

Module 3

Iteration Constructs: Top Tested vs Bottom Tested - while - for - do while - Nesting of loops - skipping breaking loops. Arrays - 1D and 2D, 3 D - Case study: Developing solutions (flowcharts and algorithms) for problems using various iteration constructs - Comparative Study of various iteration constructs - Converting a solution C5 using one iteration construct with other iteration constructs. Functions and function calling mechanisms.

Iteration Constructs

In C programming, **iteration constructs** (or loops) allows you to repeatedly execute a block of code until the specified condition is met. It allows programmers to execute a statement or group of statements multiple times without repetition of code.

There are two primary types of iteration constructs based on when the loop condition is tested:

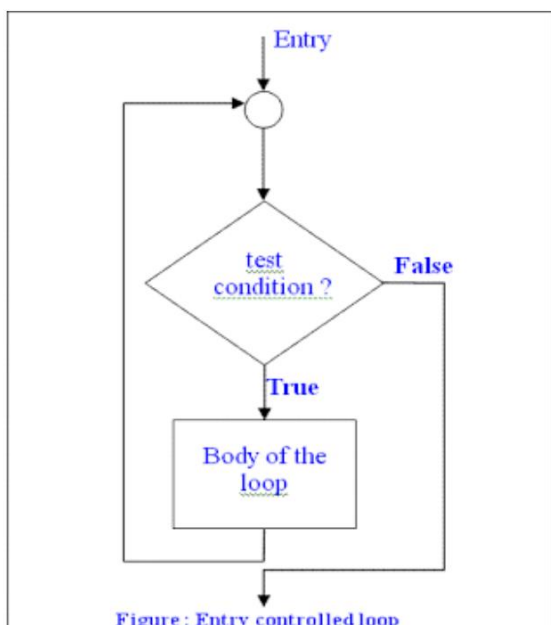
1. **Top-tested Loop** (Pre-condition Checked Loops/Entry Controlled Loops)
2. **Bottom-tested Loop** (Post-condition Checked Loops/Exit Controlled Loops)

1. Top-Tested Loops

In top-tested loops, the condition is checked **before** executing the body of the loop. If the condition is false at the start, body of the loop may never execute. It is also called '**pre-condition checked loops**' or '**entry controlled loop**'.

Example:

- **for** loop
- **while** loop



Nature of Execution:

- **Initialization:** Typically, initialization of the loop variable is done before the loop starts.
- **Condition Checking:** The condition is checked before the execution of the loop body.

- **Execution:** If the condition is true, the loop body is executed. If it's false, the loop is skipped entirely.

Advantages:

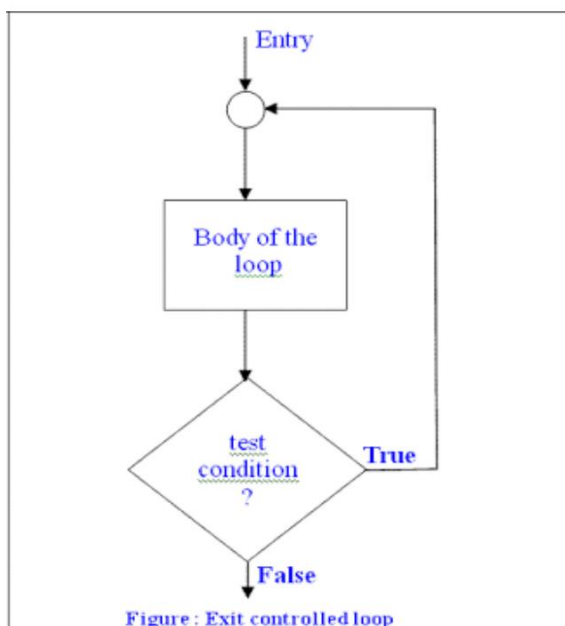
- Prevents the loop body from executing if the condition is false from the beginning.
- More straightforward in terms of readability and understanding, especially in scenarios where the initial condition is crucial for the loop execution.
- Best used when the number of iterations is known beforehand or when it's necessary to check the condition before executing the loop body.

2. Bottom-Tested Loops

In bottom-tested loops, the condition is checked **after** executing the body of the loop at least once. This means the loop body always executes at least once, irrespective of whether the condition is true or false. It is also called '**Post-condition Checked Loops**' or '**Exit Controlled Loops**'.

Example:

- **do-while loop**



Nature of Execution:

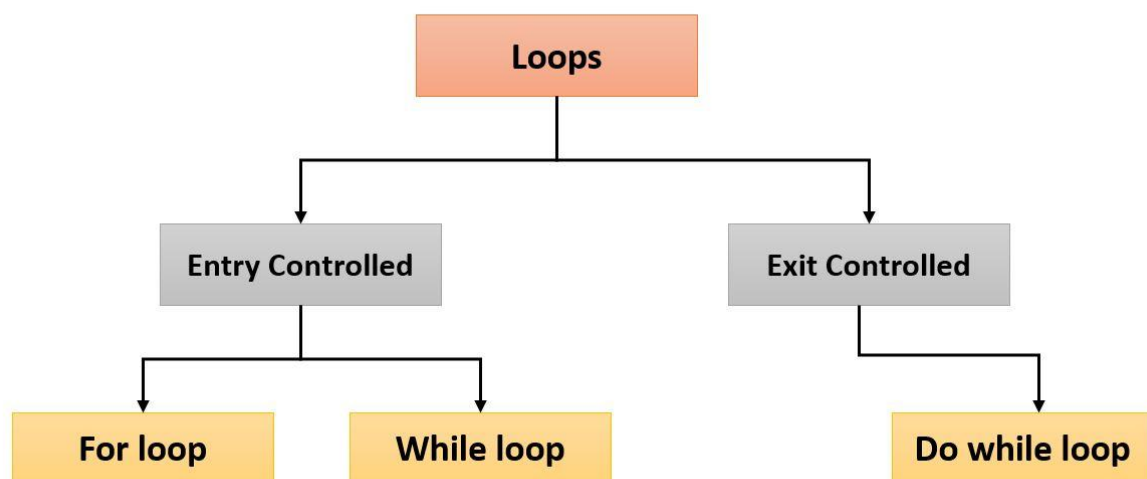
- **Initialization:** Like entry-controlled loops, initialization usually occurs before the loop.
- **Execution:** The loop body is executed first.
- **Condition Checking:** After the loop body is executed, the condition is checked. If true, the loop continues; if false, the loop ends.

