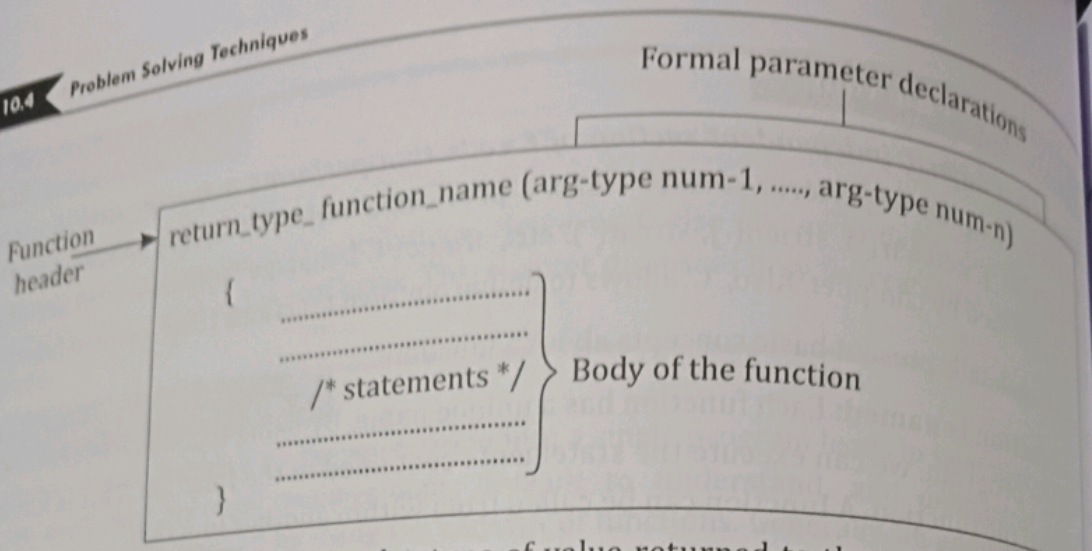
Unit - 2

Chapter 10

Functions

Chapter Outline

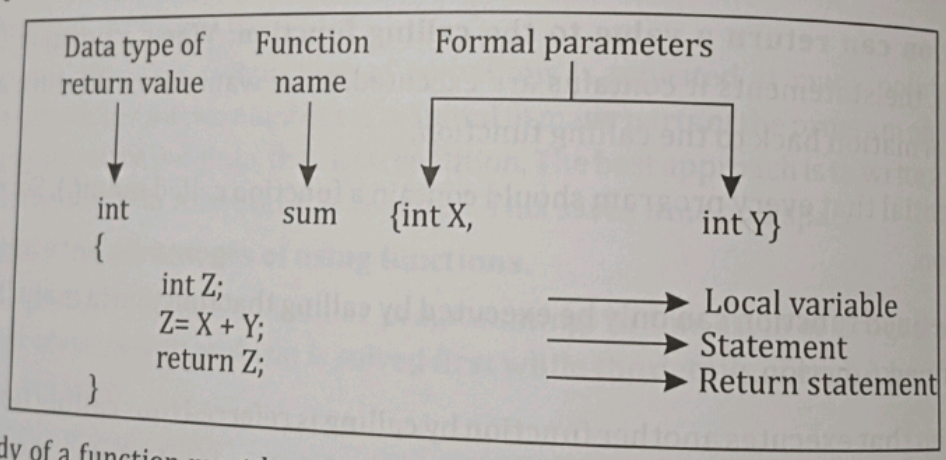
- Introduction
- Need for Functions
- What is a Function?
- Function Definition
- Function Prototyping
- Types of Functions
  - Library Functions
  - User Defined Functions
- Categories of Functions
  - Function with no arguments and no returning value
  - Function with arguments but no returning value
  - Function with no arguments and returning a value
  - Function with arguments and returning value
- Function Call
  - Passing parameters to the called function
- Types of Arguments
- Variable Length Argument List
- Nesting of Functions
- Passing Arrays to Functions
  - Passing Individual Elements
  - Passing Entire Array to a Function
- Advantages of User Defined Functions
- Review Questions



The return-type specifies the datatype of value returned to the calling function. Naming a function is similar to naming a variable. The formal parameters receive values from the calling function. the formal parameter declaration can be an optional.

