C Programming Short Notes

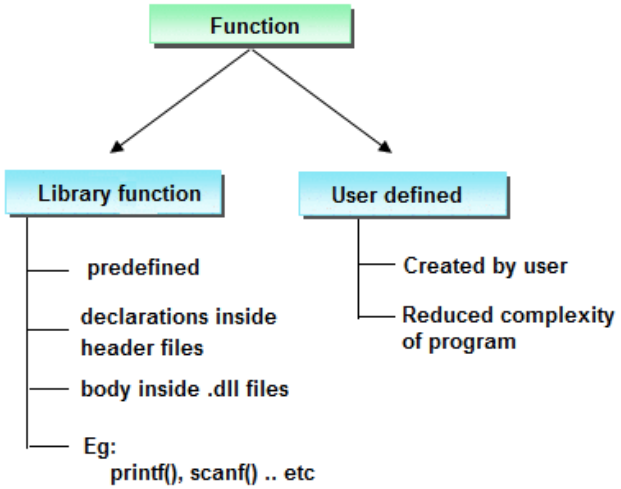
Part 3

1. Functions

- A function in C is a **set of statements that perform a specific task.**
- It serves as the **basic building block of a C program**, providing **modularity** and code **reusability.**
- Functions are enclosed within { } braces and can be thought of as subroutines or procedures in other languages.

2. Types of functions

- **Library Functions**
- **User-defined Functions**



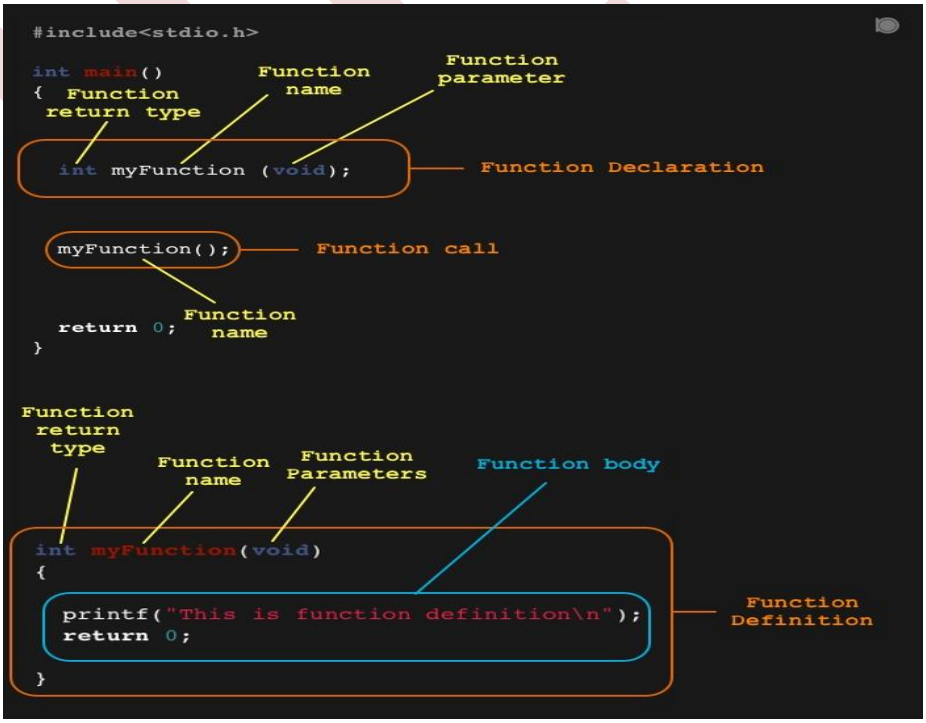
2.1 Library Functions (Predefined Functions):

- These functions are **built-in and provided by the C standard library**.
- They serve common purposes and can be directly used in your programs.
- Examples of library functions include **printf(), scanf(), sqrt(), and strlen().**
- These functions are **already defined and ready for use without any additional effort on your part.**

2.2 User-Defined Functions:

- These functions are **created by the programmer**.
- They allow you to **break down your program into smaller, manageable pieces.**
- User-defined functions enhance modularity and code reusability.
- You can define your own functions **based on the specific requirements of your program**.
- These functions are **declared and defined by you, and they perform custom tasks within your program**

Now we see how to create User-Defined Functions:-



2.2.1 Function Declaration:

In a function declaration, we specify:

- The function name.
- Its return type.
- The number and type of its parameters (if any).

