CHAPTER - 6
Functions

1. Function

A function is a collection of related statements that is used to perform a specific task. The specific task is repeated each time when the function is called. So, this avoids the need of rewriting the same code again and again. Every program must contain one function named main () from where the program always begin execution.

Complex problems can be solved by breaking them into a set of sub-problems, called Modules or Functions. This technique is called divide and conquer. Each module can be implemented independently and later can be combined into a single unit.

2. Advantages of Function

- **Avoids redundant code:** The repeated instructions can be written as a function, which can then be called whenever it is needed. So there is no chance of code duplication.
- **Code reusability:** The functions created in one program can be accessed in other programs.
- C functions can be used to build a customized library of frequently used routines.
- Program can be developed in short period of time using function.
- It can be used in the minimization of code.

3. Types of Function

C functions are classified into two categories. They are:

3.1 Library functions

The function that does not require the definition to be written by user is called library function. They are created by the developer. Their function and declaration are predefined. Example: printf(), scanf(), gets(), puts(), strrev(), getch(), clrscr(), etc.

3.2 User_defined_functions

The function developed by the user at the time of writing the program is called user defined function. The user defines the function code according to their requirement. Example: sum () to calculate the sum of given values.

Difference Between Library Function And User-Defined Function

S.N	Library Function	User-Defined Function
1	These are predefined and compiled functions that are placed in C-library.	These are defined by the user at the time of writing a program as per the own requirement.
2	The function's name, return type, arguments and their types cannot be changed.	The function's name, return type, arguments and their types can be changed.
3	It is difficult to understand.	It is easier to understand.
4	These are the part of header files which are called during runtime.	These are the part of the program which are compiled during runtime.
5	Example: printf(), scanf(), clrscr(), etc.	Example: sum(), reverse(), etc.

4. Parts of Function

A function consists of three parts namely,

- Function Prototype or Declaration
- Function call
- Function Definition

4.1 Function Prototype or Declaration

The prototype of a function declares the function in the same way as we declare the variables. To invoke any function, it must be declared first. Function declaration or prototype tells the program about the type of value returned by the function and the number and type of arguments. Function prototyping means writing the function prototypes before their function call occurs. The prototyping is mandatory in C. The advantage of function prototyping is that it enables a

compiler to compare each use of function with the prototype to determine whether the function is invoked properly or not. The number and types of arguments can be easily compared and any wrong number or type of the argument is reported. It consists of four parts:

- Function return type
- Function name
- Parameter list
- Terminating semicolon

Syntax:

Function-return-type function-name (parameter-list);

4.2 Defining a function (Function Definition)

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function. The function definition consists of function header and function body.

Syntax:

```
return_type  function_name(parameter  list  )
{
body  of  the  function
}
```

The return_type, function_name and parameter –list represent the function header and the function body contains local variable declaration, function statements and a return statement. The parts of a function are:

- **Return Type:** The return-type is the data type of the value the function returns. A function may return a value or may perform the desired operations without returning a value. If the function does not return any value, then the return-type is represented by keyword **void**. The return statement returns the value to the calling function. The return statement without any value returns the garbage value to the calling function. There is no restriction on the number of return statements that may be present in the function.
- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters (or argument)**: A parameter is like a placeholder. It is used to convey information to a function. Parameters are optional i.e. a function may or may not contain any parameters. The parameters are of two types: Actual parameters and Formal parameters. Parameters defined inside the function are called formal parameters whereas the parameters passed during the function call are known as actual arguments.
- **Function Body:** The function body contains a collection of statements used to perform a specific task. The function body is enclosed within the braces.

4.3 Function Call (Passing arguments to a Function)

A function is used to transfer the control to the function definition. A function can be called by using the function name followed by a list of actual parameters and semicolon. When the function completes its task, the control is return to the position from where the function is called.

Syntax:

Function-name (actual parameters);

There are two approaches to pass the information to a function through arguments. They are: Call by Value and Call by Reference.

