

MODULE 4

User defined functions

Functions

Function: A large program can be divided into manageable pieces called modules, where each module does a specific task. These modules are called as **function**. Functions are small programs that carry out some specific task.

Function Classification

1. **Library / Predefined functions**
2. **User defined functions (UDFs)**

Library / Predefined Functions

The library has a collection of various functions which perform **standard & pre-defined tasks**. The implementation details of these functions are **not known** to the programmers. The Library functions are also called as **pre-defined, built-in, inbuilt, or Standard functions**.

Various types of libraries are:

Library Name	Header file	Example functions
Math library	<math.h>	ex: <code>sqrt()</code> , <code>pow(a, b)</code> , <code>fabs(n)</code> etc
Standard I/O library	<stdio.h>	ex: <code>printf()</code> , <code>scanf()</code> .
String library	<string.h>	<code>strcpy()</code> , <code>strcmp()</code> etc.

User Defined Functions (UDFs)

Using only library functions, it is not possible to write program to achieve some tasks. In such cases, the programmers need to write their own functions to achieve the required task. These functions which are written by the programmers/users to do some specific task are called **user defined functions**.

Suppose we want to compute square of a number, the library function to do the task is not available. So, if we write a function `square()`, then the function `square()` is called a **UDF**.

Elements of User Defined Functions

1. **Function definition**
2. **Function call**
3. **Function declaration / Function prototype**

Function Definition

A function definition, also known as function implementation, shall include the following elements:

1. Function name
2. Function type
3. List of parameters
4. Local variable declarations
5. Function statements
6. A return statement

All the six elements are grouped into two parts, namely:

- **Function header** (first three elements)
- **Function body** (second three elements)

Syntax

Function Header

Function header consists of three parts: the function type, the function name and the formal parameter list.

The Function Type

Specifies the type of value that the function is expected to return to the program calling the function. If the return type is not explicitly specified, C will assume that it is an integer type. If the function is not returning anything, then we need to specify the return type as `void`.

The Function Name

Any C identifier and therefore must allow the same rules of formation as other variable names in C.

Formal Parameter List

The parameter list declares the variables that will receive the data sent by the calling program. They serve as input data to the function to carry out the specified task. They represent actual input values; they are referred to as formal parameters.

The parameter list contains declaration of variables separated by commas and surrounded by parentheses.

Example

```
int factorial(int n)
{
}

```

Function Body

The function body contains the declarations and statements necessary for performing the required task.

1. Local declarations that specify the variables needed by the function
2. Function statements that perform the task of the function
3. A return statement that returns the value evaluated by the function

Example

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

```

Return Values and Their Types

A function may or may not send back any value to the calling function. If it does, it is done through the return statement.

Syntax

```
return;           // it does not return any value
```

or

```
return(expression); // it returns the value of the expression
```

Function Calls

A function can be called by simply using the function name followed by a list of actual parameters, if any, enclosed in parentheses.

Example

```
void main()
{
    int a, b, c;
    c = add(10, 5);    // function call
    printf("%d", c);
}
```

Function Declaration

All functions in a C program must be declared before they are invoked. A function declaration consists of four parts:

1. Function type
2. Function name
3. Parameter list
4. Terminating semicolon

Syntax

```
function-type function-name(parameter list);
```

Rules

1. The parameter list must be separated by commas
2. The parameter names don't need to be the same in the prototype declaration and the function definition
3. The types must match the types of parameters in the function definition, in number and order
4. Use of parameter names in the declaration is optional
5. If the function has no formal parameters, the list is written as (void)
6. The return type is optional when the function returns `int` type data
7. The return type is `void` if no value is returned
8. When the declared types don't match with the types in the function definition, the compiler will produce an error

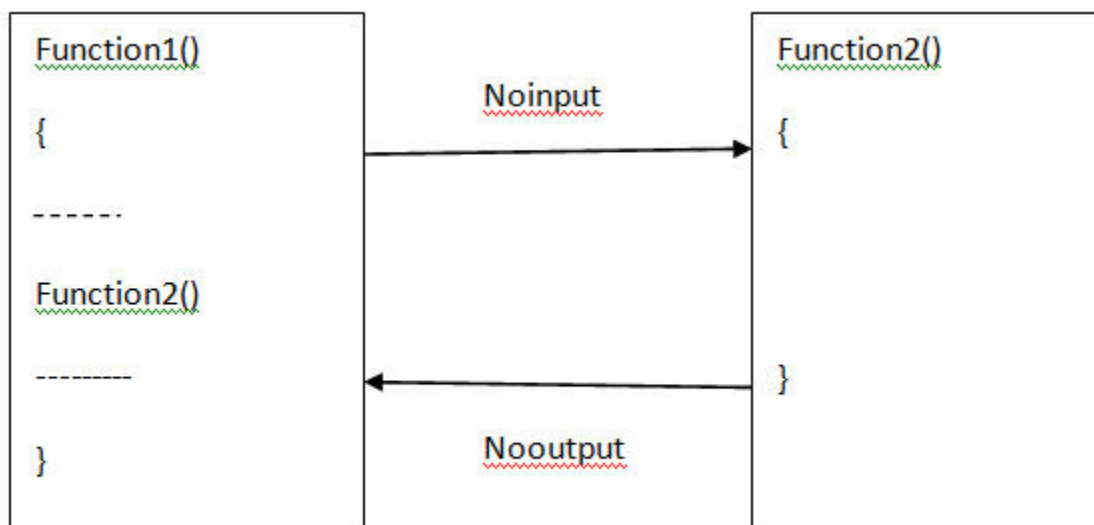
Category of Functions

1. Functions with no arguments and no return values
2. Functions with arguments and no return values
3. Functions with arguments and one return value
4. Functions with no arguments and return value

Functions with No Arguments and No Return Values

When a function has no arguments, it does not receive any data from the calling function. Similarly, it doesn't return any value.

The calling function doesn't receive any data from the called function. So, **there is no communication** between calling and called functions.



Example