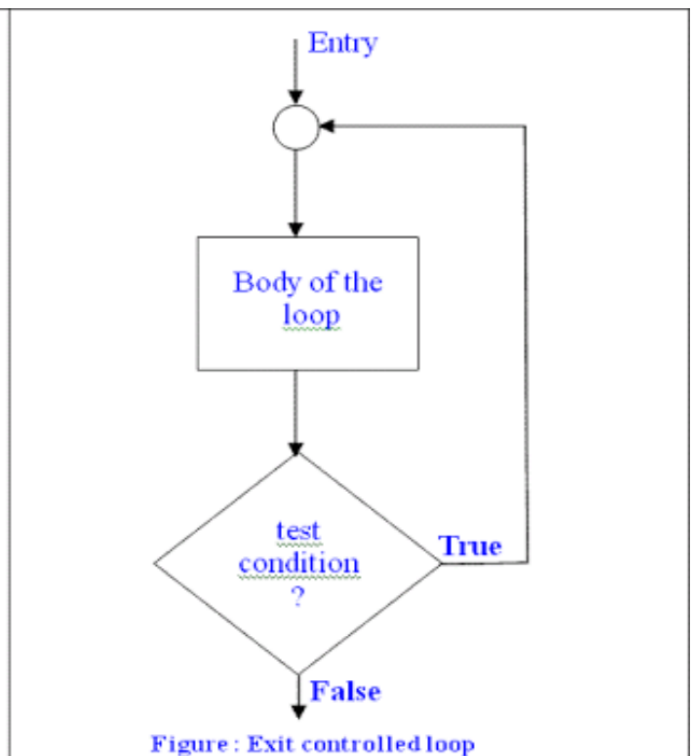
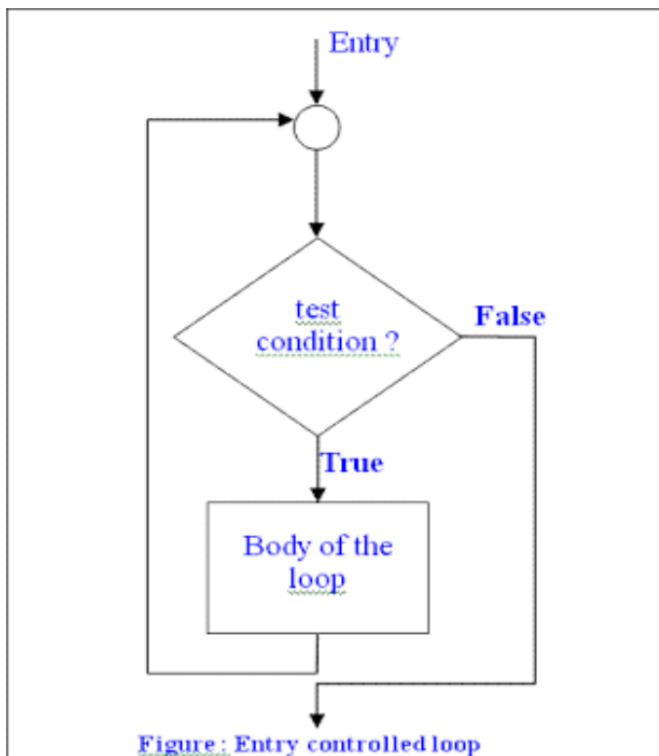
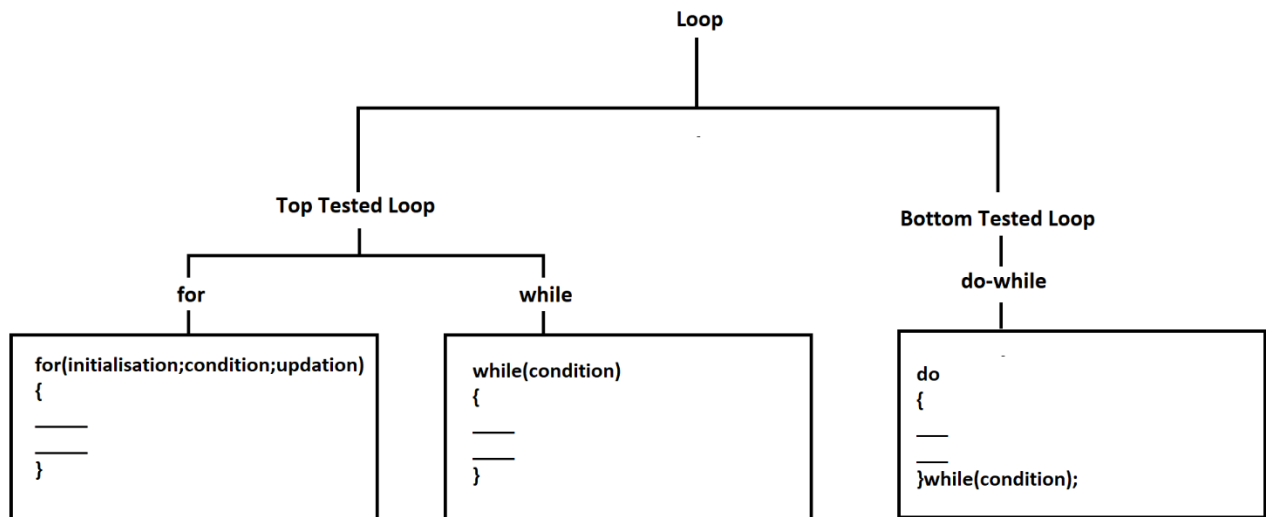
Advantages :

