FUNCTIONS

Introduction

- A function is a sub program which contains a set of instructions that are used to perform specified tasks
- A function is used to provide modularity to the software. By using function can divide complex task into manageable tasks. The function can also help to avoid duplication of work.

Advantages of Functions

- ✓ Code reusability
- ✓ Better readability
- ✓ Reduction in code redundancy
- ✓ Easy to debug & test.

How Does Function Work?

- Once a function is called, it takes some data from the calling function and returns back some value to the called function.
- Whenever function is called control passes to the called function and working of the calling function is temporarily stopped, when the execution of the called function is completed then a control return back to the calling function and executes the next statement.
- The function operates on formal and actual arguments and send back the result to the calling function using return() statement.

Types of Functions

Functions are classified into two types

a) User defined functions

b) Predefined functions or Library functions or Built-in functions

a) User-defined functions

- User-defined functions are defined by the user at the time of writing a program.

  Example: sum( ), square( )

b) Library functions [Built-in functions]

- Library functions are predefined functions. These functions are already developed by someone and are available to the user for use.

Example: printf( ), scanf( )

Terminologies used in functions

- A function f that uses another function g is known as the calling function, and g is known as the called function.
- The inputs that a function takes are known as arguments.
- When a called function returns some result back to the calling function, it is said to return that result.
- The calling function may or may not pass **parameters** to the called function. If the called function accepts arguments, the calling function will pass parameters, else not.
- **Function declaration** is a declaration statement that identifies a function's name, a list of arguments that it accepts, and the type of data it returns.
- **Function definition** consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function.

Function prototype

User Defined Function

- The function defined by the users according to their requirements is called user defined functions. The users can modify the function according to their requirement.

Need For user Defined Functions

- While it is possible to write any complex program under the main () function and it leads to a number of problems, such as
- The program becomes too large and complex
- The users cannot go through at a glance.
- The task of debugging, testing and maintenance becomes difficult.

Advantages of user Defined Functions

- The length of the source program can be reduced by dividing it into the smaller functions.
- By using functions it is very easy to locate and debug an error.
- The user defined function can be used in many other source programs whenever

necessary.

- Functions avoid coding of repeated programming of the similar instructions.
- Functions enable a programmer to build a customized library of repeatedly used routines.
- Functions facilitate top-down programming approach.

Elements of user Defined Functions

- In order to write an efficient user defined function, the programmer must be familiar with the following three elements.
  - Function declaration
  - Function definition
  - Function call

Function Declaration

- A function declaration is defined as a prototype of a function which consists of the functions return type, function name and arguments list
- It is always terminated with semicolon (;)
- Function prototypes can be classified into four types
  a) Function with no arguments and no return values
  b) Function with arguments and no return values
  c) Function with arguments and with return values
  d) Function with no arguments and with return values

Syntax

return_type function_name (parameter_list);

Where,

return_type can be primitive or non-primitive data type

function_name can be any user specified name

parameter_list can consist of any number of parameter of any type

Example

void add(void);

void add(int,int);

int add(int,int);

int add(void);

a) Function with No Arguments and No Return Values

- In this prototype, no data transfer takes place between the calling function and the called function. (i.e) the called program does not receive any data from the calling program and does not send back any value to the calling program.

Syntax:

```
void function_name(void);
void main()
{
…….
function_name();
.....
}
void function_name(void)
{
.....
……
}
```

Program 1.22

/*Implementation of function with no return type and no argument list*/

```
#include<stdio.h>
#include<conio.h>
void add(void); //function declaration with no return type and no
arguments list
void main()
{
        add(); //function call
}
void add()
{
```

```
int a,b,c;
printf("Enter the two numbers . . . ");
scanf(" %d %d",&a,&b);
c=a+b;
printf("sum is . . . %d",c);
}
```

Output:

Enter the two numbers . . . 10 20

Sum is . . . 30

b) Function with Arguments and No Return Values

- In this prototype, data is transferred from calling function to called function. i.e the called program receives some data from the calling program and does not send back any values to calling program.