4.3.1 Argument (or Parameter) Pass By Value (Function Call By Value)

The process of passing the value of variable in function is called function call by value. Arguments are usually passed by value in C function calls. Actual and formal arguments refer to the different memory locations and the value of actual argument is copied into the formal argument. So, any changes made to the formal argument are not reflected in their corresponding actual arguments. The value of the actual argument will remain same.

Example: Program that illustrates call by value mechanism

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

```
void swap(int, int);
void main()
{
int a, b;
a=10;
b=20;
swap(a, b); //Argument pass by value
printf("\n%d %d", a, b); // prints 10 20
getch();
}
void swap(int c, int d)
{
int temp;
temp = c;
c = d;
d = temp;
printf("%d %d", c, d); //prints 20 10
}
```

4.3.2 Argument Pass By Reference (Function Call By Reference)

In this approach, the addresses of actual arguments are passed to the function call and the formal arguments will receive the address. The actual and formal arguments refer to the same memory location. So, changes in the formal arguments are reflected in actual arguments. Actual arguments are address of the ordinary variable, pointer variable or array name. It helps in returning back more than one value to the calling function.

Example: Program that illustrates call by reference mechanism

```
#include<stdio.h>
#include<conio.h>
void swap(int *x, int *y);
void main()
{
int a, b;
a=10;
b=20;
swap(&a, &b); // Argument pass by reference where the addresses of a and b are passed
 printf("\n%d %d", a, b); // prints 20 10
getch();
}
void swap(int *c, int *d) // reference is made
{
int temp;
temp = *c;
*c = *d;
*d = temp;
printf("%d %d", c, d); //prints address represented by c and d
}
```

5. Category of Functions

a function can be categorized in four types on the basis of whether arguments are present or not and whether a value is returned or not.

- **Function with parameters but no return type**

  Example: Program to illustrate function with parameters but no return type

```
#include<stdio.h>
#include<conio.h>
void add(int, int);
void main()
{
int a=5, b=2;
add(a,b);
getch();
clrscr();
}
void add(int a, int b)
{
int c;
c=a+b;
printf(“\n The sum of two numbers is: %d”,c);
}
```

- **Function with parameters and return type**

Example: Program to illustrate function with parameters and return values.

```
#include<stdio.h>
#include<conio.h>
int  add(int,int);
void  main()
{
int  a,b,result;
printf("Enter  the  values  of  two  numbers:");
scanf("%d%d",&a,&b);
result=add(a,b);
printf("\nThe  sum=%d",result);
getch();
clrscr();
}
int  add(int  p,int  q)
{
int  sum;
sum=p+q;
return(sum);
}
```

- **Function with no parameters and no return type**

Example: Program to illustrate function with no parameters and no return values.

```
#include<stdio.h>
#include<conio.h>
void  add();
void  main()
{
add();
getch();
clrscr();
}
void  add()
{
int  a,b,sum;
printf("Enter  two  numbers:");
scanf("%d%d",&a,&b);
```

```
sum=a+b;
printf("\nsum=%d",sum);
}
```

- **Function with no parameters but return type**

```
Program to illustrate function with no parameters but return values.*/
#include<stdio.h>
#include<conio.h>
int  add();
void  main()
{
int  result;
result=add();
printf("\nThe  sum=%d",result);
getch();
clrscr();
}
int  add()
{
int  a,b,sum;
printf("Enter  two  numbers:");
scanf("%d%d",&a,&b);
sum=a+b;
return(sum);
}
```

6. Macro

Macro is a preprocessor which hold the value of any constant so it is also called symbolic constant. It is globally declared before the main functions. It makes execution fast because of pre-execution but takes more memory space. Macro and its expansion are separated by blanks or tabs. Macro definition is never terminated by a semicolon.

Use of macro

- The execution is fast due to the pre-execution
- It reduces the program size.
- The macro is useful in the place where the numbers are not to be allowed.