2. **Body of function:** The body of function may contain local variables, statements, or return statement. The actual task will be performed in the body of the function.

Example:



The body of a function must be enclosed in curly braces. This may include declarations as well as statements. A **return** statement in a function can be followed by an optional value which causes the function to return to where it was called from.

A function can contain multiple return statements. The first return statement is the only one that has any effect. The return statement can return only one value to the calling function. The default return type of the function is **int.** Therefore writing **int** in the function header is an optional.

10.5 Function Prototyping

The avoid ambiguity in data transfer to the called function and in returning a value from a called function to the calling function, ANSI (American National Standard Institute) lays a standard that the type of function and types of variables passed be declared either in the calling function (or) prior to it. This is done using function prototype statement and whose syntax is

```
return-type  function-name (format
```

Example: `float sum (float f1, float f2);`

The above **sum** function returns a float value and which takes two parameters, both of which are floats.

- The prototypes are usually placed just before the main function of the program.
- A function prototype need not have variable names. Instead it can contain only the data types of variables.

Example: `float sum(float,float)`

- The function header must match with prototype in terms of function name, return type, number of data types, and order of arguments.
- In some cases, if the function definition is placed before the main( ), then function prototyping not necessary. Otherwise; function prototype is necessary.

10.6 Types of Functions

There are basically two types of functions. They are

1. Library Functions
2. User Defined Functions

10.6.1 Library Functions

The functions that are predefined in the header file are called library functions. The library functions are compiled, tested, and then are included in the header file.

Example

`pow(), sin(), cos(), sqrt()` etc are mathematical build in functions (or) library functions. These are available in header file **math.h.**

Example

`strlen(), strcmp(), strcat(), srtcpy()` etc are string functions which are available in header file **string.h.**

Example

`printf(), scanf(), getchar(), putchar()` etc., are input/output functions available in header file **stdio.h.**

To use library functions (or) built-in-function in our program, we should include the corresponding header file at the top of a program using **#include** preprocessor directive. The following program uses mathematical library functions.

Program 1 To demonstrate the library functions

```
#include<stdio.h>
#include<math.h>
main( )
{
      int x, y, z;
      printf("\n Enter the value of X:");
      scanf("%d", &x);
      y = pow(2, x);
      z = sin(x);
      printf("\n y=%d", y);
      printf("\n z=%d", z);
}
```

Explanation:

In the above program, if the value of x is 4, then pow (2, 4) results in 16 and which is assigned to y. Similarly sin(4) value will be assigned to z and last two printf( ) statements prints the values of y and z respectively.

Note

To call any function, specify the function name, enclosing the arguments within parenthesis.

10.6.2 User Defined Functions

The functions that are defined by the user are called **user-defined functions.** User-defined functions can be created in the same way as we write function **main( ).** Similar to main( ) function, the other user defined functions can be created by specifying the function name, formal parameters (if any),and enclosing group of statements within curly braces as shown below.

```
void display( )
{
      ……..……..…
      ….……..……
      ……..……..…
}
```

Example 1

```
#include<stdio.h>
void display();                          ──► Function prototype
void main()
{
      display();
}
void display()                           ──► Function header.
{
      printf("\n Welcome to functions");
}
```

Explanation:

We know that execution of program always starts from **main( )**. When the statement **display( )** is encountered in main( ), the control transfers to the function **display( )** and the statement within the display( ) will be executed and it prints the message **"Welcome to functions".** Once the body of the function is executed, then the control is returned back to the function **main( ).**

In **main( )** function we have called display( ) function. The statement display( ) in main is referred to as **function call.** The **main( )** function is referred to as **calling function.** The **display( )** function is referred to as **called function.**

