Module 4: Functions or Subprograms

Introduction to Functions

As we know, every C program permits the programmer to make use of only one main() to solve a problem. Hence, it is very difficult for the programmer to solve a large and complex problem by using a single main() with only built-in functions. Hence, programmers need to write to their own functions (sub programs) to do specific divided task. These subprograms written by programmers are called subprograms or user defined functions or modules.

The programming approach in which the given large and complex problem will be divided into many subprograms and are called by main() to do specific task is called modular programming approach.

What is function?

A function can be defined as a subprogram that also contains a group of statements to do specific task.

Advantages of writing user defined functions

1. Reduction in the size of main( ).
2. Reusability of the written functions.
3. Program maintenance, testing and debugging is easier.
4. Functions can be shared among many C files.
5. Programmers can create their own functions library like header files.
6. Modular programming approach.

Types of Functions:

In C language, functions are classified into following two types.

1. Built-in functions: The functions designed and developed by C developers are called built in or pre-defined functions.

   Examples: scanf(), printf(), getch(), pow(),sqrt(),malloc(), strlen(), strcpy(), etc.
2. User defined function: The functions written by programmers are called user defined functions.

   Examples: isprime(n), fact(n), strcopy(str1,str2), sort(a,n), sum(n1,n2), etc.

Elements of user defined functions:

In order to design and develop user defined functions , the programmer needs to establish and use the following three elements of user defined functions.

1. Function declaration or prototype
2. Function call
3. Function definition

The programming example that includes all the three elements is as follows.

1. Function declaration or prototype:
The function must be declared in the global declaration part of the C program i.e. before the main(). It provides three details of a function to be write to C-compiler i.e. return type, function name and number of inputs.

Syntax:

return_type function_name (arguments list);

where,
return_type→ The data type of return value like int, float, char, double etc.
The default return type is int.
function name→ valid identifier.
arguments list→ Number of inputs or parameters or arguments.

Examples:

```
int sum (int x, int y);   or int  sum(int, int)          /* to find sum*/
float area (float x);        or float area (float)        /* to find area of circle*/
int isprime (int x);                                      /* to check of prime number*/
```

2. Function call: The user defined function is called by the main() function to do specific task by passing the number of inputs.
Examples:

i)
```
void main()
{
…
ans1=square(n);          /* function call*/
ans2=cube(n);            /* function call*/
…
}
```

3. Function definition: This is the subprogram, which can be written before or after the main(). It also contains group of statements to do specific task.

Syntax:

```
return_type   function_name (arguments list)
{
local definitions;
statements;
return (value);
}
```

```
int sum(int x, int y)    /* function definition*/
{
return (x+y);
}
```

```
/* to find sum of two numbers by using function*/
#include<stdio.h>
int sum(int x, int y);    /* function prototype*/
void main()
{
int n1,n2,ans;
clrscr();
printf("\n Enter any two numbers");
scanf("%d%d",&n1,&n2);
ans=sum(n1,n2);           /* function call*/
printf("\n Addition is %d",ans);
getch();
}
int sum(int x, int y)    /* function definition*/
{
return (x+y);
}
```

```
o/p:
Enter any two numbers: 7 3
Addition is 10
```

```
/* to find square and cube of number by using functions*/
#include<stdio.h>
int square(int);    /* function declaration*/
int cube(int);
void main()
{
int n,ans1,ans2;
clrscr();
printf("\n Enter a number");
scanf("%d",&n);
ans1=square(n);        /* function call*/
ans2=cube(n);          /* function call*/
printf("\n Square is %d",ans1);
printf("\n Cube is %d",ans2);
getch();
}

int square(int x)    /* function definition*/
{
return (x*x);
}
int cube(int x)    /* function definition*/
{
return (x*x*x);
}
```

```
o/p:
Enter a number: 2
Square is 4
Cube is 8
```

```
/* to find factorial n by using function*/
#include<stdio.h>

long int fact(int x);    /* function declaration*/

void main()
{
int n;
long int ans;
clrscr();
printf("\n Enter a number");
scanf("%ld",&n);
ans=fact(n);          /* function call*/
printf("\n Factorial is %ld",ans);
getch();
}

long int fact(int x)    /* function definition*/
{
long int prod=1;
int i;
            for (i=1;i<=x;i++)
            {
            prod=prod*i;
            }
return(prod);
}
```

```
o/p:
Enter a number: 4
Factorial is 24
```

```
/* to check for prime number by using function*/
#include<stdio.h>
int isprime(int x);    /* function declaration*/
void main()
{
int n,r;
clrscr();
printf("\n Enter a number");
scanf("%d",&n);
r=isprime(n);          /* function call*/
if(r==1)
printf("\n %d is prime number",n);
else
printf("\n %d is not prime number",n);
getch();
}
int isprime(int x)    /* function definition*/
{
int i;
  for (i=2;i<=x/2;i++)
  {
     if(x%i==0)
     return 0;
  }
return(1);
}
```

```
o/p:
Enter a number: 17
17 is prime number

o/p:
Enter a number:24
24 is not prime number
```

```
/* to copy string1 into string2 by using user defined function*/
#include<stdio.h>
void STRCOPY(char str1[],char str2[]);    /* function declaration*/
void main()
{
char str1[50],str2[50];
clrscr();
printf("\n Enter string1 to be copy: ");
gets(str1);
STRCOPY(str1,str2);          /* function call*/
getch();
}
void STRCOPY(char str1[],char str2[])    /* function definition*/
{
int i;
  for (i=0;str1[i]!='\0';i++)
  {
    str2[i]=str1[i];
  }
  str2[i]='\0';
printf("\n String copied...");
printf("\n string1=%s string2=%s",str1,str2);
}
```

```
o/p:
Enter a string1 to be copy : Happy
String copied…
String1= Happy    String2=Happy
```

