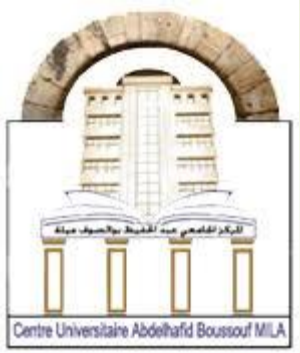
Abdelhafid Boussouf
University
Center of Mila



Structure of Computers and Applications
1st year ST – ENGINEERING

Part 2: The basics of Algorithm and Program

Course 11_12_13: Functions in C Language

By

Dr. Farouk KECITA

Academic year : 2024/2025

Introduction

❖ What is a Function?

A function as **series** of instructions or **group** of statements with one **specific purpose**.

- A function is a **program segment** that **carries** out some specific, well defined **task**.
- A function is a **self contained block** of code that performs a **particular task**.

❑ Suppose we are building an application in C language and in one of our program, we need to perform a **same task more than once**. In such case we have two options:

a. **Use the same set of statements every time** we want to perform the **task**

b. **Create a function** to perform that **task**, and just call it every time we need to perform that task.

Here option (b) is obviously a good practice.

Introduction

❖ Need functions in C

Functions are used because of following reasons :

- To improve the readability of code.
- Improves the reusability of the code, same function can be used in any program rather than writing the same code again.
- Debugging of the code would be easier if we use functions, as errors are easy to be traced.
- Reduces the size of the code, duplicate set of statements are replaced by function calls.

I. Types of functions

C functions can be classified into **two** types,

1) **Library functions** /pre defined functions /standard functions /built in functions
2) **User defined functions.**

1. Library functions:

- These functions are defined in the **library of C compiler** which are used frequently in the C program.
- These functions are **written by designers** of **C compiler**.

❑ C supports many built in functions like:

- Mathematical functions.
- String manipulation functions.
- Input and output functions.
- Memory management functions.
- Error handling functions.

I. Types of functions

❑ **EXAMPLES:**

- **pow(x,y)-**computes **x^y.**
- **sqrt(x)-**computes **square** root of x.
- **printf()-**used **to print** the data on the screen.
- **scanf()-**used **to read** the data from keyboard.

2. User Defined Functions:

➢ The functions written by **the programmer** /user **to do the specific tasks** are called user defined function(UDF's).

➢ The **user** can construct their own functions **to perform some specific task**. This type of functions created by the user is **termed** as **User defined functions.**

II. Elements of User Defined Function

The Three Elements of User Defined function structure consists of :

a) **Function Definition**

b) **Function Declaration / prototype**

c) **Function call**

A. Function Definition:

- A program Module written to achieve a specific task is called as function definition.
- Each function definition consists of two parts:

i. **Function header**

ii. **Function body**

II. Elements of User Defined Function

❖ General syntax of function definition

Function Definition Syntax	Function Definition Example

Datatype functionname(parameters)
{
declaration part;
executable part;
return statement;
 	
}
	void add()
	{
	int sum,a,b;
	printf("enter a and b\n");
	scanf("%d%d",&a,&b);
	sum=a+b;
	printf("sum is %d",sum);
	}

II. Elements of User Defined Function

i. Function header

- **Syntax:** **data_type function_name(parameters)**
- It consists of **three** parts:

a) Data_type:

- **The data type can be int, float, char, double, void.**

This is the data type of the value that the function is expected to return to calling function

b) function_name:

✓ The name of the function. It should be a valid identifier.

c) parameters

✓ The parameters are list of variables enclosed within parenthesis.

✓ The list of variables should be separated by comma.

❑ **EXAMPLE: int add(int a, int b)**

II. Elements of User Defined Function

ii. Function body

- The function body consists of the set of instructions enclosed between **{ and } .**
- The function body consists of following three elements:

a) **declaration part:** **variables** used in function body.

b) **executable part:** set of **Statements** or **instructions** to do specific activity.

c) **return :** It is a keyword, it is used to **return control back to calling function.**

- ✓ If a function is not ***returning value*** then statement is:

return;

- ✓ If a function is ***returning value*** then statement is:

return value;

II. Elements of User Defined Function

B. Function Declaration / prototype

- function declaration Consists of the data type of function, name of the function and parameter list ending with semicolon.