Example 2

```
#include<stdio.h>
int abs(int a);          // Function Prototype Declaration
void main()
{
    int x, y, z;
    x = 10;
    y = -20;
    z = abs(x);
    printf("\n The absolute value of x = %d", z);
    z = abs(y);
    printf("\n The absolute value of y = %d", y);
}
int abs(int a)
{
   if (a>0)
      return(a);
   else
      return(- a);
}
```

Explanation

In the above program, **abs( )** function is a user-defined function which performs a specific task. The call to the function is illustrated in the following statement.

```
z = abs(x);
```

The value of x is sent to the **abs** function which receives it in another variable called **'a'**. The abs function checks whether 'a' is greater than zero (or) not. If it is greater than zero, then it returns the same value to the main( ) and main( ) receives it in the variable 'z'; Otherwise, it returns the negative value of 'a' to the main( ).

Example 3

Consider the following code

```
void main()
{
    int i:
    for(i=1;i<=5; i++)
    {
        display();
    }
}

display()
{
    printf("Hello\n");
}
```

In function **main( )**, the statement **display( )** is called in the for loop. The function **display( )** is executed 5 times by displaying the message **"Hello".**

Example 4

Consider the following code.

```
void main( )
{
    MSc( );
}

MSc( )
{
    BSc( );
}

BSc( )
{
    PUC( );
}

PUC( )
{
    printf("Hello");
}
```

The main( ) function calls MSc( ) function; the MSc( ) calls BSc( ); The BSc( ) calls PUC( ); Finally PUC() function prints the message "Hello" and control is transferred back to BSc( ) and BSc( ) to MSc( ) and MSc( ) to main() function; and finally program terminates.

Example 5

Consider the following code

```
void main()
{
    printf("What is your name?\n");
    find_name();
    printf("Good name!\n");
}
find_name()
{
    printf("My name is SNIGDHA\n");
}
```

Output

```
What is your name?
My name is SNIGDHA
Good name!
```

Explanation

As we know program execution begins with main(). The first printf() statement in main() is executed first by printing the message is "What is your name?", and then it calls the function called **find_name( ).** The find_name( ) prints the message "My name is SNIGDHA" and returns the control to the main( ). Then main() executes the last printf() function and it prints "Good name"! and program terminates.

10.7 Categories of Functions

We have seen that a function receives several values from the calling function, and after executing the body of the function, it returns a value to the calling function. But depending on the programming situations, receiving values (or) returning value may be missing or it may have both. Depending on whether arguments are present or not and whether a value is returned or not, functions can be classified as

1. Function with no arguments and no returning value.
2. Function with arguments but no returning value.
3. Function with no arguments and returning value.
4. Function with arguments and returning value.

10.7.1 Function with no arguments and no returning value

In this case, function does not contain any formal parameters and it does not return any value to the calling function. Any function which does not return any value is nothing but **void functions.**

Example

```
#include<stdio.h>
void sum();
main()
{
   sum( );
}
No arguments

void sum()
{
        int X = 10, Y = 20, Z;
        Z = X + Y;
        printf("%d", Z);
}
No return value
```

The sum( ) function does not receive any values from main( ). And also, it does not return any value to the main( ).

10.7.2 Function with arguments but no returning value

This is nothing but one-way communication between calling function and called function. That is, called function receives the values from the calling function but it does not send any value back to the calling function.

Example

```
#include<stdio.h>
void sum(int a, int b);
main()
{
    int x = 10, y = 20
    sum(x,  y);
}
Function with arguments

void sum(int a, int b)
   {
       int c
       c = a + b;
       printf("sum = %d",  c);
   }
No return value
```

Here, sum( ) function is called in main( ) with two arguments x and y. The values of x and y receive in sum( ) function which are stored in a and b respectively. The sum( ) performs addition of a and b and result will be stored in c. The value of c is printed in sum( ) function itself.