```
#include <stdio.h>
#include <conio.h>

void add();

void main()
{
    add();
    getch();
}
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

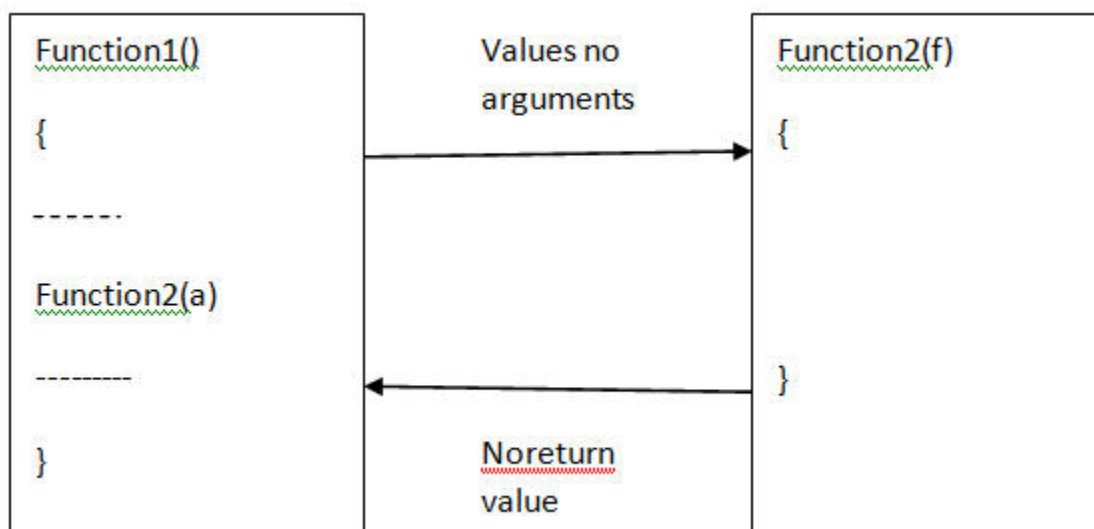
```
void add()
{
    int a, b, c;
    printf("Enter the value of a and b: ");
    scanf("%d%d", &a, &b);
    c = a + b;
    printf("%d", c);
}
```

Sample Output

```
Enter the value of a and b:
25 20
45
```

Functions with Arguments but No Return Values

The nature of data communication between the calling function and the called function with arguments but no return value.



Example

```
#include <stdio.h>
#include <conio.h>

void add(int a, int b);

void main()
{
    int a, b;
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

```
printf("Enter the value of a and b: ");
scanf("%d%d", &a, &b);
add(a, b);
getch();
}