- Ensures that the loop body is executed at least once.
- It is useful in scenarios where the initial iteration needs to happen without condition checking, like initializing a process or displaying a menu.
- Ideal for situations where the loop must be executed at least once, such as when the loop body initializes or modifies the condition variable.

Top Tested vs Bottom Tested

Top Tested Loop	Bottom Tested Loop
Test condition is checked first, and then loop body will be executed.	Loop body will be executed first, and then condition is checked.
If Test condition is false, loop body will not be executed.	If Test condition is false, loop body will be executed once.
for loop and while loop are the examples of Entry Controlled Loop.	do while loop is the example of Exit controlled loop.
Used when checking of test condition is mandatory before executing loop body.	Used when checking of test condition is mandatory after executing the loop body.
Chosen when the initial condition is crucial.	Chosen when at least one execution is required, regardless of the condition.
Also called ' pre-condition checked loops ' or ' entry controlled loop '.	Also called ' Post-condition Checked Loops ' or ' Exit Controlled Loops '.





Loop Type	Description
for loop	first Initializes, then condition check, then executes the body and at last, the update is done.
while loop	first Initializes, then condition checks, and then executes the body, and updating can be inside the body.
do-while loop	do-while first executes the body and then the condition check is done.

for Loop

The **for** loop is a top-tested loop where the condition is checked before executing the loop body. It is often used when you know in advance how many times the loop should run.

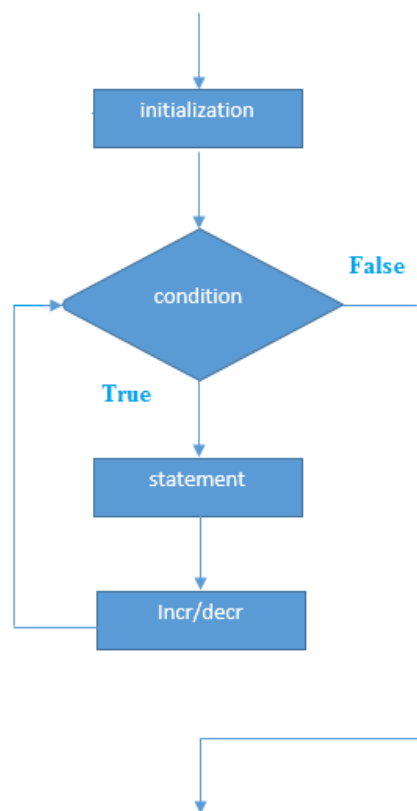
Syntax:

```
for (initialize expression; test expression; update expression)
{
    // body of for loop
}
```

In for loop, a loop variable is used to control the loop. Firstly we initialize the loop variable with some value, then check its test condition. If the statement is true then control will move to the body and the body of for loop will be executed. Then increment or decrement operation will be executed. Steps will be repeated till the exit condition becomes true. If the test condition will be false then it will stop.

1. Initialisation expression is executed.
2. The condition checking part is executed.
3. If the condition is **true** goto step 4, else goto step 7.
4. Body of the loop will be executed.
5. Updating expression will be executed.
6. Then steps 2 to 6 will be executed repeatedly.
7. exit from the loop.

Flow Diagram of for loop is as follows,



Example 1:

```
for(int i = 0; i < n; ++i)
{
    printf("Body of for loop which will execute till n");
}
```

- **Initialization Expression:** In this expression, we assign a loop variable or loop counter to some value. for example: `int i=1;`
- **Test Expression:** In this expression, test conditions are performed. If the condition evaluates to true then the loop body will be executed and then an update of the loop variable is done. If the test expression becomes false then the control will exit from the loop. for example, `i<=9;`
- **Update Expression:** After execution of the loop body loop variable is updated by some value it could be incremented, decremented, multiplied, or divided by any value.

Example 2:

```
for (int i = 0; i < 5; i++)
{
    printf("%d\n", i);
}
```

- **Flow of for loop:**
 1. **Initialization:** `int i = 0` is executed once.
 2. **Condition check:** `i < 5` is checked before every iteration.
 3. **Execution:** If the condition is true, the loop body (`printf`) executes.
 4. **Increment:** `i++` is executed after each iteration.

while Loop

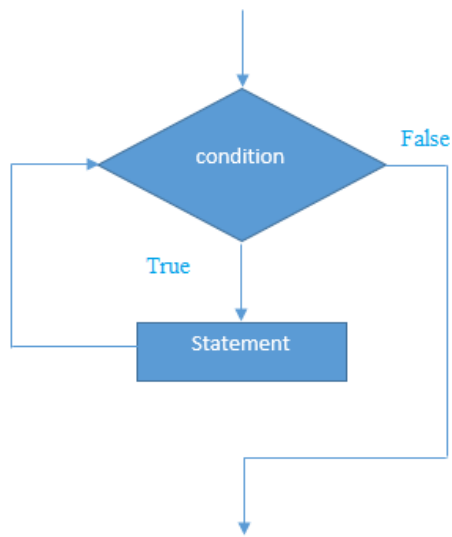
The while loop evaluates the test expression. If the test expression is true (nonzero), codes inside the body of while loop is evaluated. Then, again the test expression is evaluated. The process goes on until the test expression is false. When the test expression is false, the while loop is terminated.

