Unit - 3

C Functions

In c, we can divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the C program. In other words, we can say that the collection of functions creates a program. The function is also known as *procedure* or *subroutine* in other programming languages.

Advantage of functions in C

There are the following advantages of C functions.

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call C functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- **Reusability is the main achievement of C functions.**
- However, Function calling is always a overhead in a C program.

Dr. Uma

Function Aspects

There are three aspects of a C function.

- **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.

- **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.

- **Function definition** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is

called. Here, we must notice that only one value can be returned from the function.

SN	C function aspects	Syntax
1	Function declaration	return_type function_name (argument list);
2	Function call	function_name (argument_list)
3	Function definition	return_type function_name (argument list) {function body;}

The syntax of creating function in c language is given below:

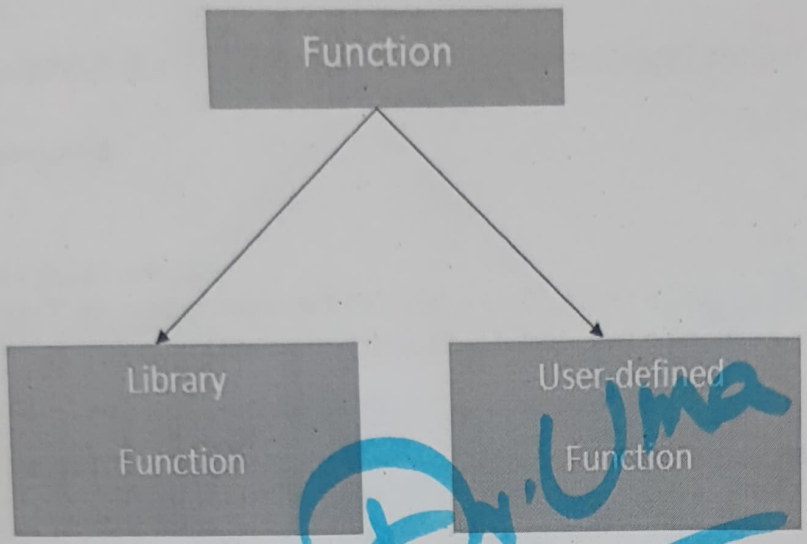
Dr. Uma

```
return_type function_name(data_type parameter...){
//code to be executed
}
```

Types of Functions

There are two types of functions in C programming:

1. **Library Functions**: are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.
2. **User-defined functions**: are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.



Return Value

A C function may or may not return a value from the function. If you don't have to return any value from the function, use void for the return type.

Let's see a simple example of C function that doesn't return any value from the function.

Example without return value:

void → no type, no value, no parameters ⇓ empty or blank.

#include <stdio.h>

```
1. void hello(){
2. printf("hello c");
3. }
```

void can be used as a data type that represents no data.

If you want to return any value from the function, you need to use any data type such as int, long, char, etc. The return type depends on the value to be returned from the function.

Let's see a simple example of C function that returns int value from the function.

Example with return value:

```
1. int get(){
2. return 10;
3. }
```

In the above example, we have to return 10 as a value, so the return type is int. If you want to return floating-point value (e.g., 10.2, 3.1, 54.5, etc), you need to use float as the return type of the method.

```
1. float get(){
2. return 10.2;
3. }
```

Now, you need to call the function, to get the value of the function.

Different aspects of function calling

A function may or may not accept any argument. It may or may not return any value. Based on these facts, There are four different aspects of function calls.

- function without arguments and without return value
- function without arguments and with return value
- function with arguments and without return value
- function with arguments and with return value

Example for Function without argument and return value

Example 1

```
1. #include<stdio.h>
2. void printName();
3. void main ()
4. {
5.     printf("Hello ");
6.     printName();
7. }
8. void printName()
9. {
10.    printf("SRMIST");
11.}
```

Output

```
Hello SRMIST
```

Example2

```
1. #include<stdio.h>
2. void sum();
3. void main()
4. {
5.     printf("\nGoing to calculate the sum of two numbers:");
6.     sum();
7. }
8. void sum()
9. {
10.    int a,b;
11.    printf("\nEnter two numbers");
12.    scanf("%d %d",&a,&b);
```

```
13.  printf("The sum is %d",a+b);
14.}
```

