

3.7 STANDARD AND USER DEFINED FUNCTIONS:

A user-defined function is a type of function in C language that is defined by the user himself to perform some specific task.

It provides code reusability and modularity to our program. User-defined functions are different from built-in functions as their working is specified by the user and no header file is required for their usage.

a) How to use User-Defined Functions in C?

To use a user-defined function, we first have to understand the different parts of its syntax. The user-defined function in C can be divided into three parts:

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

C Function Prototype

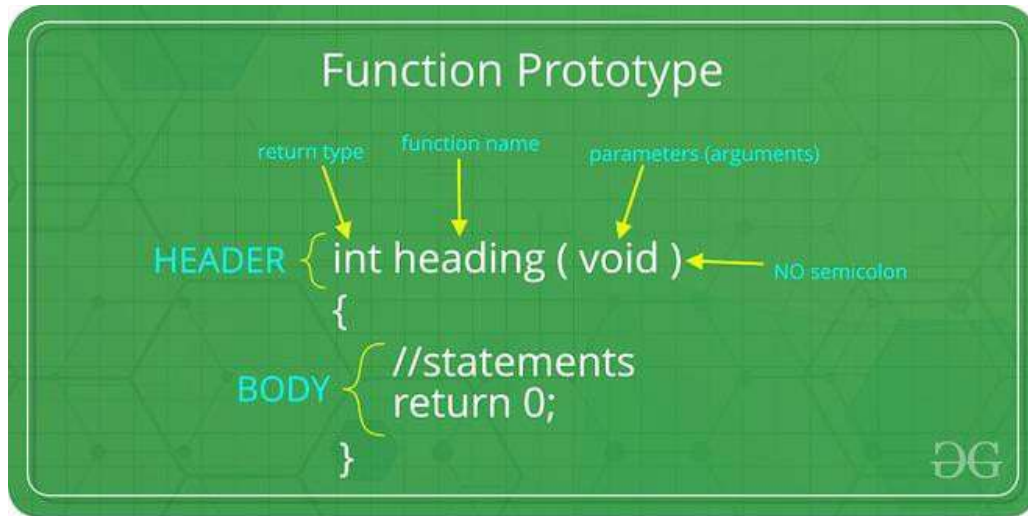
A function prototype is also known as a function declaration which specifies the function's name, function parameters, and return type. The function prototype does not contain the body of the function. It is basically used to inform the compiler about the existence of the user-defined function which can be used in the later part of the program.

Syntax

```
return_type function_name (type1 arg1, type2 arg2, ... typeN argN);
```

We can also skip the name of the arguments in the function prototype. So,

```
return_type function_name (type1 , type2 , ... typeN);
```



C Function Definition

Once the function has been called, the function definition contains the actual statements that will be executed. All the statements of the function definition are enclosed within { } braces.

Syntax

```

return_type function_name (type1 arg1, type2 arg2 .... typeN argN) {

    // actual statements to be executed
    // return value if any
}
  
```

C Function Call

In order to transfer control to a user-defined function, we need to call it. Functions are called using their names followed by round brackets. Their arguments are passed inside the brackets.

Syntax

```
function_name(arg1, arg2, ... argN);
```

Example of User-Defined Function


// C Program to illustrate the use of user-defined function

```
#include <stdio.h>

// Function prototype
int sum(int, int);

// Function definition
int sum(int x, int y)
{
    int sum;
    sum = x + y;
    return x + y;
}

// Driver code
int main()
{
    int x = 10, y = 11;
    // Function call
    int result = sum(x, y);
    printf("Sum of %d and %d = %d ", x, y, result);
    return 0;
}
```



Output

Sum of 10 and 11 = 21

b) Components of Function Definition

There are three components of the function definition:

1. Function Parameters
2. Function Body
3. Return Value

1. Function Parameters

Function parameters (also known as arguments) are the values that are passed to the called function by the caller. We can pass none or any number of function parameters to the function.

We have to define the function name and its type in the function definition and we can only pass the same number and type of parameters in the function call.

Example

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

Here, **a** and **b** are function parameters.

2. Function Body

The function body is the set of statements that are enclosed within { } braces. They are the statements that are executed when the function is called.

Example

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

Here, the statements between { and } is function body.

3. Return Value

The return value is the value returned by the function to its caller. A function can only return a single value and it is optional. If no value is to be returned, the return type is defined as void.