While loop does not depend upon the number of iterations. In for loop the number of iterations was previously known to us but in the While loop, the execution is terminated on the basis of the test condition. If the test condition will become false then it will break from the while loop else body will be executed.

Syntax:

```
initialization_expression;
while (test_expression)
{
    // body of the while loop
    update_expression;
}
```

Flowchart of while loop is as shown below,



Example:

```
int i = 0;
while (i < 5)
{
    printf("%d\n", i);
    i++;
}
```

Flow of while loop:

1. **Condition check:** $i < 5$ is checked before each iteration.
2. **Execution:** If the condition is true, the loop body executes (printf and i++).

do-while Loop

The **do-while** loop ensures that the loop body is executed at least once before the condition is checked.

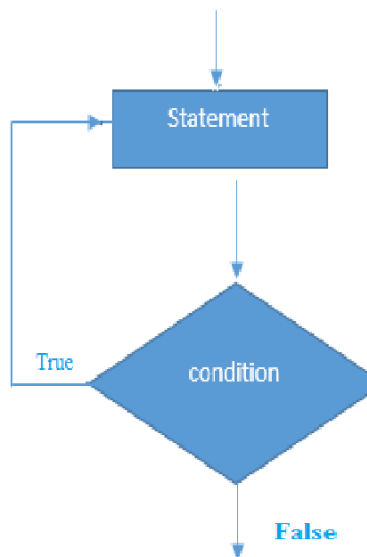
The do-while loop is similar to a while loop but the only difference lies in the do-while loop **test condition is tested at the end of the body**.

Syntax:

```
initialization;  
do  
{  
    // body of do-while loop  
    update_expression;  
} while (condition);
```

The code block (loop body) inside the braces is executed once. Then, the test expression is evaluated. If the test expression is true, the loop body is executed again. This process goes on until the test expression is evaluated to 0 (false). When the test expression is false (nonzero), the do...while loop is terminated.

Flow diagram of do-while loop



Example:

```
int i = 0;  
do {  
    printf("%d\n", i);  
    i++;  
} while (i < 5);
```

- **Flow of do-while loop:**

1. **Execution:** The loop body (printf and i++) executes first.
2. **Condition check:** After the loop body executes, $i < 5$ is checked.
3. If the condition is true, the loop repeats. If false, it terminates.

Infinite loop in C

An infinite loop is executed when the test expression never becomes false and the body of the loop is executed repeatedly. A program is stuck in an Infinite loop when the condition is always true. Mostly this is an error that can be resolved by using Loop Control statements.

Infinite for loop

To make a for loop infinite, we need not give any expression in the syntax. Instead of that, we need to provide two semicolons to validate the syntax of the for loop. This will work as an infinite for loop.

Syntax

```
for(;;)
{
    //statements
}
```

Example 1

```
#include<stdio.h>
void main ()
{
    for(;;)
    {
        printf("welcome infinite loop\n");
    }
}
```

Output

```
welcome infinite loop
welcome infinite loop
welcome infinite loop
welcome infinite loop
....
```

Example 2

```
#include <stdio.h>
int main ()
{
    int i;
    for ( ; ; )
    {
        i++;
        printf("This loop will run forever.\n");
    }

    return 0;
}
```

Output

```
This loop will run forever.
This loop will run forever.
This loop will run forever.
```

...

Infinite while loop

The following is the definition for **infinite while loop**,

```
while(1)
{
    // body of the loop
}
```

Example 1

```
#include <stdio.h>
int main()
{
    while (1)
        printf("This loop will run forever.\n");
    return 0;
}
```

Output

This loop will run forever.

This loop will run forever.

This loop will run forever.

...

Example 2

```
#include <stdio.h>
int main()
{
    int i=0;
    while(1)
    {
        i++;
        printf("i is :%d",i);
    }
    return 0;
}
```

Here, we have defined a while loop, which runs infinite times as it does not contain any condition. The value of 'i' will be updated an infinite number of times.

Infinite do..while loop

The **do..while** loop can also be used to create the infinite loop. The following is the syntax to create the infinite **do..while** loop.

```
do
{
    // body of the loop..
}while(1);
```

The above do..while loop represents the infinite condition as we provide the '1' value inside the loop condition. As we already know that non-zero integer represents the true condition, so this loop will run infinite times.

Example:

```
#include <stdio.h>
int main()
{
    do
    {
        printf("This loop will run forever.\n");
    } while (1);
    return 0;
}
```

Output

This loop will run forever.

This loop will run forever.

This loop will run forever.

...

Loop Control Statements

Loop control statements in C programming are used to change execution from its normal sequence.

Name	Description
break statement	The break statement is used to terminate the switch and loop statement. It transfers the execution to the statement immediately following the loop or switch.
continue statement	Continue statement skips the remainder body and immediately resets its condition before reiterating it.
goto statement	goto statement transfers the control to the labeled statement.