Function Declaration Syntax
datatypefunctionname(type p1,type p2,.....type pn);
Example
int add(int a, int b);
void add(int a, int b);

- **Note: The function declaration should end with a semicolon ;**

II. Elements of User Defined Function

C. Function Call

- The method of calling a function to achieve a specific task is called as **function call**.
- A function call is defined as **function name followed by semicolon.**
- A function call is nothing but invoking a function at the **required place** in the program to **achieve a specific task**.

❑ EXAMPLE:

```
void main() {
  add( ); // function call without parameter
  }
```

II. Elements of User Defined Function

❖ Formal Parameters and Actual Parameters

1) Formal Parameters:

- The variables defined in the **function header of function definition** are called **formal** parameters.
- All the variables should be separately declared and each declaration must be separated by **commas.**
- The formal parameters **receive the data from actual parameters.**

2) Actual Parameters:

- The **variables** that are used when a function is invoked **(in function call)** are called **actual** parameters.
- Using actual parameters, the data can be transferred from ***calling function*** to the ***called function***.

II. Elements of User Defined Function

❖ Formal Parameters and Actual Parameters

- The corresponding **formal** parameters in the **function definition** receive them.
- The **actual** parameters and **formal** parameters must match in number and type of data.

3) Differences between Actual and Formal Parameters

Actual Parameters	Formal Parameters
Actual parameters are also called as **argument list.** Ex: add(m,n)	Formal parameters are also called as **dummy parameters.** **Ex:**int add(int a, int b)
The variables used in function call are called as actual parameters	The variables defined in function header are called formal parameters

II. Elements of User Defined Function

3) Differences between Actual and Formal Parameters

Actual parameters are used in calling function when a function is called or invoked Ex: add(m,n) Here, m and n are called actual parameters	Formal parameters are used in the function header of a called function. Example: int add(int a, int b) { ..... } Here, a and b are called formal parameters.
Actual parameters sends data to the formal parameters	Formal parameters receive data from the actual parameters.

III. Categories of the functions

- In C programming language, **function** can be called either with or without arguments and might return values. They may or might not return values to the calling functions.

1. Function **with no parameters** and **no return values.**
2. Function **with no parameters** and **return values.**
3. Function **with parameters** and **no return values.**
4. Function **with parameters** and **return values**.

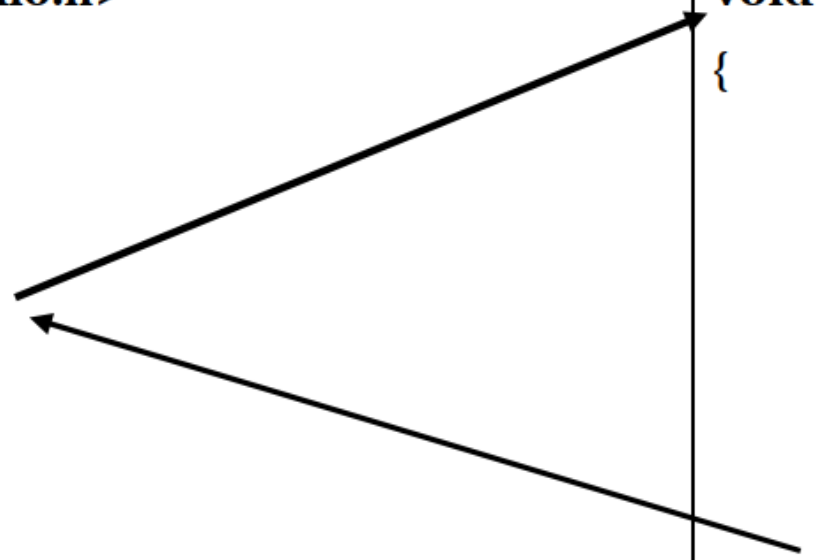
III. Categories of the functions

1. Function with no parameters and no return values.

1. Function with no parameters and no return values
(void function without parameter)

Calling function	Called function

/*program to find sum of two numbers
using function*/
#include<stdio.h>
void add( );
void main()
{
add();
}
	void add ( )
	{
	int sum;
	printf("enter a and b values\n");
	scanf("%d%d",&a,&b);
	sum=a+b;
	printf("\n The sum is %d", sum);
	return;
	}



III. Categories of the functions

1. Function with no parameters and no return values.

- In this category **no data** is transferred from **calling function** to **called function**, hence **called function cannot receive any values**.
- In the above example, no arguments are passed to user defined function **add( ).**
- Hence **no parameter** are defined in **function header**.
- When the control is transferred from calling function to called function, a and b values are read, they are added, the result is printed on monitor.
- When return statement is executed ,control is transferred from called function / add to calling function / main.