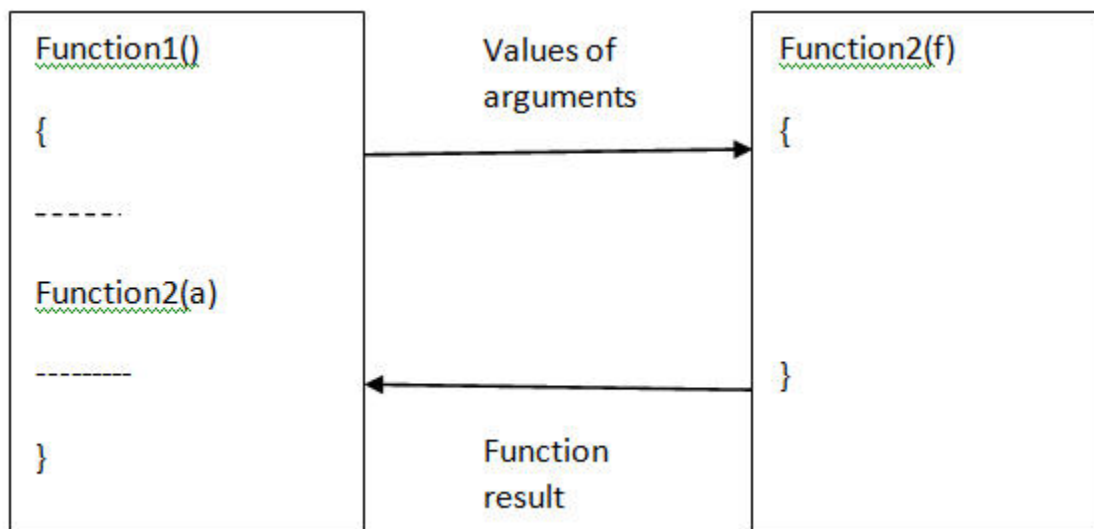
void add(int x, int y)
{
    int c;
    c = x + y;
    printf("%d", c);
}
```

Output

```
Enter the value of a and b:
25 20
45
```

Functions with Arguments and Return Values

The function receives a predefined form of inputs and outputs a desired value. Such functions have **two-way data communication** between calling and called functions.



Example

```
#include <stdio.h>
#include <conio.h>

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

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

```
void main()
{
    int a, b, c;
    printf("Enter the value of a and b: ");
    scanf("%d%d", &a, &b);
    c = add(a, b);
    printf("%d", c);
    getch();
}

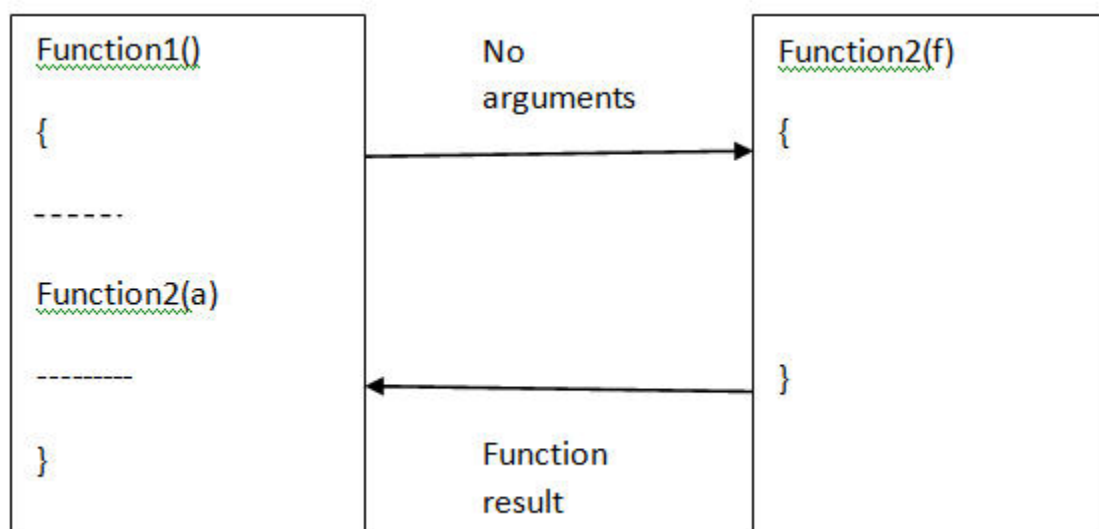
int add(int x, int y)
{
    int c;
    c = x + y;
    return c;
}
```

Output

```
Enter the value of a and b:
25 20
45
```

Functions with No Arguments but Returns a Value

Sometimes we may need to design functions that may not take any arguments but return a value to the calling function.



Example

```
#include <stdio.h>
#include <conio.h>

int add();

void main()
{
    int c;
    c = add();
    printf("%d", c);
    getch();
}

int add()
{
    int x, y, c;
    printf("Enter the value of a and b: ");
    scanf("%d%d", &x, &y);
    c = x + y;
    return c;
}
```

Output

```
Enter the value of a and b:
25 20
45
```

Nesting of Functions

C permits nesting of functions freely.

main can call function1, which calls function2, which calls function3, and so on.

There is no limit on how deeply functions can be nested.

Example

```
#include <stdio.h>
#include <conio.h>

void fun1();
void fun2();

void main()
{
    fun1();
    getch();
}
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

```
}  
  
void fun1()  
{  
    printf("Hai\t");  
    fun2();  
}  
  
void fun2()  
{  
    printf("Welcome");  
}
```

Output

Hai Welcome