A function declaration informs the compiler that a function with the given name exists elsewhere in the program.

The parameter names are optional in the declaration.
Example:

```
int sum(int a, int b);
```

or even without parameter names:

```
int sum(int, int);
```

2.2.2 Function Definition:

- The function definition contains **the actual statements executed when the function is called.**
- A C function is typically defined and declared in a single step.

Example:

```
int sum(int a, int b) {
   return a + b;
}
```

2.2.3 Function Call:

- A function call **instructs the compiler to execute the function**.
- We use the function name and provide the necessary parameters.

Example:

```
int main() {
   int result = sum(10, 30);
   printf("Sum is: %d", result);
   return 0;
 }                                    // Output: “Sum is: 40”
```

2.2.4 Function Return Type:

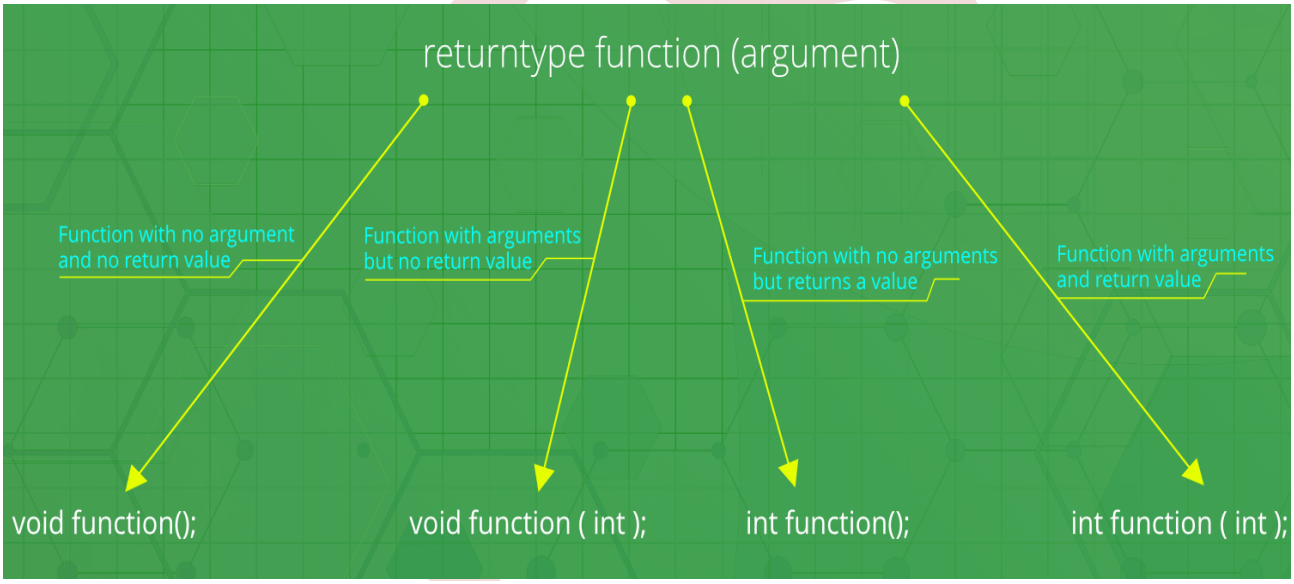
- The return type specifies the type of **value returned after executing the function**.
- If no value needs to be returned, we can use the **void data type**.

Example:

```
int func(int param1, int param2);
```

Here, the function returns an integer value after executing its statements**. A function call is necessary to execute the function's statements. If not called, the function's code won't run.** Functions enhance code organization, readability, and maintainability in C programs.

3. Types of Return Types and Arguments in Functions



3.1 Function with No Arguments and No Return Value:

These functions perform a specific task without taking any input parameters and do not return any value.

Example:

```
void greetUser() {
   printf("Hello, user!\n");
```

```
}          //When you call greetUser(), it simply prints “Hello, user!” to the console.
```

3.2 Function with No Arguments and With Return Value:

These functions do not take any input parameters but return a value.
Example:

```
int getRandomNumber() {
   return rand(); // Returns a random integer
}
```

You can call getRandomNumber() to get a random integer.

3.3 Function with Arguments and No Return Value:

These functions accept input parameters (arguments) but do not return any value.
Example:

```
void printSum(int a, int b) {
    printf("Sum: %d\n", a + b);
}
```

Calling printSum(10, 20) will print the sum of 10 and 20.