June/July 2015, Dec. 2015/Jan. 2016

Write a C program to sort the elements by passing array as function argument. (08 marks)

```
#include<stdio.h>
void sort(int a[], int x);

int main()
{
int a[25], i, n;
printf("Enter number of elements");
scanf("%d",&n);
printf("\n Enter %d  elements",n);
        for(i=0;i<n;i++)
        {
        scanf("%d",&a[i]);
        }
sort(a,n);
return(0);
}

void sort(int a[], int x)
{
int i,j,temp;
        for(i=0;i<n;i++)
        {
                for(j=0;j<n-i-1;j++)
                {
                        if [aj]>a[j+1])
                        {
                        temp=a[j];
                        a[j]=a[j+1];
                        a[j+1]=temp;
                        }
                }
        }
printf("\n After Sorting \n");
        for(i=0;i<n;i++)
        {
        printf("\n %d",a[i]);
        }

}
```

Actual and Formal Parameters

The inputs, whichever we pass to functions to do specific task are called parameters or arguments.

1. Actual parameters
2. Formal parameters

Actual Parameters:

The parameters passed during the function call in the main() function are called actual parameters. These are actual (original) input data passed to user define function to do specific task. i.e. the contents of actual parameters will be copied to formal parameters of a function.

Formal Parameters:

The parameters used in the function header of function definition are called formal parameters. These parameters receive input data for processing from the actual parameters from the calling function main(). i,e. formal parameters receive data from actual parameters. The changes made to formal parameters do not affect the contents of actual parameters.

Eqxamples:

```
/* Example for actual parameters */
#include<stdio.h>
void test(int x,int y);

void main()
{
int x=7,y=3;
clrscr();
printf("\n Before calling function: Actual parameters ");
printf("\n x=%d y=%d", x,y);
test(x,y);                              /* Here, x & y are actual parameters*/
printf("\n After calling function: Actual parameters ");
printf("\n x=%d y=%d", x,y);
getch();
}

void test(int x,int y)                  /* Here, x & y are formal parameters*/
{
x++;
y++;
printf("\n Inside function: Formal parameters");
printf("\n x=%d y=%d",x,y);
}
```

```
o/p:
Before calling function: Actual parameters
x = 7 y = 3
Inside function: Formal parameters
x = 8 y = 4
After calling function: Actual parameters
x = 7 y = 3
```

Categories of Functions based on number of arguments and return value:

The functions can be categorized into following 4 types based on the number of parameters or inputs they receive to do specific task and return value.

