal integer argument 'count' is created and it obtained copy of integer from previous recursive call of function 'countdown()'. Therefore, there were created 101 integers with same name 'count', last one with value 0. Each 'count' is private/local to its function and so there's no same name conflict. We know **in C that when a function is called, it's allotted a portion of temporary area called STACK for placing up there its private/local arguments, return addresses of calling functions etc.** This way, for 101 recursive function calls to itself, there were created 101 STACKS, one for each function. Further, this caused recursion slow. Also, **recursion exhausts systems important memory resources and might cause the program to abnormally aborted.**

Also, we observed here that we must design some condition in recursive function which during successive recursive calls eventually fails and recursion stops. If type of recursive function isn't void, each recursive call only after recursion stopped returns that type of value to its calling function.

Now, we discuss same problem using loops. Here, we implemented for loop, just took one integer variable 'num' and performed the job efficiently with single variable. There were no trade-offs involved in creating separate STACK for every Iteration instead iterations used the modified single integer and performed the task efficiently.

6.10 Summary

- C functions are basic building blocks in a program. All C programs are written using functions to improve re-usability, understandability and to keep track on them.
- Any function can be called from any other function. Even **main( )** can be called from other functions.
- The standard library functions are built-in functions in C programming to handle tasks such as mathematical computations, I/O processing, string handling etc
- A function prototype gives information to the compiler that the function may later be used in the program.
- When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.
- The type of arguments passed to a function and the formal parameters must match, otherwise the compiler throws error.
- Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

- The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
- By default, C programming uses *call by value* to pass arguments.
- The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter.
- Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code.
- Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.
- You can pass any number of arguments to a function but can return only one value at a time.
- If a function does not return anything, ***void*** specifier is used in the function declaration.
- The storage class determines the scope and lifetime of a variable.
- There are 4 types of storage class:
    - automatic
    - external
    - static
    - register
- A function is called 'recursive' if a statement within the body of a function calls the same function.

6.11 Questions for Exercises

1. What is Function? Explain different types of functions with examples.
2. Discuss about Global and Local variables.
3. What is storage class?
4. What is recursion**?** Explain with example. Differentiate between Iteration and Recursion.
5. Explain Call-by-value and Call-by-reference giving proper examples.
6. Write a function that receives 5 integers and returns the sum, average and standard deviation of these numbers. Call this function from **main( )** and print the results in **main( )**.
7. A 5-digit positive integer is entered through the keyboard, write a function to calculate sum of digits of the 5-digit number:
8. Without using recursion
9. Using recursion
10. Write a recursive function to obtain the first 25 numbers of a Fibonacci sequence. In a Fibonacci sequence the sum of two successive terms gives the third term. Following are the first
    few terms of the Fibonacci sequence:
    1 1 2 3 5 8 13 21 34 55 89...
11. Write a function to find the binary equivalent of a given decimal integer and display it.

6.12 Suggested Readings

- *Programming in C*, Third Edition, Stephen G. Kochan
- *Let Us C***,** Fifth Edition, Yashavant P. Kanetkar
- *The C programming language*, Brain W Kernighan, Dennis M Ritchie, PHI.
- *Programming with C*, Second Edition, Byron Gottfried, Tata McGraw Hill, 2003.
- *C. The Complete Reference*, Fourth Edition, Herbert Schildt, Tata McGraw Hill, 2002.
- *The C Primer*, Leslie Hancock, Morris Krieger, Mc Gravy Hill, 2013.