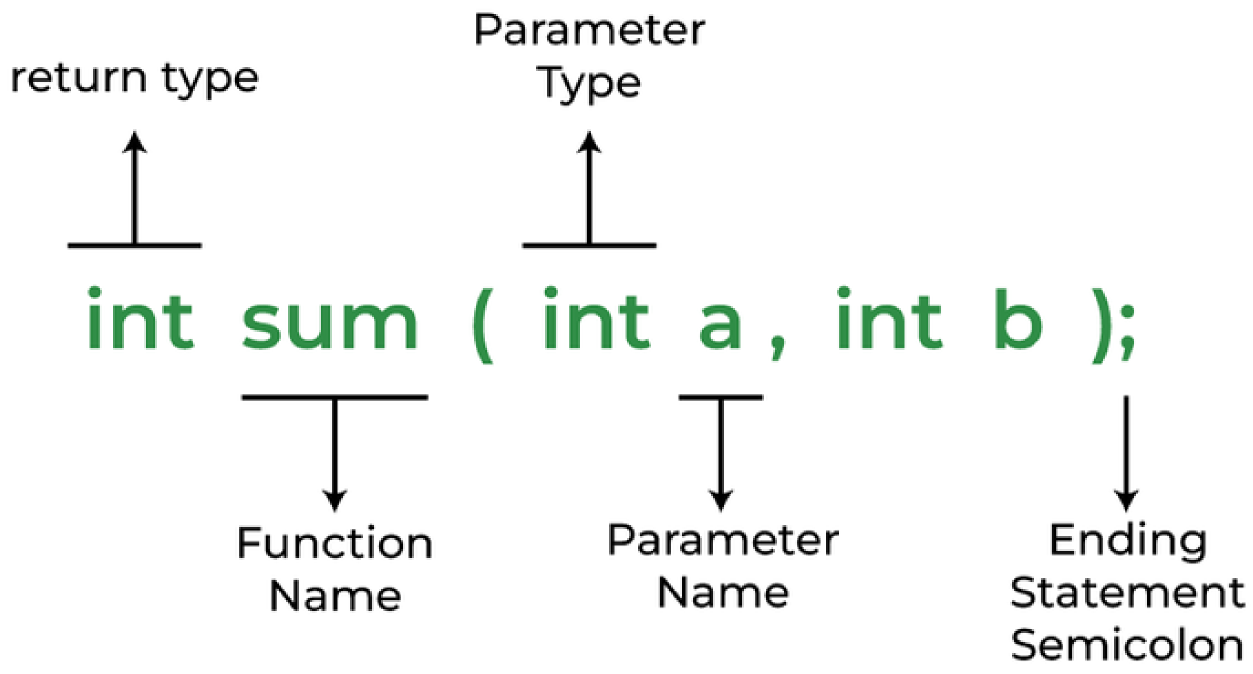
Function

In C programming, a function is a block of statements that performs a specific task when called. Functions enhance modularity and code reusability, allowing developers to break down complex problems into manageable sub-tasks. In other programming languages, functions are also referred to as subroutines or procedures.

Advantages of Using Functions

- **Modularity** --> Functions allow the decomposition of a program into smaller, manageable sections, making the code easier to understand and maintain.
- **Code Reusability** --> Once a function is defined, it can be reused multiple times throughout the program, reducing redundancy.
- **Ease of Testing** --> Individual functions can be tested independently, facilitating debugging and validation.

Function Components



A typical function in C consists of:

1. **Return Type** --> Specifies the type of value the function returns. If no value is returned, the return type is `void`.
2. **Function Name** --> An identifier for the function, following the same naming conventions as variables.
3. **Parameters (Optional)** --> Variables that accept values from the function call. Functions can have zero or more parameters.
4. **Function Body** --> A block of code enclosed in `{}` braces that defines the operations performed by the function.

Function Declaration (Prototype)

A function declaration, or prototype, informs the compiler about a function's name, return type, and parameters before its actual definition. This is essential for ensuring that function calls are correctly matched with their definitions.