For example: WAP to calculate the area of a circle using macro.

```
#include<stdio.h>
#include<stdio.h>
#include<conio.h>
#define  PI  3.14159
float area(float);
void  main()
{
float  r,a;
printf("Enter  the  radius  of  a  circle:  ");
scanf("%f",&r);
a=area(r);
printf("The  area  of  a  circle  is  %f",a);
getch();
clrscr();
}
float  area(float  radius)
{
float  area;
```

```
area=PI*radius*radius;
return(area);
}
```

Here,
# = Pre-processor
define = Macro representation
PI = Macro template or Macro
3.14159 = Macro expansion

6.1 Predefined Macros

C defines a number of macros that should not be directly modified.

Macro	Description
__DATE__	The current date as a character literal in "MMM DD YYYY" format
__TIME__	The current time as a character literal in "HH:MM:SS" format
__FILE__	This contains the current filename as a string literal.
__LINE__	This contains the current line number as a decimal constant.

For example:

```
#include<stdio.h>
#include<conio.h>
void  main()
{
printf("File  :%s\n",  __FILE__  );
printf("Date  :%s\n",  __DATE__  );
printf("Time  :%s\n",  __TIME__  );
printf("Line  :%d\n",  __LINE__  );
getch();
}
```

Output

```
File  :test.c
Date  :Jun  2  2012
Time  :03:36:24
Line  :8
```

6.2 Parameterized Macros

Macros with arguments must be defined using the #define directive before they can be used. The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between and macro name and open parenthesis.

For example - 1:

```
#include  <stdio.h>
#define  MAX(x,y)  ((x)  >  (y)  ?  (x)  :  (y))
void  main()
{
printf("Max  between  20  and  10  is  %d\n",  MAX(10,  20));
}
```

Output: Max between 20 and 10 is 20

7. Storage Classes

Variables in C can be characterized by their data type and storage classes. Data type refers to the type of information represented by a variable and storage classes define its life time and scope.

The storage class generally specifies:

1. **Storage**: Where the variable would be stored i.e. memory or CPU registers.
2. **Default initial value**: What will be the initial value of the variable, if the initial value is not specifically assigned?
3. **Scope:** What is the scope of the variable i.e. in which function the value of the variable is available and where it can be used.
4. **Life time:** What is the life of the variable i.e. how long the variable value would exist in the memory?

Syntax:
storage-class-specifier type-specifier variable-name;

7.1 Automatic or Local variables (Auto storage class)

Storage Location	Memory
Default initial value	Unpredictable (garbage value)
Scope	Local to the function in which the variable is defined
Life time	Till the control remain in the function in which it is defined

If no storage class is specified, by default it is an auto variable.

Example:

```
#include<stdio.h>
#include<conio.h>
void function1(void);
void function2(void);
void main()
{
        auto int m = 1000;
        function1();
        function2();
        printf ("%d \n" , m);
        getch();
}
void function1()
{
        auto int m = 10;
        printf ("%d \n" , m);
}
void function2()
{
        auto int m = 100;
        printf ("%d \n" , m);
}
```

Output:

```
10
100
1000
```

7.2 Static variables (static storage class)

Storage Location	Memory
Default initial value	Zero
Scope	Local to the function in which the variable is defined
Life time	Life of the program (till the end of the program)

Example

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

```
void  incre(void);
void  main()
{
int  i;
incre();
incre();
getch();
}
void  incre()
{
        static  int  x ;
        x  =  x  +1;
        printf("  x  =  %d\n",x);
}
```

Output:

x=1

x=2

7.3 External or Global Variables (extern storage class)

Storage Location	Memory
Default initial value	Zero
Scope	Global
Life time	As long as the programs execution does not come to end.

The variables that are common to all the functions are declared outside the functions. If it is declared in the Global declaration section, it is used by all the functions in the program. Memory for the global variables is allocated when the program gets executed and de allocated only at the end of program execution. The return statement is used to transfer the control back to the calling program.

If we use the 'extern' keyword then we must initialize the value.