Continue statement

The **continue** statement skips the current iteration of the loop and continues with the next iteration. Its syntax is:

continue;

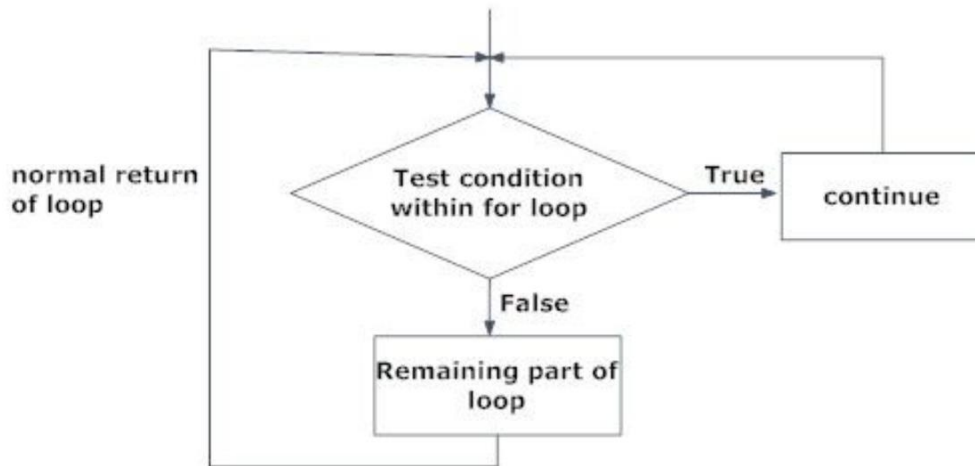


Fig: Flowchart of continue statement

<pre>while (testExpression) { // codes if (testExpression) { continue; } // codes }</pre>	<pre>do { // codes if (testExpression) { continue; } // codes } while (testExpression);</pre>
<pre>for (init; testExpression; update) { // codes if (testExpression) { continue; } // codes }</pre>	

Working of Continue in C

// C program to explain the use of continue statement.

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

```
int main()
{
    // for loop to print 1 to 8
    for (int i = 1; i <= 8; i++) {
        // when i = 4, the iteration will be skipped and for
        // will not be printed
        if (i == 4) {
            continue;
        }
        printf("%d ", i);
    }
    printf("\n");
    return 0;
}
```

```
}
```

Output

1 2 3 5 6 7 8

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

```
int main()
```

```
{
```

```
int i = 0;
```

```
    // while loop to print 1 to 8
```

```
    while (i < 8)
```

```
    {
```

```
        // when i = 4, the iteration will be skipped and 4 will not be printed
```

```
        i++;
```

```
        if (i == 4) {
```

```
            continue;
```

```
        }
```

```
        printf("%d ", i);
```

```
    }
```

```
    return 0;
```

```
}
```

Output

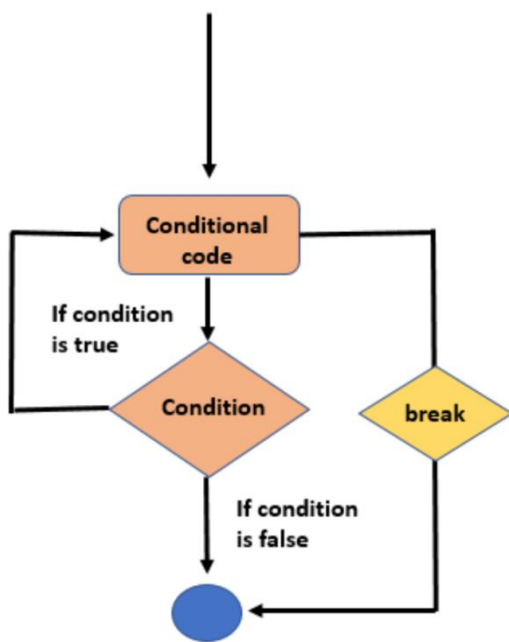
1 2 3 5 6 7 8

break statement

The break statement ends the loop immediately when it is encountered. Its syntax is:

break;

Figure – Flowchart of the break statement



<pre>while (testExpression) { // codes if (condition to break) { break; } // codes }</pre>	<pre>do { // codes if (condition to break) { break; } // codes } while (testExpression);</pre>
<pre>for (init; testExpression; update) { // codes if (condition to break) { break; } // codes }</pre>	

Working of break in C

// C Program to demonstrate break statement.

```
#include <stdio.h>  
int main()  
{  
    printf("break in for loop\n");  
    for (int i = 1; i < 5; i++)  
    {  
        if (i == 3)  
        {  
            break;  
        }  
        else
```

```

    {
        printf("%d ", i);
    }
}
return 0;
}

```

Output

break in for loop

1 2

```

#include <stdio.h>
int main()
{

    printf("\nbreak in while loop\n");
    int i = 1;
    while (i < 20)
    {
        if (i == 3)
            break;
        else
            printf("%d ", i);
        i++;
    }
    return 0;
}

```

Output

break in while loop

1 2

NESTED LOOP

Nested loops refer to placing one loop inside the body of another loop. Any type of loop can be nested within another loop, including for, while, and do-while loops.

Working of Nested Loops :

- The **inner loop** is executed completely every time the outer loop executes one iteration.
- The outer loop controls the number of times the inner loop runs.

Syntax of Nested Loops:

1. Nested for loop

```

for (initialization; condition; update)
{
    // Outer loop body
    for (initialization; condition; update)
    {
        // Inner loop body
    }
}

```

Example

```
#include <stdio.h>

int main() {
    int i, j;

    // Outer loop
    for (i = 1; i < 3; i++) {
        printf("Outer loop iteration: %d\n", i);

        // Inner loop
        for (j = 1; j < 3; j++) {
            printf("    Inner loop iteration: %d\n", j);
        }
    }

    return 0;
}
```

Output

```
Outer loop iteration: 1
    Inner loop iteration: 1
    Inner loop iteration: 2
Outer loop iteration: 2
    Inner loop iteration: 1
    Inner loop iteration: 2
```

2. Nested while loop

```
while(condition1)
{
    //outer loop body
    while(condition2)
    {
        //inner loop body
    }
}

#include <stdio.h>
int main() {
    int i = 1, j;
    while (i < 5)
    {
        j = 1;
        while (j < 3)
        {
            printf("%d\t", i * j);
            j++;
        }
        printf("\n");
        i++;
    }
    return 0;
}
```