III. Categories of the functions

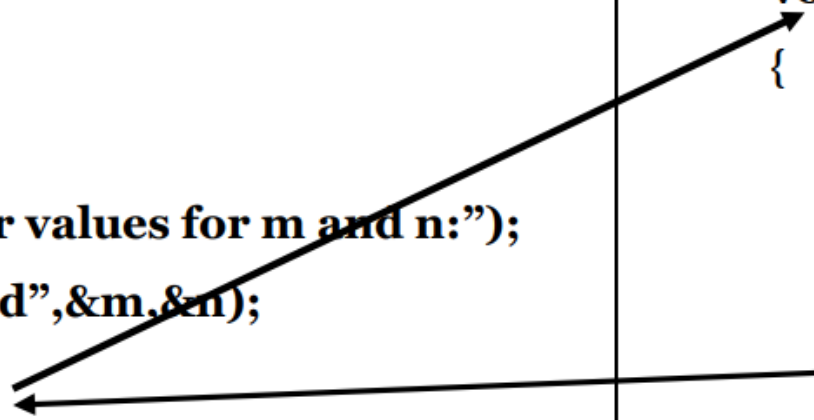
2. Function with no parameters and return values.

2. Function with parameters and no return values

(void function with parameter)

Calling function	Called function

/*program to find sum of two numbers
using function*/
#include<stdio.h>
void add(int m, int n);
void main()
{
int m,n;
printf("enter values for m and n:");
scanf("%d %d",&m,&n);
add(m,n);
}
	void add(int a, int b)
	{
	int sum;
	sum = a+b;
	printf("sum is:%d",sum);
	return;
	}



III. Categories of the functions

2. Function with no parameters and return values.

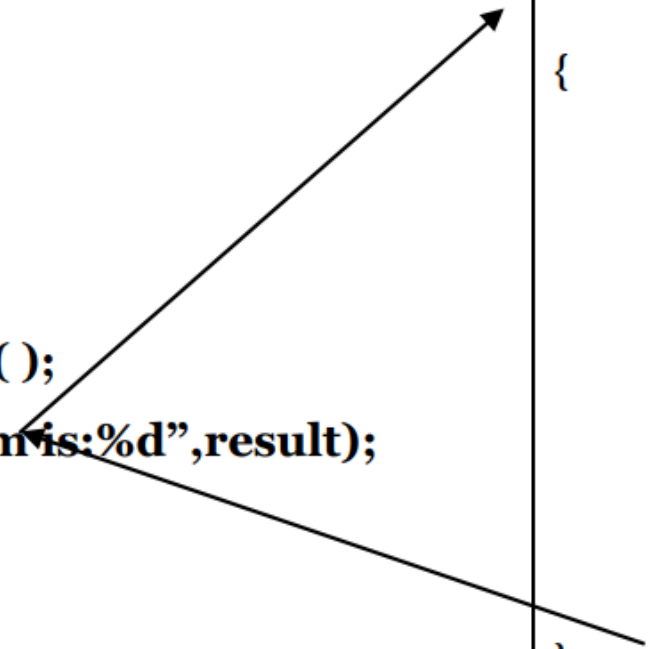
- In this category, there is **data transfer** from the **calling function** to the **called function** using parameters.
- But there is **no data transfer** from called function to the calling function.
- The values of actual parameters **m** and **n** are **copied** into **formal** parameters **a** and **b**.
- The value of a and b are added and result stored in sum is displayed on the screen in **called function itself**.

III. Categories of the functions

3. Function with parameters and no return values.

3. Function with no parameters and with return values	
Calling function	**Called function**

/*program to find sum of two numbers using function*/
#include<stdio.h>
int add();
void main()
{
int result;
result=add( );
printf("sum is:%d",result);
}
	int add( ) /* function header */
	{
	int a,b,sum;
	printf("enter values for a and b:");
	scanf("%d %d",&a,&b);
	sum= a+b;
	return sum;
	}



III. Categories of the functions

3. Function with parameters and no return values.

- In this category there is **no data transfer from** the **calling function to** the **called function**.
- But, there is **data transfer from called function to the calling function.**
- No **arguments** are **passed** to the function add( ). So, no **parameters** are **defined** in the **function header**.
- When the function returns a value, the calling function receives one value from the called function and **assigns** to variable **result**.
- The **result** value is **printed** in calling function.

