UNIVERSITY OF MILA

Faculty of Science and Technology

Department of Process Engineering

Level: 1st year ST - ENG & LMD

Introduction to Programming

Lectures 03 & 04:

Functions in C

---

by:

Dr. Farouk KECITA

Academic Year 2025/2026

1 Introduction to Functions

A function is a series of instructions or group of statements with one specific purpose. It 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 the same task more than once. In such case we have two options:

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

1.1 Why 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.

2 Types of Functions

C functions can be classified into two types:

1. Library functions / predefined functions / standard functions / built-in functions
2. User-defined functions

2.1 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

Examples of Library Functions:

- `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

2.2 User-Defined Functions

The functions written by the programmer/user to do specific tasks are called user-defined functions (UDF's). The user can construct their own functions to perform some specific task.

3 Elements of User-Defined Function

The three elements of user-defined function structure consists of:

1. Function Definition
2. Function Declaration / Prototype
3. Function Call

3.1 Function Definition

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

- Function header
- Function body

General Syntax of Function Definition:

```
datatype functionname(parameters)
{
    declaration part;
    executable part;
    return statement;
}
```

Example of Function Definition:

```
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);
}
```

3.1.1 Function Header

Syntax: `data_type function_name(parameters)`

It consists of three parts:

- 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.
- function_name: The name of the function. It should be a valid identifier.
- 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)`

3.1.2 Function Body

The function body consists of the set of instructions enclosed between `and` . The function body consists of following three elements:

- Declaration part: variables used in function body.
- Executable part: set of statements or instructions to do specific activity.
- Return: It is a keyword, it is used to return control back to calling function.

Return Statement

- If a function is not returning value: `return;`
- If a function is returning value: `return value;`

3.2 Function Declaration / Prototype

Function declaration consists of the data type of function, name of the function and parameter list ending with semicolon.

Syntax: `datatype functionname(type p1, type p2, ..., type pn);`

Examples:

- `int add(int a, int b);`
- `void add(int a, int b);`

Note The function declaration should end with a semicolon `;`

3.3 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
}
```

3.4 Formal Parameters and Actual Parameters

3.4.1 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.

3.4.2 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.

3.4.3 Differences between Actual and Formal Parameters

Formal Parameters	Actual Parameters
1. They are used in function header.	1. They are used in the function call.
2. They are used to receive values that are passed to the function in function call.	2. They are the actual values that are passed to function definition in function call.
3. They are similar to local variables of the function in which they are used.	3. They may be constant values or variable names.

Table 1: Comparison between Formal Parameters and Actual Parameters

4 Categories of 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

4.1 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. No arguments are passed to user-defined function.

Listing 1: Function with no parameters and no return values

```c
#include <stdio.h>
void add();  // function declaration

void main() {
    add();  // function call
}

void add() {  // function definition
    int sum, a, b;
    printf("enter a and b values\n");
    scanf("%d%d", &a, &b);
    sum = a + b;
    printf("\n The sum is %d", sum);
    return;
}
```

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).

4.2 Function with Parameters and No 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.

Listing 2: Function with parameters and no return values

```c
#include <stdio.h>
void add(int m, int n);  // function declaration

void main() {
    int m, n;
    printf("enter values for m and n: ");
    scanf("%d %d", &m, &n);
    add(m, n);  // function call with parameters
}

void add(int a, int b) {  // function definition
    int sum;
    sum = a + b;
    printf("sum is: %d", sum);
    return;
}
```

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.

4.3 Function with No Parameters and 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.

Listing 3: Function with no parameters and return values

```c
#include <stdio.h>
int add();  // function declaration

void main() {
    int result;
    result = add();  // function call
    printf("sum: %d", result);
```

```
}

int add() {  // function definition
    int a, b, sum;
    printf("enter values for a and b: ");
    scanf("%d %d", &a, &b);
    sum = a + b;
    return sum;  // return value to 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.

4.4 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 function returns a value, the calling function receives a value from the called function.