Output

Example for Function without argument and with return value

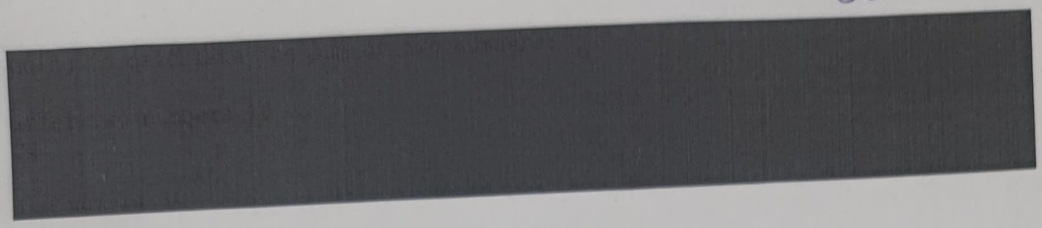
Example 1

```
1.  #include<stdio.h>
2.  int sum();
3.  void main()
4.  {
5.      int result;
6.      printf("\nGoing to calculate the sum of two numbers:");
7.      result = sum();
8.      printf("%d",result);
9.  }
10. int sum()
11. {
12.     int a,b;
13.     printf("\nEnter two numbers");
14.     scanf("%d %d",&a,&b);
15.     return a+b;
16. }
```

Output

Going to calculate the Sum of two nu-:

Enter two numbers 10
20
30



Example 2: program to calculate the area of the square

```
1. #include<stdio.h>
2. int sum();
3. void main()
4. {
5.     printf("Going to calculate the area of the square\n");
6.     float area = square();
7.     printf("The area of the square: %f\n",area);
8. }
9. int square()
10. {
11.     float side;
12.     printf("Enter the length of the side in meters: ");
13.     scanf("%f",&side);
14.     return side * side;
15. }
```

Dr. Um

Output

Example for Function with argument and without return value

Example 1

```
1. #include<stdio.h>
2. void sum(int, int);
3. void main()
4. {
5.     int a,b,result;
6.     printf("\nGoing to calculate the sum of two numbers:");
7.     printf("\nEnter two numbers:");
8.     scanf("%d %d",&a,&b);
9.     sum(a,b);
10. }
11. void sum(int a, int b)
12. {
13.     printf("\nThe sum is %d",a+b);
14. }
```

Output

Example 2: program to calculate the average of five numbers.

```
1. #include<stdio.h>
2. void average(int, int, int, int, int);
3. void main()
4. {
5.     int a,b,c,d,e;
6.     printf("\nGoing to calculate the average of five numbers:");
7.     printf("\nEnter five numbers:");
```

```
8.  scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);
9.  average(a,b,c,d,e);
10.}
11.void average(int a, int b, int c, int d, int e)
12.{
13.  float avg;
14.  avg = (a+b+c+d+e)/5;
15.  printf("The average of given five numbers : %f",avg);
16.}
```

Output

Example for Function with argument and with return value

Example 1

```
1. #include<stdio.h>
2. int sum(int, int);
3. void main()
4. {
5.    int a,b,result;
6.    printf("\nGoing to calculate the sum of two numbers:");
7.    printf("\nEnter two numbers:");
8.    scanf("%d %d",&a,&b);
9.    result = sum(a,b);
10.   printf("\nThe sum is : %d",result);
11.}
```

```
12. int sum(int a, int b)
13. {
14.     return a+b;
15. }
```

Output

Example 2: Program to check whether a number is even or odd

```
1.  #include<stdio.h>
2.  int even_odd(int);
3.  void main()
4.  {
5.   int n,flag=0;
6.   printf("\nGoing to check whether a number is even or odd");
7.   printf("\nEnter the number: ");
8.   scanf("%d",&n);
9.   flag = even_odd(n);
10.  if(flag == 0)
11.  {
12.     printf("\nThe number is odd");
13.  }
14.  else
15.  {
16.     printf("\nThe number is even");
17.  }
18. }
19. int even_odd(int n)
20. {
```

```
21.  if(n%2 == 0)
22.  {
23.     return 1;
24.  }
25.  else
26.  {
27.     return 0;
28.  }
29. }
```

Output

C Library Functions

Library functions are the inbuilt function in C that are grouped and placed at a common place called the library. Such functions are used to perform some specific operations. For example, printf is a library function used to print on the console. The library functions are created by the designers of compilers. All C standard library functions are defined inside the different header files saved with the extension **.h**. We need to include these header files in our program to make use of the library functions defined in such header files. For example, To use the library functions such as printf/scanf we need to include stdio.h in our program which is a header file that contains all the library functions regarding standard input/output.

The list of mostly used header files is given in the following table.