III. Categories of the functions

4. Function with parameters and return values.

4. Function with parameters and with return values

Calling function	Called function

/*program to find sum of two numbers using function*/
#include<stdio.h>
int add();
void main()
{
int result,m,n;
printf("enter values for m and n:");
scanf("%d %d",&m,&n);
result=add(m,n);
printf("sum is:%d",result);
}
	int add(int a, int b) /* function header */
	{
	int sum;
	sum= a+b;
	return sum;
	}

III. Categories of the functions

4. Function with parameters and return values.

- In this category, there is data transfer between the calling function and called function.
- When Actual parameters values are passed, the formal parameters in called function can receive the values from the calling function.
- When the add function returns a value, the calling function receives a value from the called function.
- The values of actual parameters m and n are copied into formal parameters a and b.
- Sum is computed and returned back to calling function which is assigned to variable result.

IV. Passing parameters to functions / Types of argument passing

- The different ways of passing parameters to the function are:
- ✓ **Pass by value or Call by value.**
- ✓ **Pass by address or Call by address.**

1. Pass by value / Call by value.

- A copy of **actual arguments** is **passed** to **formal arguments** of the called function and **any change** made to the formal arguments in the called function have **no effect** on the values of actual arguments in the calling function.

2. Pass by address / Call by address.

- The **location (address)** of **actual arguments** is passed to **formal arguments** of the **called function**.
- This means by **accessing the addresses** of actual arguments we can **alter** them within from the **called function.**

IV. Passing parameters to functions or Types of argument passing

❖ Difference between call by value and call by reference:

Call by Value	Call by Address
When a function is called the **values of variables are passed**	When a function is called the **addresses of variables are passed**
The type of formal parameters should be same as type of actual parameters	The type of formal parameters should be same as type of actual parameters, but they have to be declared as **pointers.**
Formal parameters contains the values of actual parameters	Formal parameters contain the addresses of actual parameters.

❖ Example 01: of call by value:

```c
#include<stdio.h>
void change(int x);
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;
}
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);
}
```

OUT PUT:

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

❖ Example 02: of call by value:

```c
#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;
}
```

OUT PUT:

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

❖ Example 03: of call by reference:

```c
#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;
}
```

OUT PUT:

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

❖ Example 04: of call by reference:

```c
#include <stdio.h>
void swap(int *, int *);
int main()
{
    int a = 10;
    int b = 20;
    printf("Before swapping the values in main a = %d, b = %d\n",a,b);
    swap(&a,&b);
    printf("After swapping values in main a = %d, b = %d\n",a,b);
}
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);
}
```

OUT PUT:

```
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
```

V. Recursion functions

- **Recursion** is a method of solving the problem where the solution to a problem depends on solutions to **smaller instances** of the same problem.
- Recursive function is a **function** that **calls itself** during the execution.

❖ Basic Structure of Recursive Functions

The basic syntax structure of the recursive functions is:

type **function_name** (args) {
/// function statements
*// **base condition***
// recursion case (recursive call)
}

```
type function_name (args) {
// function statements
// base condition
// recursion case (recursive call)
}
```

❖ **Example 01:** In this example, recursion is used to calculate the factorial of a number.

```c
#include <stdio.h>
int fact (int) ;
int main()  {
    int n,f;
    printf("Enter the number: ");
    scanf("%d",&n);
    f = fact(n);
    printf("factorial = %d",f);
}
int fact(int n)  {
    if (n==0)  {
        return 0;
    }
    else if ( n == 1) {
        return 1;
    }
    else  {
        return n*fact(n-1);
    }
}
```

OUT PUT 01:

```
Enter the number: 5
factorial = 120
```

OUT PUT 02:

```
Enter the number: 10
factorial = 3628800
```

❖ **Example 02:** In this example, recursion is used to calculate the sum of the first N natural numbers.

```c
#include <stdio.h>
int nSum(int n);
int main() {
    int N;
    printf("Enter the velue of N: ");
    scanf("%d", &N);
    // calling the function
    int sum = nSum(N);
    printf("Sum of First %d Natural Numbers: %d", N, sum);
    return 0; }
    int nSum(int N){
// base condition to terminate the recursion when N = 0
    if (N == 0) {
        return 0;
    }
    // recursive case / recursive call
    int res = N+ nSum(N- 1);
    return res;}
```

OUT PUT 01:

```
Enter the velue of N: 5
Sum of First 5 Natural Numbers: 15
```

OUT PUT 02:

```
Enter the velue of N: 10
Sum of First 10 Natural Numbers: 55
```