UNIT V

User defined functions

Functions are blocks of codes to perform specific tasks and return the result. However it is not mandatory for a function return anything and also a function is not limited to performing one task only. User defined functions are basic building blocks of a program and can be found in the basic structure of C program.

Functions can be classified into two categories, namely, library functions and user-defined functions. The functions which are developed by user at the time of writing a program are called user defined functions. Thus, user defined functions are functions developed by user.

Advantages of User Defined Functions

When not using user defined functions, for a large program the tasks of debugging, compiling etc may become difficult in general. That's why user defined functions are extremely necessary for complex programs. The necessities or advantages are as follows,

1. It facilitates top-down modular programming. In this programming style, the high level logic of the overall problem is solved first while the details of each lower-level function are addressed later.
2. The length of a source program can be reduced by using functions at appropriate places.
3. It is easy to locate and isolate a faulty function for further investigations.
4. A function may be used by many other programs. This means that a programmer can build on what others have already done, instead of starting all over again from scratch.

Multifunction program : A function is a self-contained block of code that performs a particular task. Once a function has been designed and packed, it can be treated as a 'black box' that takes some data from the main program and returns a value. Thus a program, which has been written using a number of functions, is treated as a multi-function program.

Elements of User Defined Functions

In order to make use of an user defined function, we need to establish three elements, which are as follows

1. Declaration
2. Definition
3. Call
4.

Function Declaration

Declaration of function is simply declaring the name of the function, the arguments and their types and the return type of the function. User needs to declare a function prior to the definition of the main() function when the definition of the function is written after the definition of the main() function. If the user writes the definition of the function prior to the definition of the main() function then it is not needed to declare the function explicitly.

```
//declaring an user defined function which has been defined later

floataddNumbers(float a, float b);
```

Function Definition

Function definition includes the parts of the function declaration along with the body or code-block for the function. User can define a function before or after the main() function.

```
//defining an user defined function which has been declared earlier.

floataddNumbers(float a, float b){

return (a+b);

    }
```

Function Call

Calling an user defined function is similar to calling a library function, write the name of the function and provide the arguments. Also, if you need to store the returned data in a variable, then assign this call to a variable.

```
int main(){

float result;
```

```
   /* calling user defined function from the main function */

result = addNumbers(.5, .8);



return 0;

   }
```

In this part of the example, we are calling the addNumbers() function and assigning it's returned value to a variable result.

C - Functions

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.

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 C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.

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

Defining a Function

The general form of a function definition in C programming language is as follows −

```
return_typefunction_name( parameter list ){
bodyof the function
}
```

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function −

- **Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** − The function body contains a collection of statements that define what the function does.

Example

Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value between the two −

```
/* function returning the max between two numbers */
int max(int num1,int num2){

/* local variable declaration */
int result;

if(num1 > num2)
result= num1;
else
result= num2;

return result;
}
```

Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

return_typefunction_name(parameter list);

For the above defined function max(), the function declaration is as follows −

int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration −

int max(int, int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.

For example – C

```
#include<stdio.h>

/* function declaration */
int max(int num1,int num2);

int main (){

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

/* calling a function to get max value */
ret= max(a, b);

printf("Max value is : %d\n", ret );

return0;
}

/* function returning the max between two numbers */
int max(int num1,int num2){

/* local variable declaration */
int result;

if(num1 > num2)
result= num1;
else
result= num2;

return result;
}
```

We have kept max() along with main() and compiled the source code. While running the final executable, it would produce the following result −

Max value is : 200

Function Arguments

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 −

Sr.No.	Call Type & Description
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.
2	Call by reference This method 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. This means that changes made to the parameter affect the argument.

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.

Function call by Value in C

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.

```
/* 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;
}
```

Now, let us call the function **swap()** by passing actual values 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 */
swap(a, b);

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

return0;
}
```

Let us put the above code in a single C file, compile and execute it, it will produce the following result −

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

It shows that there are no changes in the values, though they had been changed inside the function

Function call by reference in C

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.

```
/* 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 aie. 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 );

return0;
}
```

Let us put the above code in a single C file, compile and execute it, to produce the following result −

```
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

Types of User-defined Functions in C

There can be 4 different types of user-defined functions, they are:

1. Function with no arguments and no return value
2. Function with no arguments and a return value
3. Function with arguments and no return value

4. Function with arguments and a return value

Function with no arguments and no return value

Such functions can either be used to display information or they are completely dependent on user inputs.