The return keyword is used to return the value from a function.

Syntax

```
return (expression);
```

Example

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

c) Passing Parameters to User-Defined Functions

We can pass parameters to a function in C using two methods:

1. Call by Value
2. Call by Reference

1. Call by value

In call by value, a copy of the value is passed to the function and changes that are made to the function are not reflected back to the values. Actual and formal arguments are created in different memory locations.


Example

// C program to show use of call by value

```
#include <stdio.h>

void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

// Driver code
int main()
{
    int x = 10, y = 20;
    printf("Values of x and y before swap are: %d, %d\n", x,y);
    swap(x, y);
    printf("Values of x and y after swap are: %d, %d", x,y);
    return 0;
}
```



Output

Values of x and y before swap are: 10, 20

Values of x and y after swap are: 10, 20

2. Call by Reference

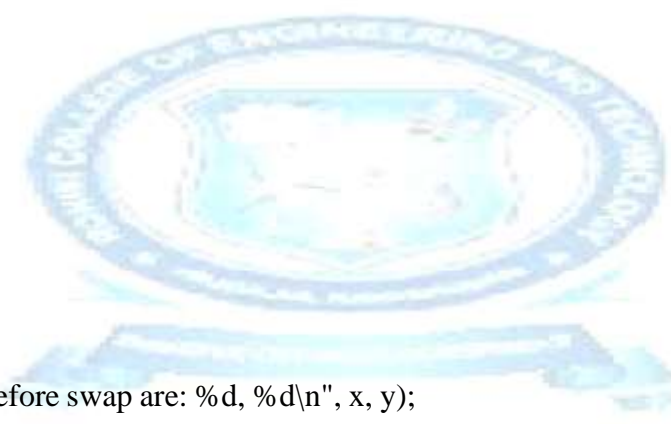
In a call by Reference, the address of the argument is passed to the function, and changes that are made to the function are reflected back to the values. We use the pointers of the required type to receive the address in the function.

Example

```
// C program to implement
// Call by Reference
#include <stdio.h>

void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Driver code
int main()
{
    int x = 10, y = 20;
    printf("Values of x and y before swap are: %d, %d\n", x, y);
    swap(&x, &y);
    printf("Values of x and y after swap are: %d, %d", x, y);
    return 0;
}
```



Output

Values of x and y before swap are: 10, 20

Values of x and y after swap are: 20, 10

Advantages of User-Defined Functions

The advantages of using functions in the program are as follows:

- One can avoid duplication of code in the programs by using functions. Code can be written more quickly and be more readable as a result.

- Code can be divided and conquered using functions. This process is known as Divide and Conquer. It is difficult to write large amounts of code within the main function, as well as testing and debugging. Our one task can be divided into several smaller sub-tasks by using functions, thus reducing the overall complexity.
- For example, when using pow, sqrt, etc. in C without knowing how it is implemented, one can hide implementation details with functions.
- With little to no modifications, functions developed in one program can be used in another, reducing the development time.

3.8 C STANDARD LIBRARY FUNCTIONS:

The Standard Function Library in C is a huge library of sub-libraries, each of which contains the code for several functions.

In order to make use of these libraries, link each library in the broader library through the use of header files.

The actual definitions of these functions are stored in separate library files, and declarations in header files. In order to use these functions, we have to include the header file in the program. Below are some header files with descriptions:

| S No. | Header Files | Description |
|-------|--------------|---|
| 1 | <assert.h> | It checks the value of an expression that we expect to be true under normal circumstances. If the expression is a nonzero value, the assert macro does nothing. |
| 2 | <complex.h> | A set of functions for manipulating complex numbers. |
| 3 | <float.h> | Defines macro constants specifying the implementation-specific properties of the floating-point library. |
| 4 | <limits.h> | These limits specify that a variable cannot store any value beyond these limits, for example- An unsigned character can store up to a maximum value of 255. |
| 5 | <math.h> | The math.h header defines various mathematical functions and one macro. All the Functions in this library take double as an argument and return double as the result. |
| 6 | <stdio.h> | The stdio.h header defines three variable types, several macros, and various function for performing input and output. |
| 7 | <time.h> | Defines date and time handling functions. |

| | | |
|---|------------|--|
| 8 | <string.h> | Strings are defined as an array of characters. The difference between a character array and a string is that a string is terminated with a special character '\0'. |
|---|------------|--|