10.7.3 Function with no arguments and returning a value

In this case, calling function does not send any values to the called function but called function sends the value to the calling function. This is also one-way communication between calling function and called function.

```
#include<stdio.h>
int sum();
main()
 {
   int z;
No arguments
   z = sum( );
   printf("sum = %d", z);
 }

int sum()
{
    int a = 10, b = 20, c;
    c = a + b;
    return(c);
}
        Return value
```

In the above example, sum( ) function is called in main( ) without any arguments. The sum function performs addition of a and b, and stores the result in c. The value of c is returned back to the main( ) and it is stored in variable z. The value of z (i.e., 30) is printed in the main( ) function.

10.7.4 Function with arguments and returning value

This code is considered as two-way communication between calling the called functions. The calling function sends the values to the called function and in turn called function also returns the value to the calling function.

```
#include<stdio.h>
int sum(int a, int b);
main( )
{
   int X = 10, Y = 20, Z;
With arguments
   z = sum(X, Y );
   printf("sum = %d", Z);   }
}

int sum(int a, int b)
{
    int c
    c = a + b;
    return(c);
                  Return value
```

Here, the main() function calls the sum() function by sending the values of X and Y and the sum( ) receives it with a and b respectively. The addition of a and b is performed in sum() and result c is returned back to the main( ) function. The main( ) function receives the returned value in Z. The value Z (i.e., 30) is printed in the main( ).

10.8 Function Call

We know that a function call has to be present in the function, for it to call another function. A function call consists of name of called function followed by a list of actual arguments in a pair of parenthesis. The syntax of function call is

```
Function-name (actual arguments list);
```

Example

```
main( )
{
    int x = 10, y = 20;
    sum(x, y);          <-- Function call
}
```

- sum(x, y) is a function call which supplies values to called function and do not receive a value from the called function.
- Z = sum(x, y)
  is used to call a function that receive values and returns a value. The return value will b
- sum( );
  is used to call a function which neither receives values nor return a value.
- Z = sum( );
  is used to call a function which does not receive any values but returns a value.

10.8.1 Passing parameters to the called function

The arguments present in the function call is called as **actual arguments.** There are two ways in which actual arguments are passed to the called functions.

1. Call by value
2. Call by reference

1. **Call by value:** In call by value, the actual parameters will be in the form of variables, constants, or expressions. In this, only the values of actual arguments of the function call are transferred to the called function. That is, it passes the values of the arguments to the function.

Example1

```
sum(X, Y);
sum(10, 20);
```

Any modification made to the received values in the called function can only be seen in that function and these modifications can not seen in the calling function.

2. **Call by reference:** Call by reference involves passing memory addresses of the actual arguments to the called function rather than passing their values. In this case, modification of received values in the called function can be seen in the calling function.

Example 2

```
sum(&X, &Y);
sum(X, &Y);
```

10.9 Types of Arguments

There are basically two types of arguments. They are

1. Actual arguments/parameters
2. Formal arguments/parameters.

1. **Actual arguments:** The arguments associated with a function call is referred to as actual arguments. The arguments that are sent from a function call to the called function are called **actual arguments.**

```
main()
{
      int X = 10, Y = 20;
                                        /*X and Y are actual arguments*/
      sum(X,  Y);
          |
          |
          v
      Actual arguments
}
```

2. **Formal arguments:** The arguments associated with called functions are referred to as formal arguments/parameters. That is, the arguments specified in the function header is nothing but a formal arguments. The arguments that can have the copy or the reference of actual arguments are called **formal arguments.**

- Any number of arguments can be passed to a function being called. However, the type, order, and number of the actual and formal parameters must be same.

```
                Formal arguments
                    |       |
void sum(int a,         int b)
{
      int c;
      c = a + b;
      printf("sum = %d", c);
}
```

- Formal arguments (or) parameters can also declared as shown below.

```
void sum(a, b)
int a, b;
{
      int c;
      c = a + b;
      printf("sum = %d", c);
}
```

- Any number of arguments can be passed to a function being called. However, the type, order, and number of the actual and formal parameter must be same.

10.10 Variable Length Argument List

We learned to define our own functions, passing arguments to a function, returning value from a function. In this section, we will discuss something interesting about passing variable length arguments to a function.