Listing 4: Function with parameters and return values

```
#include <stdio.h>
int add(int a, int b);  // function declaration

void main() {
    int result, m, n;
    printf("enter values for m and n: ");
    scanf("%d %d", &m, &n);
    result = add(m, n);  // function call with parameters
    printf("sum is: %d", result);
}

int add(int a, int b) {  // function definition
    int sum;
    sum = a + b;
    return sum;  // return value to calling 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.

5 Parameter Passing Methods / Types of Argument Passing

The different ways of passing parameters to the function are:

1. Pass by value or Call by value
2. Pass by address or Call by address

5.1 Pass by Value / Call by Value

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

5.2 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.

5.3 Difference between Call by Value and Call by Address

CALL BY VALUE	CALL BY REFERENCE
Calling function sends copies of data.	Calling function sends addresses of data.
The formal parameters are ordinary variables.	The formal parameters are pointer variables.
At most only one value can be sent back to the calling function.	Several results can be sent back to the calling function.
Actual parameters are not affected by changes made within the function.	Direct changes are made to the actual parameters.

Table 2: Difference between Call by Value and Call by Reference

5.4 Example 01: Call by Value

Listing 5: Example 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;
}
```

```
11
12 void change(int num) {
13     printf("Before adding value inside function num = %d \n",
           num);
14     num = num + 100;
15     printf("After adding value inside function num = %d \n", num
           );
16 }
```

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

Note that the value of x in main() remains unchanged (100) even though it was changed inside the function. This demonstrates that call by value passes a copy, not the original variable.

5.5 Example 02: Call by Value (Alternative)

Listing 6: Another Example of Call by Value

```
1  #include <stdio.h>
2  void change(int num) {
3      printf("Before adding value inside function num = %d \n",
           num);
4      num = num + 100;
5      printf("After adding value inside function num = %d \n", num
           );
6  }
7
8  int main() {
9      int x = 100;
10     printf("Before function call x = %d \n", x);
11     change(x);  // passing value in function
12     printf("After function call x = %d \n", x);
13     return 0;
14 }
```

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

5.6 Example 03: Call by Address / Reference

Listing 7: Example of Call by Address

```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 address in function
    printf("After function call x = %d \n", x);
    return 0;
}
```

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

Notice that the value of x in main() has changed to 200. This is because we passed the address of x, allowing the function to modify the original variable directly.

5.7 Example 04: Call by Address - Swapping Two Numbers

Listing 8: Swapping Numbers using Call by Address

```c
#include <stdio.h>
void swap(int *, int *);
int main() {
    int a = 10;
    int b = 20;
    printf("Before swapping in main: a = %d, b = %d\n", a, b);
    swap(&a, &b);  // passing addresses
    printf("After swapping in main: a = %d, b = %d\n", a, b);
    return 0;
}
void swap(int *a, int *b) {
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
    printf("After swapping in function: a = %d, b = %d\n", *a, *
       b); }
```

```
Before swapping in main: a = 10, b = 20
After swapping in function: a = 20, b = 10
After swapping in main: a = 20, b = 10
```

This example demonstrates a classic use of call by address. By passing addresses, the swap function can directly modify the original variables a and b in main().

6 Recursive Functions

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

6.1 Basic Structure of Recursive Functions

Syntax Structure of Recursive Functions:

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

6.2 Example 01: Factorial using Recursion

Listing 9: Calculate Factorial using Recursion

```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);
    return 0;
}
int fact(int n) {
    if (n == 0) {
        return 0;
    }
    else if (n == 1) {
        return 1;
    } else {
        return n * fact(n - 1);  // recursive call
    } }
```

```
OUTPUT 01:
Enter the number: 5
factorial = 120
OUTPUT 02:
Enter the number: 10
factorial = 3628800
```

6.3 Example 02: Sum of Natural Numbers using Recursion

Listing 10: Sum of First N Natural Numbers using Recursion

```c
#include <stdio.h>
int nSum(int n);

int main() {
    int N;
    printf("Enter the value 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;
}
```

```
OUTPUT 01:
Enter the value of N: 5
Sum of First 5 Natural Numbers: 15

OUTPUT 02:
Enter the value of N: 10
Sum of First 10 Natural Numbers: 55
```

How Recursion Works:

- The function calls itself with a smaller value each time
- A base condition stops the recursion (when N == 0)
- Without a base condition, recursion would continue indefinitely (stack overflow)

Summary

Key Takeaways:

- Functions improve code reusability, readability, and maintainability
- Always declare functions before using them (function prototype)
- Use call by value when you don't want to modify original variables
- Use call by address when you need to modify original variables
- Recursion requires a base condition to prevent infinite loops
- Each recursive call creates a new stack frame