Below is an example of a function, which takes 2 numbers as input from user, and display which is the greater number.

```
#include<stdio.h>

voidgreatNum();      // function declaration

int main()
{
greatNum();       // function call
return 0;


voidgreatNum()       // function definition

inti, j;
printf("Enter 2 numbers that you want to compare...");
scanf("%d%d", &i, &j);
if(i> j) {
printf("The greater number is: %d", i);
   }
else {
printf("The greater number is: %d", j);
   }
}
```

Function with no arguments and a return value

We have modified the above example to make the function `greatNum()` return the number which is greater amongst the 2 input numbers.

```
#include<stdio.h>


intgreatNum();      // function declaration


intmain()

{

int result;

result = greatNum();       // function call

printf("The greater number is: %d", result);

return0;

}


intgreatNum()       // function definition

{

inti, j, greaterNum;

printf("Enter 2 numbers that you want to compare...");

scanf("%d%d", &i, &j);

if(i> j) {

greaterNum = i;

    }

else {

greaterNum = j;

    }

// returning the result

returngreaterNum;
```

```
}
```

Function with arguments and no return value

We are using the same function as example again and again, to demonstrate that to solve a problem there can be many different ways.

This time, we have modified the above example to make the function greatNum() take two int values as arguments, but it will not be returning anything.

```
#include<stdio.h>


voidgreatNum(int a, int b);      // function declaration


intmain()
{
inti, j;
printf("Enter 2 numbers that you want to compare...");
scanf("%d%d", &i, &j);
greatNum(i, j);       // function call
return0;
}


voidgreatNum(int x, int y)       // function definition
{
if(x > y) {
printf("The greater number is: %d", x);
   }
else {
printf("The greater number is: %d", y);
   }
```

```
}
```

Function with arguments and a return value

This is the best type, as this makes the function completely independent of inputs and outputs, and only the logic is defined inside the function body.

```
#include<stdio.h>


intgreatNum(int a, int b);       // function declaration


intmain()

{

inti, j, result;

printf("Enter 2 numbers that you want to compare...");

scanf("%d%d", &i, &j);

result = greatNum(i, j); // function call

printf("The greater number is: %d", result);

return0;

}


intgreatNum(int x, int y)       // function definition

{

if(x > y) {

return x;

   }

else {

return y;

   }

}
```

Nesting of Functions

C language also allows nesting of functions i.e to use/call one function inside another function's body. We must be careful while using nested functions, because it may lead to infinite nesting.

```
function1()
{
// function1 body here

function2();

// function1 body here
}
```

If function2() also has a call for function1() inside it, then in that case, it will lead to an infinite nesting. They will keep calling each other and the program will never terminate.

Not able to understand? Lets consider that inside the main() function, function1() is called and its execution starts, then inside function1(), we have a call for function2(), so the control of program will go to the function2(). But as function2() also has a call to function1() in its body, it will call function1(), which will again call function2(), and this will go on for infinite times, until you forcefully exit from program execution.

Recursion

Recursion is a special way of nesting functions, where a function calls itself inside it. We must have certain conditions in the function to break out of the recursion, otherwise recursion will occur infinite times.

```
function1()
{
// function1 body
```

```
function1();
// function1 body
}
```

Example: Factorial of a number using Recursion

```
#include<stdio.h>

intfactorial(int x);       //declaring the function

voidmain()
{
int a, b;

printf("Enter a number...");
scanf("%d", &a);
    b = factorial(a);       //calling the function named factorial
printf("%d", b);
}

intfactorial(int x) //defining the function
{
int r = 1;
if(x == 1)
return1;
else
        r = x*factorial(x-1);       //recursion, since the function calls itself
```

```
return r;
}
```

FUNCTIONS;

What is a Function

Functions allow the programmer to modularize a program. All variables declared in function definitions are local variables – they are known only in the function in which they are defined. Most functions have a list of parameters. The parameters provide the means for communicating information between functions. A function parameters are also local variables.

Example1:

sqrt(x), exp(x) **log(x),** **fabs(x),**

Example 2:

ceil(x) **ceil(9.2) is 10.0 ceil(-9.8) is -9.0**

floor (x) **floor(9.2) is 9.0 floor(-9.8) is -10.0**

FUNCTIONS DEFINITION

The format of a function definition is

```
return-value-type     function-name(parameter-list)
{
        declarations
        statements

}
```

The function-name is any valid identifier.

The return-value-type is the data type of the result returned to the caller.