We know that a function usually takes a fixed number of arguments and their data type is also declared in function declaration. In such scenarios, we are aware of the number of arguments required. Sometimes, we may come across a situation, when we want to have a function, which can take variable number of arguments instead of predefined number of arguments. The C programming language provides a solution for this situation and we are allowed to define a function which can accept variable number of parameters based on our requirement.

The most well known example would be printf and scanf functions. We have seen that functions like printf() and scanf() accept any number of arguments passed.

Example

```
printf("Programming in C");                    // Single argument
printf("Max number is %d", max);               // Two arguments
printf("Date is %d-%d-%d", 03, 02, 2022);      // Four arguments
```

The printf or scanf function works for n number of arguments, but how it works?

Lets us consider that we want to write a function to find maximum number from set of numbers. We will end up with function declaration as shown in the below example.

Example

```
int maximum(int n1, int n2, int n3);   // Find maximum between three numbers
int maximum(int n1, int n2, int n3, int n4, int n5); // Find maximum between five
numbers
```

However, none of the above declaration is suitable for the case. First function will find maximum of three numbers similarly second will find maximum of five numbers. What if we need to find maximum of 20 numbers. For such scenario, we use variable length arguments in a function.

Variable length arguments is a programming construct that allows programmers to pass n number of arguments to a function. We can also call variable length argument as **var-args**.

Syntax

```
return_type function_name(int num, ...);
```

- `return_type`: Specify function return type.
- `function_name` : Name of the function.
- `int num` : int num specify number of variable length arguments passed.
- `Ellipsis symbol ...` : Specify that the function is ready to accept n number of arguments.

Example

```
int max(int num, ... )
{
   .
   .
   .
}
int main()
{
   max(3, 1, 2, 3);
   max(6, 1, 2, 3, 4, 5, 6);
   max(10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
}
```

It should be noted that the function max() has its last argument as ellipses, i.e. three dotes (...) and the one just before the ellipses is always an int which will represent the total number variable arguments passed.

Macros used in Variable Length Arguments:

To use variable length arguments functionality, we need to make use of **stdarg.h** header file which provides the functions and macros to implement the functionality of variable arguments.To implement var-args in our program we will use following macros definitions.

- va_list - Data type to define a va_list type variable.
- va_start - Used to initialize va_list type variable.
- va_arg - Retrieves next value from the va_list type variable.
- va_end - Release memory assigned to a va_list variable.

Steps to implement variable length arguments in a function

Step 1: Include **stdarg.h** header file to access variable length argument.

Step 2: Define a function to accept variable length arguments.

```
void myfunction(int num, ...);
```

Step 3: Inside function create a **va_list** type variable.

```
va_list list;
```

Step 4: Initialize **list** variable using **va_start** macro. va_start macro accepts two parameters. First va_ list type variable and number of arguments in the list.

```
va_start(list, num);
```

Step 5: Run a loop from 1 to number of arguments. Inside the loop, use **va_arg** to get next argument passed as variable length arguments. **va_arg** accepts two parameter. First va_list type variable from where to fetch values. Second, type (data type) of value we want to retrieve.

```
va_arg(list, int);
```

Step 6 : Finally release memory occupied by va_list using va_end.

```
va_end(list);
```

Program 2 **Write a C Program to accept n number of arguments using variable length arguments. Return maximum number of all values.**

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

/* Variable length arguments function declaration */
int maximum(int num, ...);

void main()
{
  printf("Max(10,20) = %d\n", maximum(2, 10, 20));
  printf("Max(89,35,99,26) = %d\n", maximum(4, 89, 35, 99, 26));
  printf("Max(10,20,30,40,50,60,70) = %d\n", maximum(7, 10, 20, 30, 40, 50, 60, 70));
  printf("Max(25,15,35,65,75,95,45,05,85,55)= %d\n",maximum(10,25,15,35,65,75,95,45,05,85, 55));
}
int maximum(int num, ...)
{
  int count, value, max;

  /* Declare va_list type variable */
  va_list list;