Syntax:

```
void function_name(arguments_list);
void main()
{
…….
function_name(argument_list);
.....
}
void function_name(arguments_list)
{
//function body
}
```

Program 1.23

/*Implementation of function with no return type and with argument list*/

```
#include<stdio.h>
#include<conio.h>
int add(int,int); ); //function declaration with no return type and with arguments
```

```
list
void main()
{
        int a,b,;
        clrscr();
        printf("enter the two values: ");
        scanf(" %d %d",&a,&b);
        add(a,b); /*calling a function with arguments */
        getch();
}
void add(int x,int y)
{
        int z;
        z=x+y;
        printf("Sum is.....%d",z);
}
```

Output:

Enter two values: 10 20

Sum is 30

c) Function with arguments and With Return Values

- In this prototype, data is transferred between calling function and called function.(i.e) the called program receives some data from the calling program and send back a return value to the calling program.
- Value received from a function can be further used in rest of the program.

Syntax:

```
return_type function_name(arguments_list);
void main()
{
…….
variable_name=function_name(argument_list);
```

```
.....
}
return_type function_name(arguments_list)
{
//function body
}
```

Program 1.24

```
#include<stdio.h>
#include<conio.h>
int add(int,int); //function declaration with return type and with arguments list
void main()
{
        int a,b,c;
        clrscr();
        printf("enter the two numbers: ");
        scanf(" %d %d",&a,&b);
        c=add(a,b); /*calling function with arguments*/
        printf("sum is . . . %d ", c);
        getch();
}
int add(int x,int y)
{
        int z;
        z=x+y;
        return(z); /*returning result to calling function*/
}
```

Output:

Enter the two numbers: 10 20

sum is. . 30

d) Function with No Arguments and with Return Values

- In this prototype, the calling program cannot pass any arguments to the called program. i.e) program may send some return values to the calling program.

Syntax:

```
return_type function_name(void);
void main()
{
…….
variable_name=function_name();
.....
}
return_type function_name(void)
{
//function body
}
```

Program 1.25

```
/*Implementation of function with return type and no argument list*/
#include<stdio.h>
#include<conio.h>
int add(void); //function declaration with no return type and no arguments list
void main()
{
        int c;
        c=add(); //function call
        printf("sum is . . . %d",c);
}
int add(void)
{
        int a,b,c;
        printf("Enter the two numbers . . . ");
```

```
    scanf(" %d %d",&a,&b);
    c=a+b;
    return(c);
}
```

Function Definition

- It is the process of specifying and establishing the user defined function by specifying all of its elements and characteristics.
- When a function is defined, space is allocated for that function in the memory.
- A function definition comprises of two parts:
  - ✓ Function header
  - ✓ Function Body

Syntax

```
return_type function_name(argument_list)
{
//function body
}
```

Example

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

Program 1.26

```
#include<stdio.h>
// function prototype, also called function declaration
float square ( float x );
// main function, program starts from here
int main( )
```

```
{
   float m, n ;
   printf ( "\nEnter some number for finding square \n");
   scanf ( "%f", &m ) ;
   // function call
   n = square ( m ) ;
   printf ( "\nSquare of the given number %f is %f",m,n );
}
float square ( float x )  // function definition
{
   float p ;
   p = x * x ;
   return ( p ) ;
}
```

Output

Enter some number for finding square

2

Square of the given number 2.000000 is 4.00000

Function Call

- The function can be called by simply specifying the name of the function, return value and parameters if presence.
- The function call statement invokes the function. When a function is invoked, the compiler jumps to the called function to execute the statements that are a part of that function. Once the called function is executed, the program control passes back to the calling function.

Syntax

function_name();

function_name(parameter);

variable_name=function_name(parameter);

variable_name=function_name();

Example

add();

add(a,b);

c=add(a,b);

c=add;

- There are two ways that a C function can be called from a program. They are,
  a) Call by value
  b) Call by reference

a) Function-Call by value

- In the call by value method the actual arguments are copied to the formal arguments, hence any operation performed by function on arguments doesn't affect actual parameters.

b) Function-Call by Reference

- Unlike call by value, in this method, address of actual arguments (or parameters) is passed to the formal parameters, which means any operation performed on formal parameters affects the value of actual parameters.

Actual parameters: The parameters that appear in function calls.

Formal parameters: The parameters that appear in function declarations.