i. Functions with no arguments and no return value
ii. Functions with arguments and no return value
iii. Functions with arguments and return value
iv. Functions with no arguments and return value

i. Functions with no arguments and no return value:

These functions do not receive any inputs from the calling function main() and do not return any value back to it. But, these functions do specific task. i.e. there is no data transmission between calling and called function.

```
#include<stdio.h>
void sum(void);
void main()
{
clrscr();
sum();
getch();
}
void sum(void)
{
int n1,n2,ans;
printf("\nEnter two numbers:");
scanf("%d%d",&n1,&n2);
ans=n1+n2;
printf("\nAddition is %d",ans);
}
```

```
o/p:
Enter any two numbers:
7 3
Addition is 10
```

ii. Functions with arguments and no return value:

These functions receive inputs from the calling function main() to do specific task; but, do not return any value back to it. The processed data will be displayed by it.

Example:

```
#include<stdio.h>
void sum(int x,int y);
void main()
{
int n1,n2;
clrscr();
printf("\nEnter two numbers:");
scanf("%d%d",&n1,&n2);
sum(n1,n2);
getch();
}
void sum(int x,int y)
{
int ans;
ans=x+y;
printf("\nAddition is %d",ans);
}
```

```
o/p:
Enter any two numbers:
7 3
Addition is 10
```

iii. Functions with arguments and return value:
These functions receive inputs from the calling function main() to process and return a value back to it. i.e. data transmission takes place between calling and called function.

Example:
```
#include<stdio.h>
int sum(int x, int y);
void main()
{
int n1,n2,ans;
clrscr();
printf("\n Enter any two numbers");
scanf("%d%d",&n1,&n2);
ans=sum(n1,n2);
printf("\n Addition is %d",ans);
getch();
}
int sum(int x, int y)
{
return (x+y);
}
```

```
o/p:
Enter any two numbers: 7 3
Addition is 10
```

iv. Functions with no arguments and return value:
These functions do not receive any inputs from the calling function main(); but, return a value back to it.
Example:
```
#include<stdio.h>
int sum(void);
void main()
{
int ans;
clrscr();
ans=sum();
printf("\nAddition is %d",ans);
getch();
}
int sum(void)
{
int n1,n2;
printf("\nEnter two numbers:");
scanf("%d%d",&n1,&n2);
return(n1+n2);
}
```

```
o/p:
Enter any two numbers: 7 3
Addition is 10
```

Parameter Passing Mechanisms (important)

The mechanism or the process used to send the inputs or values from calling function main() to user defined function to do specific task is called parameter passing mechanism.

C programmer makes the use of following two methods to pass the values from actual parameters to formal parameters.

i. Call by value
ii. Call by reference or address

Call by value: This is one of the most commonly used parameter passing mechanism. In this method, the values are copied from actual parameter to formal parameters and changes made to formal parameters will not affect the actual parameters. This method returns only one value back to called function.

Examples:

```
/*Example for call by value method*/
#include<stdio.h>
void test(int x,int y);

void main()
{
int x=7,y=3;
clrscr();
printf("\n Before calling function: Actual parameters ");
printf("\n x=%d y=%d", x,y);
test(x,y);                  /* Here, x & y are actual parameters*/
printf("\n After calling function: Actual parameters ");
printf("\n x=%d y=%d", x,y);
getch();
}
void test(int x,int y)            /* Here, x & y are formal parameters*/
{
x++;
y++;
printf("\n Inside function: Formal parameters");
printf("\n x=%d y=%d",x,y);
}
```

```
o/p:
Before calling function: Actual parameters
x = 7 y = 3
Inside function: Formal parameters
x = 8 y = 4
After calling function: Actual parameters
x = 7 y = 3
```

Call by reference: The call by reference method makes the use of pointers to pass the addresses (references) of actual parameters to user defined function instead of values. In this method, changes made to formal parameters will affect the actual parameters. This method is having the capability to return multiple values back to called function, whereas pass by value returns only one value.

```
/* Example for call by reference or function to swap two values */

#include<stdio.h>

void swap(int *x,int *y);

void main()
{
int x=7,y=3;
clrscr();
printf("\n Before swap or Before calling function \n");
printf("\n x=%d y=%d \n",x,y);
swap(&x,&y);
printf("\n After Swap or After calling function \n");
printf("\n x=%d y=%d \n",x,y);
getch();
}

void swap(int *x, int *y)
{
int temp;
temp=*x;
*x=*y;
*y=temp;
}
```

```
o/p:
Before swap
x=7   y=3
After swap
X=3    y=7
```