SN	Header file	Description
1	stdio.h	This is a standard input/output header file. It contains all the library functions regarding standard input/output.
2	conio.h	This is a console input/output header file.
3	string.h	It contains all string related library functions like gets(), puts(),etc.

4	stdlib.h	This header file contains all the general library functions like malloc(), calloc(), exit(), etc.
5	math.h	This header file contains all the math operations related functions like sqrt(), pow(), etc.
6	time.h	This header file contains all the time-related functions.
7	ctype.h	This header file contains all character handling functions.
8	stdarg.h	Variable argument functions are defined in this header file.
9	signal.h	All the signal handling functions are defined in this header file.
10	setjmp.h	This file contains all the jump functions.
11	locale.h	This file contains locale functions.
12	errno.h	This file contains error handling functions.
13	assert.h	This file contains diagnostics functions.

Unit-3

C library function - atoi()

The C library function **int atoi(const char *str)** converts the string argument **str** to an integer (type int).

Declaration

Following is the declaration for atoi() function.

```
int atoi(const char *str)
```

Parameters

- **str** − This is the string representation of an integral number.

Return Value

This function returns the converted integral number as an int value. If no valid conversion could be performed, it returns zero.

Dr. Uma

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main () {
  int val;
  char str[20];

  strcpy(str, "98993489");
  val = atoi(str);
  printf("String value = %s, Int value = %d\n", str, val);

  strcpy(str, "srmup.in");
  val = atoi(str);
  printf("String value = %s, Int value = %d\n", str, val);

  return(0);
}
```

Let us compile and run the above program that will produce the following result −

```
String value = 98993489, Int value = 98993489
String value =srmup.in, Int value = 0
```

sprintf() in C

sprintf stands for "string print". In C programming language

it is a file handling function that is used to send formatted output to the string. Instead of printing on console, sprintf() function stores the output on char buffer that is specified in sprintf.

Syntax

1. **int** sprintf(**char** *str, **const char** *format, ...)

Parameter values

The sprintf() function accepts some parameter values that are defined as follows -

str: It is the pointer to an array of char elements where the resulting string is stored. It is the buffer to put the data in.

format: It is C string

that is used to describe the output along with placeholders for the integer arguments to be inserted in the formatted string. It is said to the string that contains the text to be written to buffer. It consists of characters along with the optional format specifiers starting with %.

Example1 This is a simple example to demonstrate the use of sprintf() function in C. Here, we are using multiple arguments with the sprintf() function.

```
1. #include <stdio.h>
2. main()
3. {
4.     char buffer[50];
5.     int a = 15, b = 25, res;
6.     res = a + b;
7.     sprintf(buffer, "The Sum of %d and %d is %d", a, b, res);
8.     printf("%s", buffer);
9.     return 0;
10. }
```

How to read data using sscanf() in C

In C, `sscanf()` is used to read formatted data. It works much like scanf() but the data will be read from a *string* instead of the console.

'H' 'E' 'L' 'L' 'O' HELLO

How do we use this function?

In order to read formatted data from a string, first, we must know the syntax of the function, which is shown below.

Syntax

```
#include<stdio.h>

int main() {

   int sscanf(const char *read, const char *format, . . .,
              storage_variables )
```

This function can only be used when the stdio.h library is included

Now let's look at what parameters does this function take and what it returns.

Parameters

- `read`: This is the pointer to the string that is to be read.
- `storage_variables` Additional argument(s) that the function can take in order to store the value that is being read from the pointer. Here, if the value is being stored in a regular variable and not a pointer then the variable name must be preceded by the **&** sign.
- `format`: These are the **format specifiers** according to which the data will be read from the input string.

Format Specifiers

A **format specifier** is a unique set of characters which is used to format the data being read in a certain way. It is preceded by a % sign and then the relevant symbol. The symbols are as follows:

Symbol	Type
s	string
c	single character
d	decimal integer
e, E, f, g, G	floating points
u	unsigned integer
x, X	hexadecimal number

Return Value

1. The function returns an `int` value which represents the total number of items read.
2. Given there was an error and the string could not be read, an `EOF` error is returned.
3. Any other type of error which is encountered while reading is denoted by the function returning *-1*.

Now that we know about the syntax, let's look at a few examples to better understand `sscanf()`.

Examples

1. Reading one item of the same type

The first example shows you how to read just a single data type in an input string.

Dr. Ume