Output

```
1    2
2    4
3    6
```


3. Nested do-while loop

```
do
{
    //outer loop body
    do
    {
        //inner loop body
    }while(condition2);
}while(condition1);
```

Example:

```
#include <stdio.h>
int main() {
    int i = 1, j;
    do {
        j = 1;
        do {
            printf("x ");
            j++;
        } while (j <= i);
        printf("\n");
        i++;
    } while (i <= 5);
    return 0;
}
```

Output

```
x
x x
x x x
x x x x
x x x x x
```

4. Nesting different types of loop

for (initialization; condition; update)

```
{
    // Outer loop body
    while(condition)
    {
        //inner loop body
    }
}
```

Example:

```
#include <stdio.h>
int main() {
    int i, j;

    for (i = 1; i <= 2; i++) {
        printf("For loop iteration: %d\n", i);

        j = 1;
        while (j <= 3) {
            printf(" While loop iteration: %d\n", j);
```

```

        j++;
    }
}

return 0;
}

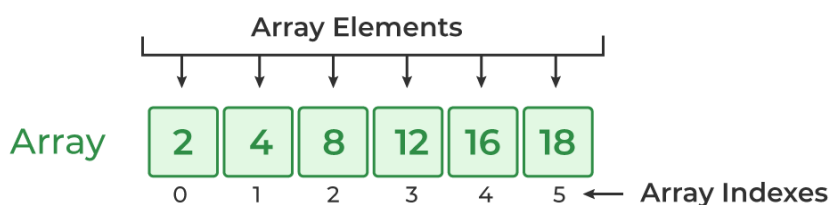
```

Output

For loop iteration: 1
 While loop iteration: 1
 While loop iteration: 2
 While loop iteration: 3
 For loop iteration: 2
 While loop iteration: 1
 While loop iteration: 2
 While loop iteration: 3

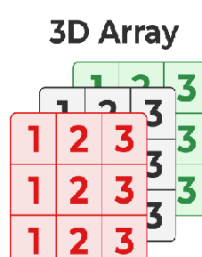
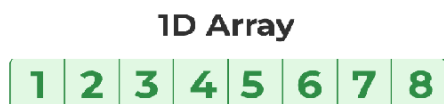
Array

An array is a fixed sized sequenced collection of elements of the same data type. They are stored in contiguous memory locations. It can be used to store the collection of primitive data types such as int, char, float, etc., and also derived and user-defined data types such as pointers, structures, etc.



Types of array

1. One dimensional array
2. Two dimensional array
3. Multi dimensional array



One-Dimensional Array

A one-dimensional (1D) array is a simple list or sequence of elements, where all elements are of the same type. It is a linear collection, accessed using a single index.

The indexing in the array always starts with 0, i.e., the **first element** is at index 0 and the **last element** is at $N - 1$ where N is the number of elements in the array.

```
data_type array_name [size];
```

example:

```
int arr[5] = {1, 2, 3, 4, 5}; // 1D array of 5 integers
```

Elements are accessed using a single index.

```
int first_element = arr[0]; // Accessing the first element (value = 1)
```

Two-Dimensional Array

A two-dimensional (2D) array is an array of arrays, structured like a matrix with rows and columns. It requires two indices for element access, one for the row and one for the column.

```
data_type array_name[rows][columns];
```

example

```
int matrix[3][3] = {  
    {1, 2, 3}, // Row 1  
    {4, 5, 6}, // Row 2  
    {7, 8, 9}  // Row 3  
};
```

Elements are accessed using two indices, one for the row and one for the column.

```
int value = matrix[1][2]; // Accessing the element at row 1, column 2 (value = 6)
```

Multi-Dimensional Array

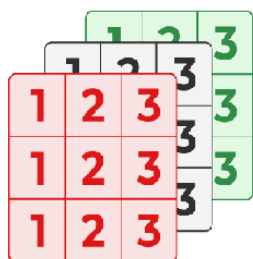
A multi-dimensional array is an array with more than two dimensions. It can be thought of as a higher-level structure, such as a cube for 3D arrays or more complex structures for higher dimensions.

```
data_type array_name[dimension1][dimension2]...[dimensionN];
```

One popular form of a multi-dimensional array is Three Dimensional Array or 3D Array. A 3D array has exactly three dimensions. It can be visualized as a collection of 2D arrays stacked on top of each other to create the third dimension.

```
array_name [size1] [size2] [size3];
```

3D Array



example

```
int arr3D[2][3][4] = { { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} }, // First 2D layer
                        { {13, 14, 15, 16}, {17, 18, 19, 20}, {21, 22, 23, 24} } // Second 2D layer
                      };
```

Elements are accessed using multiple indices, each corresponding to a specific dimension.

```
int value = arr3D[1][2][3]; // Accessing the element at 2nd layer, 3rd row, 4th column (value = 24)
```

Array Declaration

We can declare an array by specifying its name, the type of its elements, and the size of its dimensions. When we declare an array in C, the compiler allocates the memory block of the specified size to the array name.

Syntax of Array Declaration

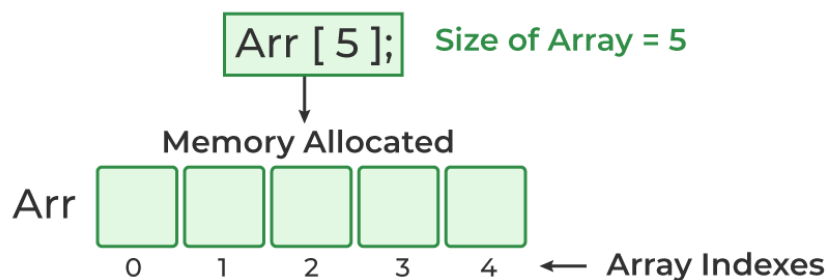
```
data_type array_name [size];
or
data_type array_name [size1] [size2]...[sizeN];
```

where N is the number of dimensions.