The return-value –type void indicates that a function does not return a value.

```
/* Example for a function  which  square's the value */
#include "stdio.h"
int square (int);
main ()
{
        int x;
        for ( x= 1; x <= 10 ; x++)
        printf("%d", square(x));
```

```
        printf("\n");
        return 0;
}
        /* Function definition */
int square(int y)
{
        return y*y;
}
```

Function Prototypes

A function prototype tells the compiler the type of data returned by the function, the number of parameters the function expects to receive, the types of the parameters and the order in which these parameters are expected. The compiler uses function prototypes to validate function calls.

```
/* Finding the maximum of three integers */
#include "stdio.h"
int maximum(int, int, int);        /* function prototype */
main ()
{
        int a, b, c;
        printf ( " Enter three integers:");
```

```
        scanf("%d %d %d, &a, &b, &c);

        printf("maximum is : %d \n", maximum ( a,b,c);

        return 0;

}

/* Function maximum definition */

int maximum (int x, int y, int z)

{

        int max = x;

        if ( y > max )

        max = y;

        if (z > max )

        max = z;

        return max;

}
```

RANDOM NUMBER GENERATING FUNCTION

To demonstrate rand function let us develop a program to simulate 20 rolls of a six-sided die and print the value of each roll. The function prototype for the rand function can be found in <stdlib.h>. The Modulus operator (%) in conjunction with rand as …. rand ( ) % 6 to produce integers in the range 0 to 5. This is called scaling. The number 6 is called the scaling factor. We then shift range of numbers produced by adding 1 to our previous result.

```
/ * Shifted, scaled integers produced by 1 + rand ( ) % 6  */
```

```
# include “stdio.h”

# include “stdlib.h”

main ( )

{

        int    i ;

        for ( i = 1; i < = 20; i++)

        {

                printf ( “ %10d”,   1 + (rand ( ) % 6 ) ) ;

                if ( i % 5 == 0 )


                printf ( “ \n “ ) ;

        }

        return 0;

}
```

```
/* Program to Roll a six – sided die 6000 times */


# include < stdio.h >

# include < stdlib.h>

main ()

{

        int face, roll ;
```

```
int freq1 = 0, freq2 = 0, freq3 = 0, freq4 = 0, freq5 = 0, freq6 = 6;

for ( roll = 1; roll < = 6000 ; roll++)

{

        face = 1 + rand () %6 ;


        switch ( face )

        {

                case 1:

                        ++ freq1;

                        break ;

                case 2:

                        ++ freq2;

                        break ;

                case 3:

                        ++ freq3;

                        break ;

                case 4:

                        ++ freq4;

                        break ;

                case 5:

                        ++ freq1;

                        break ;

                case 6:
```

```
                                ++ freq6;

                                break ;

                        }

                }

printf ( " % s % 13 s \n ",  " Face ",  " Frequency " ) ;

printf ( " 1 % 13 d \n ", freq1 ) ;

printf ( " 2 % 13 d \n ", freq2 ) ;

printf ( " 3 % 13 d \n ", freq3 ) ;

printf ( " 4 % 13 d \n ", freq4 ) ;

printf ( " 5 % 13 d \n ", freq5 ) ;

printf ( " 6 % 13 d \n ", freq6 ) ;

return 0;


}


/ * Randomizing die – rolling program * /


# include < stdlib.h >


# include < stdio.h >


main ( )
```

```
{
        int i;
        unsigned seed;
        printf ( " Enter seed" ) ;
        scanf ( " %u", &seed );
        srand (seed ) ;

        for ( i = 1; i < = 10 ; i ++ )

        {
                printf ( " %10d", 1 + (rand ( ) % 6 )) ;

                if ( i % 5 == 0 )

                        printf ( " \n " ) ;
        }
        return 0;
}

/ * Recursive factorial function * /

# include " stdio.h"
```

```
long factorial ( long ) ;


main ( )


{
        int i ;


        for ( i = 0 ; i < = 10 ; i ++ )


                printf ( " % 2d ! = % d \n ", i, factorial ( i ) ) ;
        return 0;
}


                                        /* Recursive definition of function factorial * /
long factorial ( long number )
{
        if ( number < = 1 )
        return 1 ;
        else
        return ( number * factorial ( number – 1 )) ;


}
```

Unit –V (Functions)

EXPLAIN CATEGORY OF FUNCTIONS.

There are four types of functions.