3.4 Function with Arguments and With Return Value:

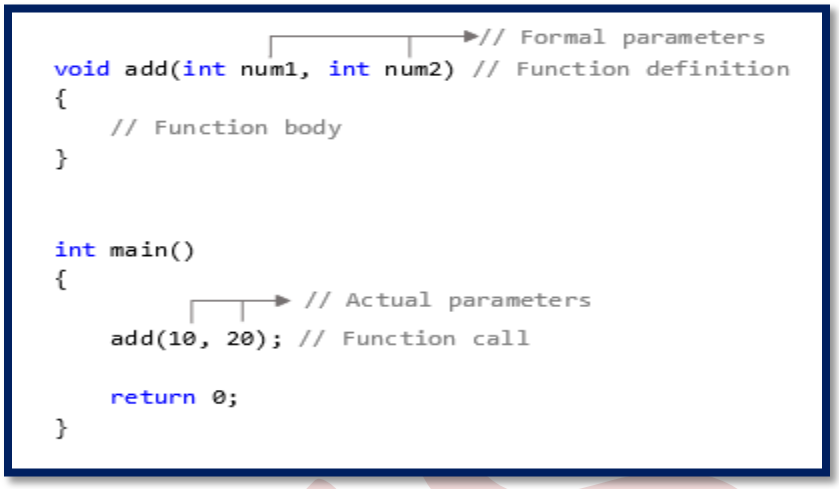
These functions take input parameters and return a value.

Example:

```
int multiply(int x, int y) {
   return x * y;
}
```

The expression multiply(5, 3) evaluates to 15.

4. Passing parameters to functions



- **Actual Parameter**
- **Formal Parameter**

Actual parameters are the **values passed during function calls**, while formal parameters are the **variables declared in the function definition** to accept those values.

1. **Formal Parameters**:

   - Formal parameters, also called **parameters** or **parameters in the function definition**, are **local variables** declared within the function.
   - These variables serve as placeholders to receive the **values** passed by the calling function (i.e., the actual parameters).

   Example:
   int a(int a, int b)

2. **Actual Parameters:**

   - Actual parameters, also known as **arguments**, are the **values** that are **passed** to a function when it is **called**.
   - These values represent the **input data** that the function will operate on.

   Example:
   int add(5,6)

We can pass arguments to the C function in two ways:

- **Call by Value:** Passes values as copies, preserving the original data.
- **Call by Reference:** Passes references (pointers) to memory locations, directly modifying the original values

Call by Value :

- In Call by Value, **the values of actual parameters (variables defined in the calling function) are copied into the formal parameters (variables declared in the invoked function).**
- There are two separate copies of parameters stored in different memory locations**: the original copy and the function's copy.**
- Any changes made inside **the function do not affect the actual parameters of the caller.**

Example

```
#include <stdio.h>
void swapx(int x, int y);
int main() {
   int a = 10, b = 20;
   swapx(a, b);
   printf("In the Caller:\n");
   printf("a = %d, b = %d\n", a, b);
   return 0;
}
void swapx(int x, int y) {
   int t;
   t = x;
   x = y;
   y = t;
   printf("Inside Function:\n");
   printf("x = %d, y = %d\n", x, y);
}
```

Output:

```
Inside Function: x = 20, y = 10
In the Caller: a = 10, b = 20
```

The actual values of a and b remain unchanged even after exchanging the values of x and y in the function.

.

1. **Call by Reference :**
   - In Call by Reference, **the address of the actual parameters is passed to the function as the formal parameters.**
   - Pointers are used to achieve Call by Reference.
   - **Both the actual and formal parameters refer to the same memory locations.**
   - Changes made inside **the function are reflected in the actual parameters of the caller.**

Example

```
#include <stdio.h>
void swapx(int* x, int* y);

int main() {
   int a = 10, b = 20;
   swapx(&a, &b);
   printf("Inside the Caller:\n");
   printf("a = %d, b = %d\n", a, b);
   return 0;
}

void swapx(int* x, int* y) {
   int t;
   t = *x;
   *x = *y;
   *y = t;
   printf("Inside the Function:\n");
   printf("x = %d, y = %d\n", *x, *y);
}
```

Output:

Inside the Function: x = 20, y = 10
Inside the Caller: a = 20, b = 10

The actual values of a and b change after exchanging the values of x and y.