```
#include<stdio.h>
int main() {
    char* buffer = "Hello";
    char store_value[10];
    int total_read;
    total_read = sscanf(buffer, "%s" , store_value);
    printf("Value in buffer: %s",store_value);
    printf("\nTotal items read: %d",total_read);
    return 0;
}
```

```
Value in buffer: Hello

Total items read: 1
```

What is the difference between printf () and sprintf ()?

The printf function formats and writes output to the standard output stream, stdout . The sprintf function formats and stores a series of characters and values

in the array pointed to by buffer. Any argument list is converted and put out according to the corresponding format specification in format.

Both printf and puts helps to display a string or a set of characters on to the standard output which is usually the computer screen.

The main difference between printf and puts is that printf does not move the cursor to the new line by default while puts moves the cursor to the new line by default.

Unit - 3

Programming and Problem Solving through C Language
O Level / A Level

Chapter - 6 : Functions

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.

Parameter

- A parameter is a special kind of variable, used in a function to refer to one of the pieces of data provided as input to the function to utilse.
- These pieces of data are called arguments.
- Parameters are Simply Variables.

Formal Parameter

- Parameter Written in Function Definition is Called "Formal Parameter.
- Formal parameters are always variables, while actual parameters do not have to be variables.

Actual Parameter

- Parameter Written in Function Call is Called "Actual Parameter".
- One can use numbers, expressions, or even function calls as actual parameters.

Example

```
void display(int para1)
{
        printf( " Number %d " , para1);
}

void main()
{       int num1;
        display(num1);
}
```

In above, **para1** is called the Formal Parameter
num1 is called the Actual Parameter.

Parameter list

- A function is declared in the following manner:

  return-type function-name (parameter-list,...)
  { body... }

- Return-type is the variable type that the function returns.
- This cannot be an array type or a function type.
- If not given, then int is assumed.
- Function-name is the name of the function.
- Parameter-list is the list of Formal Arguments Variable.
- Given below are few examples:
  - int func1(int a, int b) /* two argument – int , int */
  - float func2(int a, float b) /* two argument – int , float */
  - void func3( ) /* No argument . */

The Function Return Type

- The function return type specifies the data type that the function returns to the calling program.
- The return type can be any of C's data types: **char, int , long , float , or double** .
- One can also define a function that doesn't return a value by using a return type of **void**.
- Given below are few examples:
  - int func1(...) /* Returns a type int. */
  - float func2(...) /* Returns a type float. */
  - void func3(...) /* No Returns . */

```
#include<stdio.h>

int multiply(int a, int b);

int main()
{
    ... ...
    result = multiply(i, j);
    ... ...
}

int multiply(int a, int b)
{
    ... ...
    return a*b;
}
```

The value returned by the function must be stored in a variable.

Program – A function that return the maximum between two numbers

```
int max(int num1, int num2)
{       /* local variable declaration */
        int result;
        if (num1>num2)
            result=num1;
        else
            result=num2;
        return result;
}