Example

```
int Arr[5]; // Declares an array of 5 integers
```

Array Declaration



```
// C Program to illustrate the array declaration
```

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

```
int main()
```

```
{
```

```
    int a[5];
```

```
    char b[5];
```

```
    return 0;
```

```
}
```

Array initialization

It is the process of assigning initial values to the elements of an array.

syntax for array initialization:

```
data_type array_name[size] = {value1,value2,...valueN};
```

```
data_type array_name[rowSize] [colSize] = {value1,value2,...valueN};
```

example

```
int arr[5] = { 1, 2, 3, 4, 5 }; // All 5 elements initialized
int arr[5] = { 1, 2 }; // Remaining 3 elements initialized to 0 ie Equivalent to: { 1, 2, 0, 0, 0 }
int arr[] = { 1, 2, 3, 4, 5 }; // Compiler determines the size as 5
int arr[5] = { 0 }; // All elements are initialized to 0
int matrix[2][3] = { { 1, 2, 3 }, // Row 1 { 4, 5, 6 } // Row 2 };
int matrix[2][3] = { 1, 2, 3, 4, 5, 6 }; // Equivalent to the above
```

// C Program to demonstrate array initialization

```
#include <stdio.h>
int main()
{
    int arr[5] = { 10, 20, 30, 40, 50 };
    int arr1[] = { 1, 2, 3, 4, 5 };
    float arr2[5];
    for (int i = 0; i < 5; i++)
    {
        arr2[i] = (float)i * 2.1;
    }
    return 0;
}
```

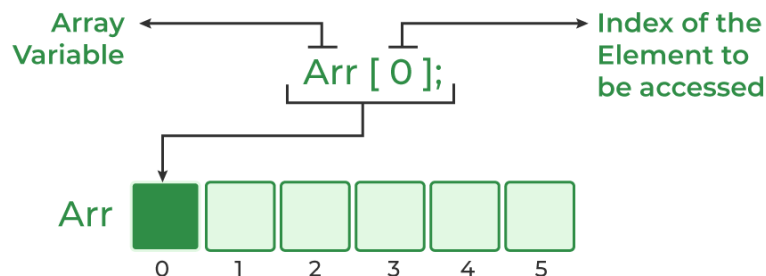
Accessing Array Elements

We can access any element of an array using the array subscript operator [] and the index value i of the element.

```
array_name [index];
```

```
array_name [rowIndex][colIndex];
```

The indexing in the array always starts with 0, i.e., the **first element** is at index **0** and the **last element** is at **N – 1** where N is the number of elements in the array.



// C Program to illustrate element access using array subscript

```
#include <stdio.h>
int main()
{
    // array declaration and initialization
    int arr[5] = { 15, 25, 35, 45, 55 };
}
```

```

// accessing element at index 2 i.e 3rd element
printf("Element at arr[2]: %d\n", arr[2]);

// accessing element at index 4 i.e last element
printf("Element at arr[4]: %d\n", arr[4]);

// accessing element at index 0 i.e first element
printf("Element at arr[0]: %d", arr[0]);

return 0;
}

```

Output

Element at arr[2]: 35

Element at arr[4]: 55

Element at arr[0]: 15

The following program demonstrates how to use an array in the C program:

```

// C Program to demonstrate the use of array
#include <stdio.h>

int main()
{
    // array declaration and initialization
    int arr[5] = { 10, 20, 30, 40, 50 };

    // modifying element at index 2
    arr[2] = 100;

    // traversing array using for loop
    printf("Elements in Array: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}

```

Output

Elements in Array: 10 20 100 40 50

Array of Characters (Strings)

A sequence of characters in the form of an array of characters terminated by a NULL character. These are called strings in C language.

// C Program to illustrate strings

```

#include <stdio.h>
int main()
{
    // creating array of character
    char arr[6] = { 'G', 'r', 'e', 'e', 'k', '\0' };

    // printing string
    int i = 0;

```

```

while (arr[i]) {
    printf("%c", arr[i++]);
}
return 0;
}

```

Output

Greek

// C Program to illustrate the 3d array

```

#include <stdio.h>
int main()
{
    // 3D array declaration
    int arr[2][2][2] = { 10, 20, 30, 40, 50, 60 };
    // printing elements
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            for (int k = 0; k < 2; k++)
            {
                printf("%d ", arr[i][j][k]);
            }
            printf("\n");
        }
        printf("\n \n");
    }
    return 0;
}

```

Output

10 20

30 40

50 60

0 0

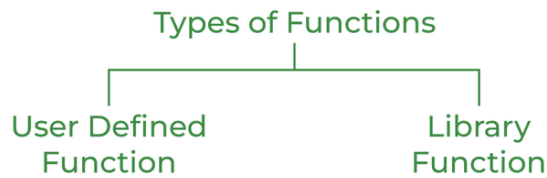
Functions

A function is a block of code enclosed within a curly braces that performs a specific task. A function can be called multiple times to provide reusability and modularity to the C program.

Types of functions

There are two types of function in C:

1. **Library functions** or **Built in functions**
2. **User-defined functions**



Library Functions/Built in functions:

A **built-in function** is a function that is provided as part of the C standard library or supported directly by the C compiler. These functions are already implemented and available for use. They can be used to perform common tasks, such as input/output operations, string manipulations, memory management, and mathematical calculations.