  /* Initialize the list */
  va_start(list, num);
  /* First element is set as maximum */
  max= va_arg(list, int);
  /* Run loop from 2 to number of arguments passed */
  for(count=2; count<=num; count++)
  {
    /* Get next argument in list */
    value = va_arg(list, int);

    /* If current argument is greater than max then store it in max */
    if(value > max)
       max = value;
  }

  /* Clean the list */
  va_end(list);

  /* Finally return max argument in list */
  return max;
}
```

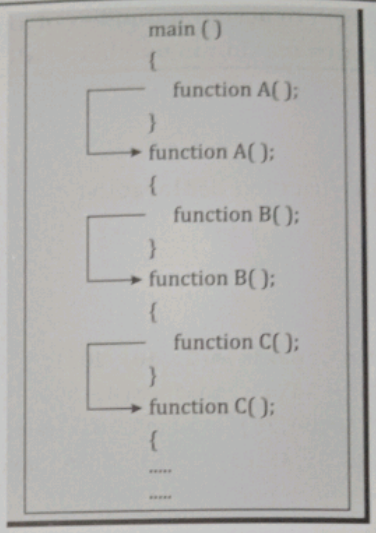
Output

```
Max(10,20) = 20
Max(89,35,99,26) = 99
Max(10,20,30,40,50,60,70) = 70
Max(25,15,35,65,75,95,45,05,85,55) = 95
```

10.11 Nesting of Functions

C allows nesting of functions freely. Main function can call functionA(), which can call functionB(), which can call functionC(), and so on. There is no limit on how deeply functions can be nested.

Example



Important Points to Know

- C program may contain one (or) move functions. But, there should be at least one function called main ( ) from where the execution starts.
- A library function (or) user-defined function can be called any number of times by any function.
- By default, return type of any function is int.

Example

```
int sum()
{
        int a = 10, b = 20, c;
        c = a + b;
        return(c);
}
```

is same as

```
sum()
{
        int a = 10, b = 20, c;
        c = a + b
        return(c);
}
```

- There is no limit on the number of functions that might be present in a C program.
- The order in which the functions are defined and the order in which they are executed need not be the same.
- A function can be called from other function, but a function cannot be defined in another function.
- The return statement can be used in function to return the value to the calling function. The general form of return statement is

```
return expression;
```

The **expression** is optional. If the expression is absent, the return statement transfer the control to the point from where the function was called.

- A function can return only one value. Thus we should not specify two values to them.

Example 1

```
return;
return(10);
return(a);
```
Valid return statements

```
return(3.14);
return(10.20);
return(a,b);
return(a,b,c);
```
Invalid return statements

- There may be any number of return statements in function, but only one return statements will activate.
- There should be one to one correspondence between the actual and formal arguments in **type, order** and **number.**
- C allows recursion. That is a function can call itself.

Example 2

```
display( )
{
        for(i=0; i<3; i++)
        {
            display();
        }
}
```

- When a function is not returning any value, **void** type can be used as return type.

Example 3

```
void display( )
{
      printf("Hello");
}
```

- The variables declared inside the function are called **local variables.** The variables declared outside the function are called **global variables.** The global variables can be accessible any where in a program, where as local variables can be accessible only within the function where it is declared.

Example 4

```
int a = 10, b = 20;                          → global variables
main( )
{
        int x = 30,  y = 40;                 → local variables
        printf("%d%d%d%d", a,b,x,y);
        display (x,  y);
                    → Actual arguments
                    → Formal arguments
}
display(int p,  int q)
{                                            → local variable
        int z;
        z = p+q;
        printf("%d%d%d", x,y,z);
}
```

10.12 Passing Arrays to Functions

We can pass individual elements of an array (or) an entire array containing all its elements to a function. The following are the details regarding passing an array to a function.

10.12.1 Passing Individual Elements

Selected individual elements of an array may be passed to a function by specifying name of array followed by the subscript value of element in the function call. Consider the following code.

Program 3