void  main()
{       int r;
        r=max( 10,15);
        print("Result = %d " , r);
}
```

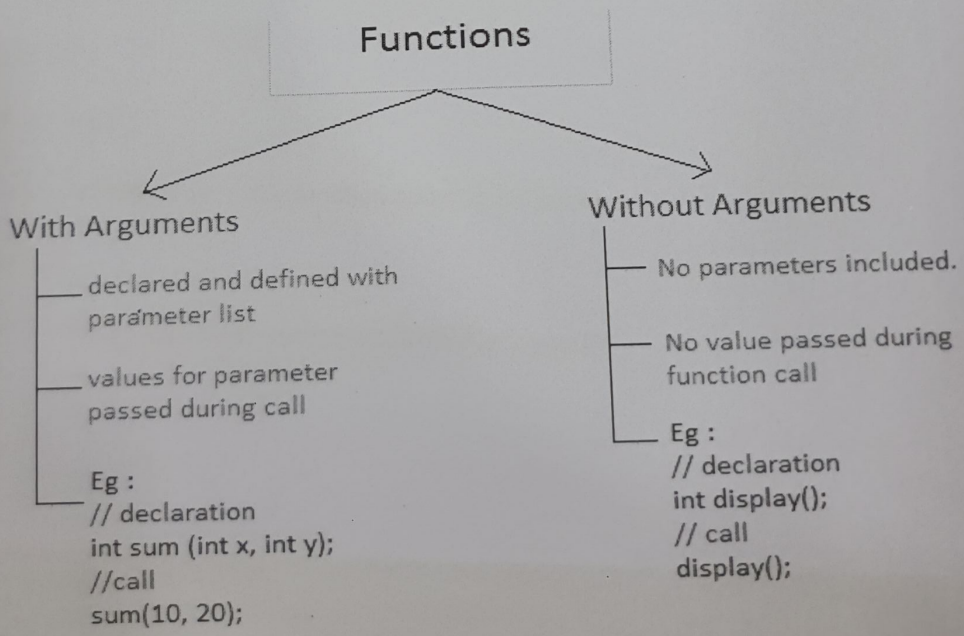
Calling Functions

There are two ways to call a function.

1) Any function can be called by simply using its name and argument list alone in a statement, as i n the following example.
   i) If the function has a return value, it is discarded.

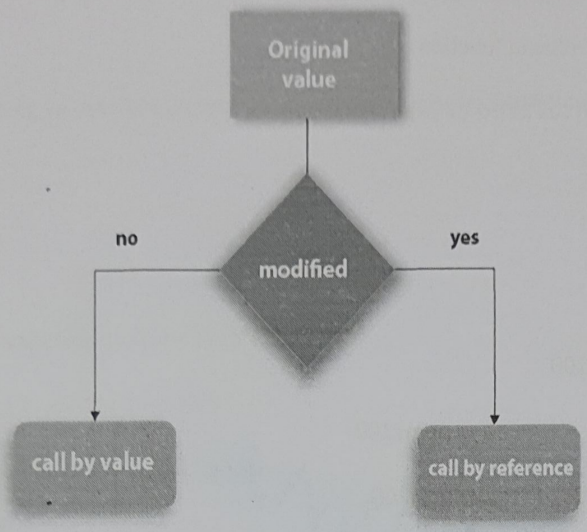
```
wait(12);
```

2) The second method can be used only with functions that have a return value.
   i) Because these functions evaluate to a value (that is, their return value), they are valid C expressions and can be used anywhere a C expression can be used.
   ii) An expression with a return value used as the right side of an assignment statement.



Call by value and Call by reference in C

There are two methods to pass the data into the function in C language, i.e., call by value and call by reference.



Call by value in C

In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.

In call by value method, we can not modify the value of the actual parameter by the formal parameter.

In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.

The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Let's try to understand the concept of call by value in c language by the example given below:

```
#include<stdio.h>

void change(int num) {

   printf("Before adding value inside function num=%d \n",num);

   num=num+100;

   printf("After adding value inside function num=%d \n", num);
```

```
}
int main() {
   int x=100;
   printf("Before function call x=%d \n", x);
   change(x);//passing value in function
   printf("After function call x=%d \n", x);
return 0;
}
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

Call by Value Example: Swapping the values of the two variables

```
#include <stdio.h>
void swap(int , int); //prototype of the function
int main()
{
   int a = 10;
   int b = 20;
   printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
   swap(a,b);
   printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual parameters do not change by changing the formal parameters in call by value, a = 10, b = 20
}
void swap (int a, int b)
```

```
{
   int temp;
   temp = a;
   a=b;
   b=temp;
   printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a = 20, b =
10
}
```

Output

Before swapping the values in main a = 10, b = 20

After swapping values in function a = 20, b = 10

After swapping values in main a = 10, b = 20

Call by reference in C

In call by reference, the address of the variable is passed into the function call as the actual parameter.

The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.

In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Consider the following example for the call by reference.

```
#include<stdio.h>
void change(int *num) {
   printf("Before adding value inside function num=%d \n",*num);
   (*num) += 100;
   printf("After adding value inside function num=%d \n", *num);
}
int main() {
```

```
    int x=100;
    printf("Before function call x=%d \n", x);
    change(&x);//passing reference in function
    printf("After function call x=%d \n", x);
return 0;
}
```

Output

Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=200

Call by reference Example: Swapping the values of the two variables

```
#include <stdio.h>
void swap(int *, int *); //prototype of the function
int main()
{
   int a = 10;
   int b = 20;
   printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
   swap(&a,&b);
   printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual parameters do change in call by reference, a = 10, b = 20
}
void swap (int *a, int *b)
{
   int temp;
   temp = *a;
```

```
*a=*b;

*b=temp;

printf("After swapping values in function a = %d, b = %d\n",*a,*b); // Formal parameters, a = 20, b
= 10

}
```

Output

Before swapping the values in main a = 10, b = 20

After swapping values in function a = 20, b = 10

After swapping values in main a = 20, b = 10

Difference between call by value and call by reference in c

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location