Differences between Call by value and Call by references methods

Sl. No.	Call by value	Call by reference
1	In this method, the values will be passed from actual to formal parameters.	In this method, the addresses of actual parameters will be passed to formal parameters.
2	The changes made to formal parameters will not affect actual parameters.	The changes made to formal parameters will affect actual parameters.
3	Capable to return only one value back to called function.	Capable to return multiple values back to called function.
4	Program execution is slower in comparison with call by reference.	Program execution is faster in comparison with call by value.
5	Pointers knowledge is not required	Pointers knowledge is required.

6	Example:
	
	int sum(int x, int y)
	{
	return(x+y);
	}
		Example:
		void swap(int *x, int *y)
		{
		int temp;
		temp=*x;
		*x=*y;
		*y=temp;
		}

Recursive Functions

A function that calls itself to do specific task is called recursive function. The recursive functions are used by programmers to solve those problems in which the next action or calculation depends upon previous result.

Example:

```
/* to find factorial n by using recursive function*/
#include<stdio.h>

long int fact(int x);    /* function declaration*/

void main()
{
int n;
long int ans;
clrscr();
printf("\n Enter a number");
scanf("%d",&n);
ans=fact(n);          /* function call*/
printf("\n Factorial is %ld",ans);
getch();
}

long int fact(int n)
{
        if (n==0 || n==1)
                return(1)
        else
                return (n*fact(n-1));
}
```

```
o/p:
Enter a number: 4
Factorial is 24
```

Do it by your own: Write recursive function to find sum of 1 to n numbers.

Storage classes used in C or Lifetime of variables

The lifetime or visibility or accessibility of a variable used in the function depends upon storage class of the variable. The storage classes are classified into following four types.

1. automatic or local or private
2. global or public
3. static
4. register

1. automatic (local) variables:

These are defined inside the functions and their lifetime is local to the function in which it has been defined. By default, all the variables are of type automatic only. These variables are also called as local or internal variables. The auto keyword can be used to define automatic variables.

```
/* Example for automatic variables*/
#include<stdio.h>
void f1(void);
void f2(void);
void main()
{
int x=7;          /* automatic or local variable*/
clrscr();
f1();
f2();
printf("\nx=%d",x);
getch();
}

void f1(void)
{
int x=17;         /*local variable*/
printf("\nx=%d",x);
}

void f2(void)
{
int x=177;        /* local variable*/
printf("\nx=%d",x);
}
```

```
o/p:
x=17
x=177
x=7
```

2. global (public) variables:

These are defined outside the functions and their lifetime is entire C program including main() and other user defined functions. These are used to share data between many functions in a c program. These are also called as global or public or external variables.

Example1:

```
int x=0;          /* global variable*/
void f1(void)
void main()
{
x++;
printf("\n x=%d",x);
f1()
printf("\n x=%d",x);
getch();
}
void f1(void)
{
x++;
}
```

```
o/p:
x=1
x=2
```

3. static variables:

These are defined inside the function with the keyword static. These are visible with the function in which they are defined and entire c program. Once, these variables become active then entire c program they go on updating.

Example:

```
void test(void);
void main()
{
int i;
        for(i=1;i<=3;i++)
        {
        test();
        }
getch();
}
void test(void)
static int x=0;  /* static variable*/
x++;
printf("\n x=%d",x);
}
```

```
o/p:
x=1
x=2
x=3
```

4. register variables:

These are defined inside or outside the function by using a keyword register. These variables will occupy space from CPU's register instead of computer's primary memory RAM. These variables used to store frequently required data by processors, so that the data stored in register can fetched in less time with higher speed.

Example:

```
void main()
{
register int i, count=0;  /* register variables*/
clrscr();
for(i=1;i<=100;i++)
{
count++;
}
getch();
}
```

…