```
#include <stdio.h>
void main()
{
      int result;
      int a[5] = {10,20,30,40,50};
      result = prod(a[1], a[2], a[4]);
      printf("Product = %d", result);
}
prod (int a, int b, int c)
{
      return (a * b * c);
}
```

Output

```
Product = 30000
```

Explanation:

In the above code, we have passed the elements a[1], a[2], and a[4] to the function prod( ); where the corresponding formal parameters are a,b and c respectively. Therefore a, b, and c will contain the values 20, 30 and 50 respectively. The product of 20 × 30 × 50 are evaluated and returned to the main( ). The main ( ) receives it in the variable **result** and which gets printed in the last printf( ) statement.

Note

When we are passing an individual elements to a function, the formal parameters in function header must be simple variables like a, b and c. If array elements are of type integer, then formal parameters must be simple integer variables. If array elements are of type float, then formal parameters must be simple float variables.

10.12.2 Passing Entire Array to a Function

Entire array can be passed to a function by specifying the name of array without subscript in the function call. Then the corresponding formal parameters in the function header of the called function should have the name of the array followed by an empty pair of brackets. It is useful that the size of array also be passed to the called function in order to facilitate operations on the array as shown below.

```
int find_sum(int a[],int n)
{
    int i, sum = 0
    for (i = 0; i<n; i++)
    {
       sum = sum + a[i];
    }
    return sum;
}
```

Output:

```
sum = 150
```

Program 5 Write a program to find the largest number among the three given numbers.

```
#include<stdio.h>
#include<conio.h>
main()
{
    float a, b, c;
    void large (float, float, float); /* function prototype */
    printf("\n Input the values of a, b and c:");
    large (a, b, c);
}
void large(float a, float b, float c)
{
    float big;
    big = a;
    if(b > big) big = b;
    if(c > big) big = c;
    printf("\n Largest number = %f", big);
}
```

Output

```
Input the values of a, b and c : 20 10 15
Largest number = 20
```

10.13 Advantages of User Defined Functions

Advantages of User Defined Functions

- **Reduction in Program Size:** A sequence of statements which are repeatedly used in a program can be combined together to form a user defined functions. And this functions can be called as many times as required. This avoids writing of same code again and again reducing program size..
- **Reducing Complexity of Program**: Complex program can be decomposed into small sub-programs or user defined functions.
- **Easy to Debug and Maintain :** During debugging it is very easy to locate and isolate faulty functions. It is also easy to maintain program.
- **Readability of Program**: By using user defined function, a complex problem is divided in to different sub-programs with clear objective and hence it makes easy to understand the logic behind the program.
- **Code Reusability:** Once user defined function is implemented it can be called or used as many times as required which reduces code repeatability and increases code reusability..

10.14 Review Questions

1. Define a function?
2. Write any two important advantages of using functions.
3. What do you mean by calling function?
4. What do you mean by called function?
5. What are the two parts of function definition?
6. What are the function header consists of?
7. What are the different types of functions?
8. Give an examples for library functions.
9. What is library function?
10. What is user defined function?
11. What are the different categories of functions?
12. What do you mean by actual arguments and formal arguments?
13. What are the different ways of calling a function?
14. What is call by value?
15. What is call by reference?
16. Explain function definition with an example.
17. Explain function prototype.
18. Explain call by value and call by reference with an example.
19. Explain the difference types of functions.
20. Differentiate between actual and formal arguments.
21. Explain the categories of functions.
22. Explain passing arrays to functions.
23. What is nesting of function?
24. What are advantages of using functions?
25. Why do we need functions?
26. Explain passing array as an argument to the function.
27. Explain pointers as function parameters with an example.
28. Explain the difference between call by value and call be reference.
29. What is the use of variable length arguments in a function?
30. Which header file must be included to use the features of variable length arguments?
31. What are the steps to be followed to implement variable length arguments in a function?
32. Write the general syntax of a function with variable length arguments.
33. Write a program to find the minimum and maximum number from a set of elements using variable length arguments in a function.
34. Explain variable length arguments with an example.