Example:

```
#include<stdio.h>
#include<conio.h>
int count=0;
void main()
{
        printf("the initial count value=%d",count);
        fun();
}
fun()
{
        count++;
        printf("\nthe  count value=%d",count);
}
```

Output:

the initial count value=0

the count value=1

7.4 Register variables (register storage class)

Storage Location	CPU Registers
Default initial value	Garbage value
Scope	Local to the function
Life time	Till the control remain in the function in which it is defined

It is possible to inform the compiler that a variable should be kept in one of the registers, instead of keeping it in the memory. Since registers are faster than memory, keeping the frequently accessed variables like a loop control variable in a register will increase the execution speed. Since the registers are less in numbers, careful selection must be made for their use. If the declaration of register variable exceeds the availability, they will be automatically converted into non register variables (automatic variable). Only the integer values are declared in register storage class.

8. Recursive Function (Nesting of Function)

In C, it is possible for the functions to call themselves. A function that calls itself repeatedly until the specific condition has been satisfied is called recursive function. And recursion is a process by which function call itself repeatedly.

Example1: Factorial of given number using recursive function (2013 fall)

```
#include<stdio.h>
#include<conio.h>
int factorial (int);
void main()
{
        int n,res;
        printf("enter the number whose factorial is to be calculated: ");
        scanf("%d",&n);
        res=factorial(n);
        printf("\nthe factorial of %d is %d",n,res);
        getch();
        clrscr();
}
int factorial(int n)
{
        if(n==0||n==1)
        {
                return(1);
        }
        else
        {
                return(n*factorial(n-1));
        }
}
```

Examples of Function

Example1: WAP to find the square and sum of square of two arguments using function

```
#include <stdio.h>
#include <conio.h>
int sqr(int);
int sumsqr(int,int);
void main()
{
int num1,num2,res1,res2,sum;
printf("Enter two numbers:");
scanf ("%d %d",&num1,&num2);
res1= sqr(num1);
res2= sqr(num2);
sum= sumsqr(num1,num2);
printf ("The square of %d is %d",num1,res1);
printf("\nThe square of %d is %d",num2,res2);
printf("\nSum of squares is %d",sum); getch();
getch();
clrscr();
}

int sqr(int z)
{
return(z*z);
}

int sumsqr(int x, int y)
{
return(x*x+y*y);
}
```

Example 2: Write a program to calculate the area and perimeter of a rectangle using function

```
#include<stdio.h>
#include<conio.h>
void area(int,int);
void perimeter(int,int);
void main()
{
int l,b;
printf("Enter length and breadth:");
scanf("%d%d",&l,&b);
area(l,b);
perimeter(l,b);
getch();
clrscr();
}

void area(int x,int y)
{
int area1;
```

```
area1=x*y;
printf("area=%d",area1);
}
void  perimeter(int  x,int  y)
{
int  peri;
peri=2*(x+y);
printf("\nperimeter=%d",peri);
}
```

Example 3: WAP to find factorial using function

```
#include<stdio.h>
#include<conio.h>
int  factorial(int);
void  main()
{
int n,result;
printf(“Enter the positive number:”);
scanf(“%d”,&n);
result=factorial(n);
printf(“\nFactorial=%d”,result);
getch();
clrscr();
}
int factorial (int n)
{
int i, fact=1;
for(i=1; i<=n; i++)
{
fact=fact*i;
}
return (fact);
}
```

Example 4: Program to calculate the sum of array elements by passing to a function

```
#include <stdio.h>
float calculateSum(float age[]);

int main()
{
   float result, age[] = {23.4, 55, 22.6, 3, 40.5, 18};

   // age array is passed to calculateSum()
   result = calculateSum(age);
   printf("Result = %.2f", result);
   return 0;
}

float calculateSum(float age[]) {

  float sum = 0.0;
```

```
  for (int i = 0; i < 6; ++i) {
                        sum += age[i];
  }

  return sum;
}
```