1. *Function with no arguments and no return value.*
2. *Function with no arguments and a return value.*
3. *Function with arguments and no return value.*
4. *Function with arguments and a return value.*

1.Function with no arguments and no return value:

- A function can have no arguments and no return value.
- Since there are no arguments, we cannot pass the data.
- During the execution of the program, data cannot be transferred from main function to the user-defined function.
- This type of function can do only a specific task.

2.Function with no arguments and a return value:

- When the function is called in the main function , the user-defined function may return a value.
- When the function is called , the control goes to the named user-defined function.
- When the user-defined function is executed, finally a value is returned.
- Eventhough there is no argument, communication between main function and user-defined function exists.
- Infact, signals exists between main function and user-defined function.

3.Function with arguments and no return value:

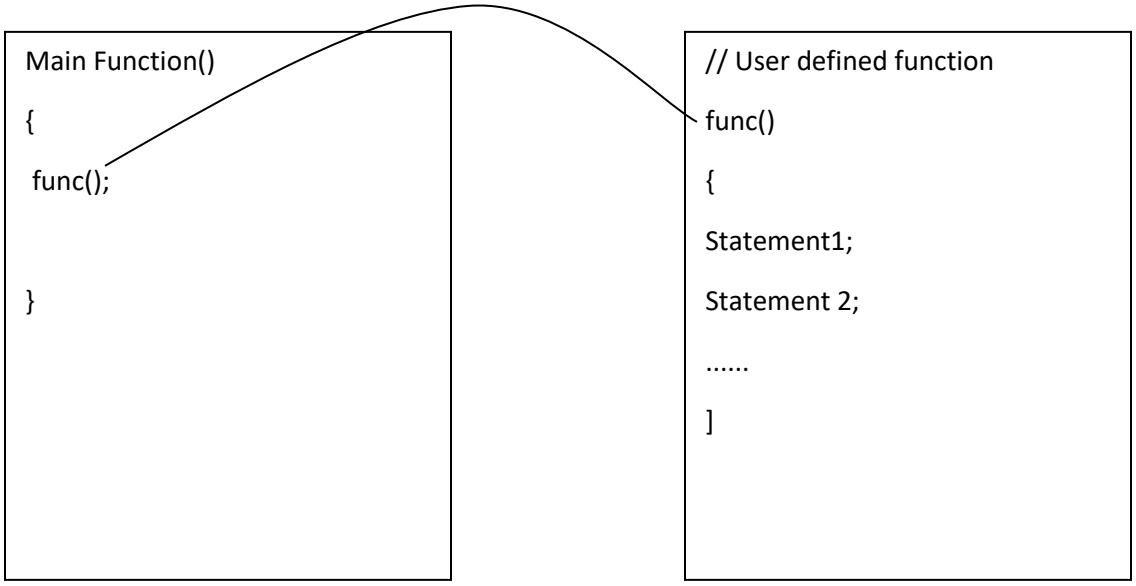
- When the function has arguments, the parameter value is passed to the user-defined function.
- The argument parameters may be a variable or pointer variable.
- A signal is sent from the main function to the user-defined function.

- When the parameter value is passed, then the user-defined function will be executed.
- During the execution of user-defined function, the function will do a specific task but it will not return any value.
- This type of function will not send any signal to the main function.

4. Function with arguments and a return value:

- A function can have any number of parameters in the arguments.
- Functions can have the same name for different functions.
- But it is identified only through the arguments and parameters.
- The data can be sent through the arguments from main function to the user-defined function.
- Signals are passed from the main function to the user-defined function and also the user-defined function to the main function.
- The returned value from the user-defined function can be used in the main function.

Figure 1: Function with no arguments and no return values.



/* Program to sort the data in ascending order using function with arguments but no return value */

```
void sort(int m, int x[ ]);
  main()
  {
     int i;
     int marks[5] = {40, 90, 73, 81, 35};
     printf("Marks before sorting\n");
     for(i = 0; i < 5; i++)
       printf("%d ", marks[i]);
     printf("\n\n");
     sort (5, marks);
     printf("Marks after sorting\n");
     for(i = 0; i < 5; i++)
```

```
      printf("%4d", marks[i]);
   printf("\n");
}
void sort(int m, int x[ ])
{
   int i, j, t;
   for(i = 1; i <= m-1; i++)
     for(j = 1; j <= m-i; j++)
       if(x[j-1] >= x[j])
       {
         t = x[j-1];
         x[j-1] = x[j];
         x[j] = t;
       }    }
```