Program 1.27

//Example program for Actual Parameter and Formal Parameters

```
#include <stdio.h>
int sum(int a, int b)
{
   int c=a+b;
   return c;
}
int main(
{
   int var1 =10;
   int var2 = 20;
   int var3 = sum(var1, var2);
```

```
    printf("%d", var3);
    return 0;
}
```

In the above example variable a and b are the formal parameters (or formal arguments). Variable var1 and var2 are the actual arguments (or actual parameters). The actual parameters can also be the values. Like sum(10, 20), here 10 and 20 are actual parameters.

Program 1.28

//Example of Function call by Value

```
#include <stdio.h>
int increment(int var)
{
   var = var+1;
   return var;
}
int main()
{
  int num1=20;
  int num2 = increment(num1);
  printf("num1 value is: %d", num1);
  printf("\nnum2 value is: %d", num2);
  return 0;
}
```

Output

num1 value is: 20

num2 value is: 21

Program 1.29

//Example 2: Swapping numbers using Function Call by Value

```
#include <stdio.h>
void swapnum( int var1, int var2 )
```

```
{
  int tempnum ;
  /*Copying var1 value into temporary variable */
  tempnum = var1;
  /* Copying var2 value into var1*/
  var1 = var2;
  /*Copying temporary variable value into var2 */
  var2 = tempnum;
}
int main( )
{
   int num1 = 35, num2 = 45;
   printf("Before swapping: %d, %d", num1, num2);
   /*calling swap function*/
   swapnum(num1, num2);
   printf("\nAfter swapping: %d, %d", num1, num2);
}
```

Output

Before swapping: 35, 45

After swapping: 45, 35

Program 1.30

Example 2: Function Call by Reference – Swapping numbers

```
#include
void swapnum ( int *var1, int *var2 )
{
  int tempnum ;
  tempnum = *var1;
  *var1 = *var2;
  *var2 = tempnum;
}
```

```
int main( )
{
  int num1 = 35, num2 = 45;
  printf("Before swapping:");
  printf("\nnum1 value is %d", num1);
  printf("\nnum2 value is %d", num2);
  /*calling swap function*/
  swapnum( &num1, &num2 );
  printf("\nAfter swapping:");
  printf("\nnum1 value is %d", num1);
  printf("\nnum2 value is %d", num2);
  return 0;
}
```

Output

Before swapping:

num1 value is 35

num2 value is 45

After swapping:

num1 value is 45

num2 value is 35

RECURSIVE FUNCTIONS

- Recursion is the process of calling the same function again and again until some condition is satisfied.
- This process is used for repetitive computation.

Syntax:

```
function_name()
{
function_name();
}
```

Types of Recursion

a) Direct Recursion

b) Indirect Recursion

a) Direct Recursion

- A function is directly recursive if it calls itself.

```
Functionname1( )
{
….
Functionname1 ( );     // call to itself
….
}
```

Example

- A function is said to be directly recursive if it explicitly calls itself. Here, the function Func() calls itself for all positive values of n, so it is said to be a directly recursive function.

```
int Func (int n)
{
if (n == 0)
return n;
else
return (Func (n–1));
}
```

b) Indirect Recursion

- Function calls another function, which in turn calls the original function.

```
Functionname1 ( )
{
…
Functionname2 ( );
…
}
Functionname1 ( )
```

```
{
…
Functionname1 ( ); // function (Functionname2) calls (Functionname1)
…
}
```

Example

- A function is said to be indirectly recursive if it contains a call to another function which ultimately calls it. These two functions are indirectly recursive as they both call each other.

```
int Funcl (int n)
{
if (n == 0)
return n;
else
return Func2(n);
}
int Func2(int x)
{
return Func1(x–1);
}
```

Program 1.31

/*C program to find factorial of a given number using recursion*/

```
#include< stdio.h>
#include<conio.h>
void main()
{
int fact(int);
int num,f;
clrscr();
printf("enter the number");
```

```
scanf("%d",&num);
f=fact(num);
printf(" the factorial of %d= %d", num, f);
}
int fact(int x)
{
int f;
if(x==1)
return(1);
else
f=x*fact(x-1); //recursive function call
return (f);
}
```

Output:

Enter the number 5

The factorial of 5=120