## 1. Types of Functions in C

C functions can generally be categorized as follows:

1. **Library Functions:** Predefined functions provided by the C standard library. Examples include `printf()`, `scanf()`, `strcpy()`, and `malloc()`.
2. **User-Defined Functions:** Functions created by the programmer to perform specific tasks. These can be further categorized based on their return type and parameters:
    - **Returning Functions:** Functions that return a value.
    - **Void Functions:** Functions that do not return a value.

## 2. Creating User-Defined Functions

To create a user-defined function, follow these steps:

1. **Function Declaration:** Declare the function prototype before `main()` or at the top of the file.
2. **Function Definition:** Define the function with the implementation.
3. **Function Call:** Call the function from `main()` or other functions.

**Example:**

```c
#include <stdio.h>

// Function declaration
int add(int a, int b);

// Main function
int main() {
    int result = add(5, 10);
    printf("Sum: %d\n", result);
    return 0;
}

// Function definition
int add(int a, int b) {
    return a + b;
}
```

## 3. Local vs Global Variables

- **Local Variables:** Variables declared inside a function. They are only accessible within that function and are created when the function starts executing and destroyed when it exits.

    ```c
    void function() {
        int localVar = 10; // Local Variable
    }
    ```

- **Global Variables:** Variables declared outside of any function. They can be accessed by any function throughout the program.

```c
int globalVar = 20; // Global Variable

void function() {
    printf("%d\n", globalVar); // Accessing global variable
}
```

### 4. Memory Layout in C

In C, the memory layout typically consists of the following segments:

- **Text Segment:** Where the compiled code is stored.
- **Data Segment:** Where global and static variables are stored (initialized and uninitialized).
- **Heap Segment:** Dynamically allocated memory, managed using functions such as `malloc()`, `calloc()`, `free()`.
- **Stack Segment:** Where local variables are stored; function calls and return addresses are also managed here.

### 5. Actual Parameter and Formal Parameter

- **Actual Parameters (Arguments):** The values or references you pass to a function when you call it. For example, in `add(5, 10)`, `5` and `10` are actual parameters.
- **Formal Parameters:** The variables defined by a function that receive the actual parameters. For example, in the function definition `int add(int a, int b)`, `a` and `b` are formal parameters.

### 6. Call by Value vs. Call by Reference

#### Call by Value

In Call by Value, a copy of the actual parameter is passed to the function. Changes made to the parameter inside the function do not affect the original variable.

```c
#include <stdio.h>

void modify(int x) {
    x = x + 10; // Changes x but does not affect original variable
}

int main() {
    int num = 5;
    modify(num);
    printf("Original num: %d\n", num); // Prints: Original num: 5
    return 0;
}
```

#### Call by Reference

In Call by Reference, a reference (pointer) to the actual parameter is passed to the function, allowing modifications made within the function to impact the original variable.

```c
#include <stdio.h>

void modify(int *x) {
    *x = *x + 10; // Changes the original variable through pointer
}

int main() {
    int num = 5;
    modify(&num); // Passes the address of num
    printf("Modified num: %d\n", num); // Prints: Modified num: 15
    return 0;
}
```

### 7. Recursive Functions

A recursive function is one that calls itself. Recursion is a useful technique for solving problems that can be divided into smaller, similar problems.

**Example:**

Here's a simple example of a recursive function that calculates the factorial of a number:

```c
#include <stdio.h>

// Recursive function to find factorial
int factorial(int n) {
    if (n == 0) { // Base case
        return 1;
    } else {
        return n * factorial(n - 1); // Recursive call
    }
}

int main() {
    int num = 5;
    printf("Factorial of %d: %d\n", num, factorial(num));
    return 0;
}
```

### Summary

- **Types of Functions:** Built-in and user-defined functions.
- **User-Defined Functions:** Plan and implement function declarations, definitions, and calls.
- **Local vs Global Variables:** Local variables have function scope; global variables have program-wide scope.
- **Memory Layout:** Consists of text, data, heap, and stack segments.
- **Actual vs Formal Parameters:** Actual parameters are passed into functions; formal parameters are declared in function definitions.
- **Call by Value vs Reference:** Determines if function modifications impact the original variable.

- **Recursive Functions:** Define and implement functions that call themselves to solve problems.