Syntax:

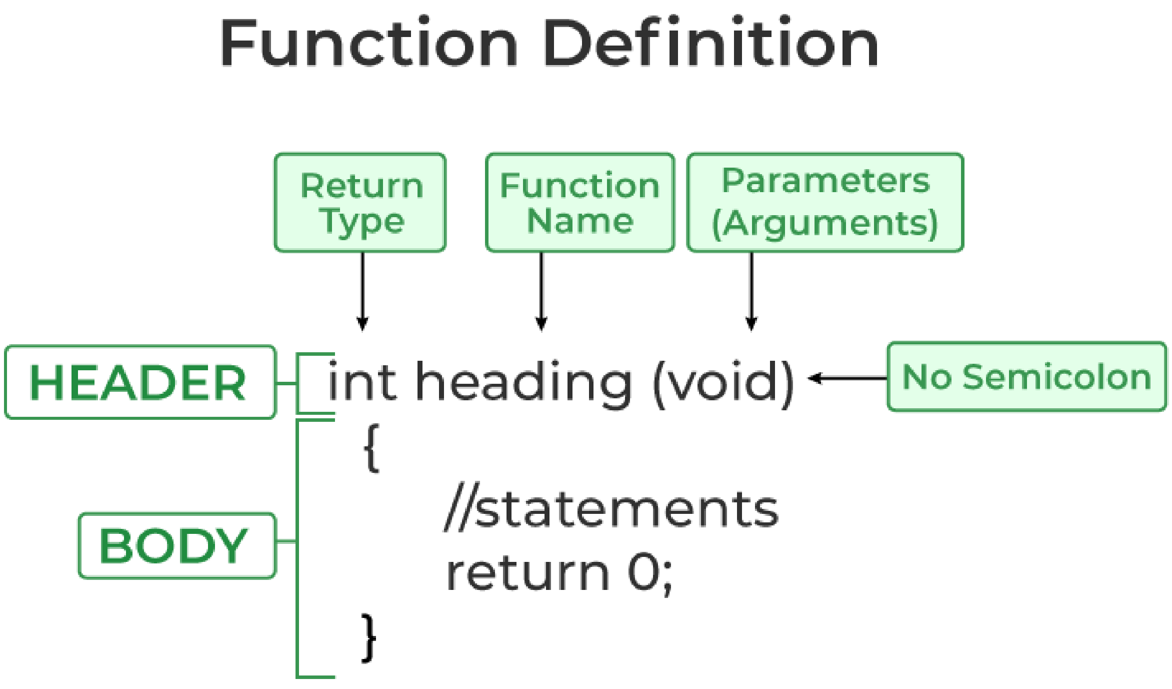
```
return_type function_name(parameter_type1 parameter1, parameter_type2 parameter2, ...);
```

Example:

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

This prototype declares a function named `add` that takes two integer parameters and returns an integer.

Function Definition



The function definition provides the actual implementation.

Syntax:

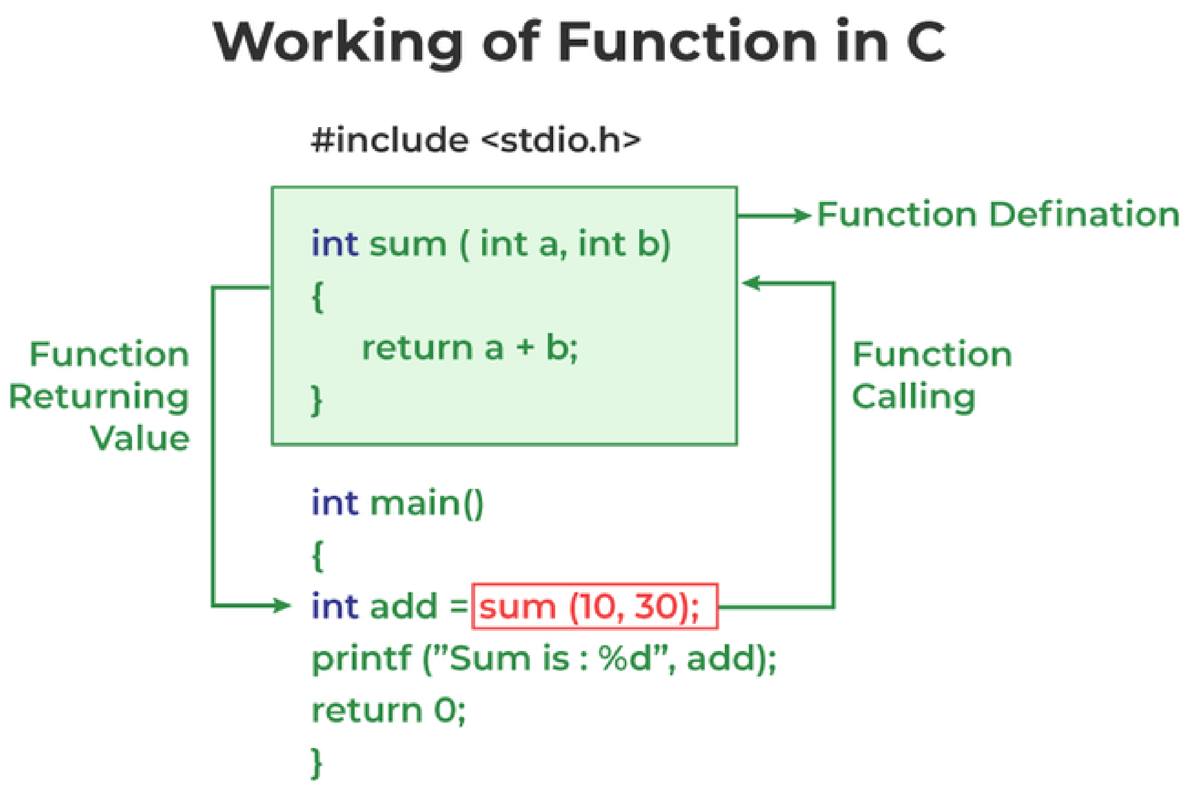
```
return_type function_name(parameter_type1 parameter1, parameter_type2 parameter2, ...) {
    // Function body
    return value; // if return_type is not void
}
```

Example:

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

This function `add` takes two integers and returns their sum.

Function Call



```
#include <stdio.h>

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

int main()
{
int add = sum (10, 30);
printf ("Sum is : %d", add);
return 0;
}
```

To execute a function, you call it by using its name followed by arguments in parentheses.

Syntax:

```
function_name(argument1, argument2, ...);
```

Example:

```
int result = add(5, 3); // result now holds the value 8
```

Here, the `add` function is called with arguments `5` and `3`, and the returned value is stored in `result`.

Types of Functions

1. **Standard Library Functions** --> Predefined functions provided by C's standard library, such as `printf()`, `scanf()`, and `strlen()`. To use these functions, include the appropriate header files (e.g., `#include <stdio.h>` for `printf()`).
2. **User-Defined Functions** --> Functions created by the programmer to perform specific tasks within the program.

Function Parameters and Return Values

Functions can be categorized based on their parameters and return values:

- **No parameters and no return value**:

  ```
  void displayMessage() {
      printf("Hello, World!\n");
  }
  ```

- **Parameters but no return value**:

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

- **No parameters but returns a value**:

  ```
  int getNumber() {
      return 42;
  }
  ```

- **Parameters and returns a value**:

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

Understanding these combinations allows for flexible function design tailored to specific needs.

Inline Functions

In C, an inline function is a special type of function whose function call is replaced with the actual code of the function rather than being invoked through the usual function call mechanism, potentially improving performance by reducing function call overhead. It is declared using the `inline` keyword and is generally used for small and frequently used functions.

Example:

```
inline int square(int x) {
    return x * x;
}
```

In this example, calls to `square(y)` may be replaced with `y * y` during compilation, eliminating the function call overhead.

Conclusion

Functions are fundamental to structured programming in C, promoting modularity, reusability, and maintainability. By understanding how to declare, define, and utilize functions, programmers can write efficient and organized code.

Revision #3
Created 11 February 2025 23:41:05 by YP
Updated 11 February 2025 23:52:05 by YP