Example:

printf(), scanf(), pow(), sqrt(), strcmp(), strcpy(), gets(), puts(), ceil(), floor() etc.

The advantages of using built-in functions in C are:

1. You don't need to write the code for common tasks because the functions are ready to use.
2. These functions are well-tested and less prone to errors.
3. They ensure that common operations, like input/output or memory management, are handled in a standard way.

Example

```
#include <math.h>
#include <stdio.h>
int main()
{
    int Num;
    Num = 9;
    int squareRoot = sqrt(Num);
    printf("The Square root of %d = %d",    Num, squareRoot);
    return 0;
}
```

Output

The Square root of 9 = 3

User-defined functions:

These are functions which are created by the programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.

Advantages of User-Defined Functions

- **Modularity:** Breaking down a large program into smaller, manageable functions makes the code easier to understand, maintain, and debug.
- **Code Reusability:** Once a function is defined, it can be reused in different parts of the program.
- **Ease of Maintenance:** If a change is required, you only need to modify the function in one place, and it will affect all calls to that function, making the program easier to maintain and update.

- **Reduces Code Duplication:** Instead of writing the same block of code multiple times, you can write a function once and call it wherever needed. This reduces the size of the code and makes it less error-prone.
- **Divide and Conquer:** Large, complex problems can be broken down into smaller sub-problems, each solved with its own function. This makes overall problem more manageable.

Example

```
#include <stdio.h>
int sum(int a, int b)
{
    return a + b;
}
int main()
{
    int a = 30, b = 40;
    int res = sum(a, b);
    printf("Sum is: %d", res);
    return 0;
}
```

Output

Sum is: 70

Aspects related to C Functions

1. **Function Declaration**
2. **Function Definition**
3. **Function Calls**

Function Declarations/ Function Prototype

In a function declaration, we must provide,

- **function name**
- **return type**
- **number and type of its parameters**

A function declaration tells the compiler that there is a function with the given name defined somewhere else in the program. A function in C must always be declared globally before calling it.

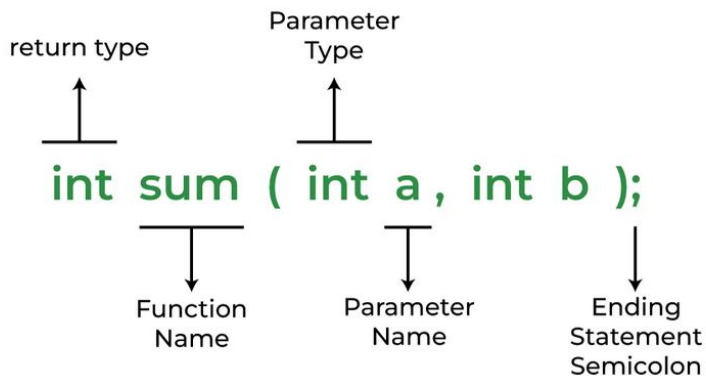
Syntax

```
return_type name_of_the_function (type parameter1, type parameter2);
```

The parameter name is not mandatory while declaring functions. We can also declare the function without using the name of the data variables.

Example

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



Function Definition

The function definition consists of block of code which are executed when the function is called. A C function is generally defined and declared in a single step because the function definition always starts with the function declaration so we do not need to declare it explicitly. The below example serves as both a function definition and a declaration.

```
return_type function_name (type parameter1, type parameter2)
{
    // body of the function
}
```

Example

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

Function Call

A **function call** is the process of invoking a function that has been defined earlier in the program. When you call a function, the program transfers control to the function's code, executes it, and then returns to the place where the function was called.

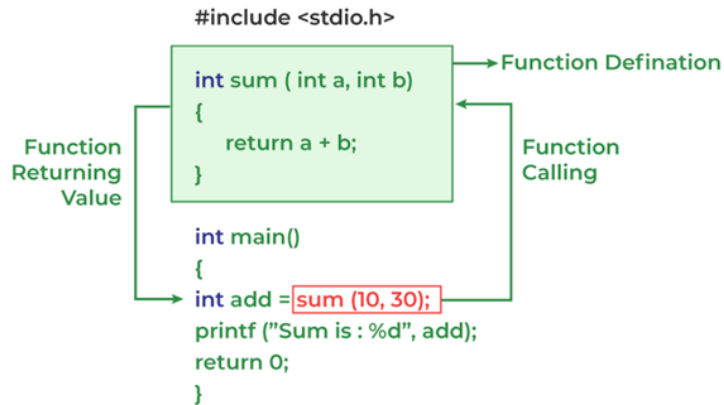
Syntax:

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

Example

```
sum(10,30);
add(x,y);
```

In the below example, the first sum function is called and 10,30 are passed to the sum function. After the function call sum of a and b is returned and control is also returned back to the main function of the program.



Example

```

#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;
}

```

Output

Sum is: 40

Function Return Type

Function return type tells what type of value is returned after the function is being executed. A function can return various types of values such as int, float, char, void. Only one value can be returned from a C function.

Example:

```
int func(parameter1,parameter2);
```

The above function will return an integer value after running statements inside the function.

Function Arguments/Function Parameter

Function Arguments are the data that is passed to a function. These values are then used inside the function to perform operations. We specify the types of the arguments in the function declaration.

Example:

```
int function_name(int var1, int var2);
```

There are two types of parameters.

4. Formal parameter
5. Actual Parameter

Formal Parameters

Formal parameters are the variables defined in a function declaration/definition.

In the function definition below, **a** and **b** are formal parameters:

```

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

```

```
}
```

Actual Parameters

Actual parameters (also known as arguments) are the actual values or expressions in the function call. They provide the data that the function needs to execute.

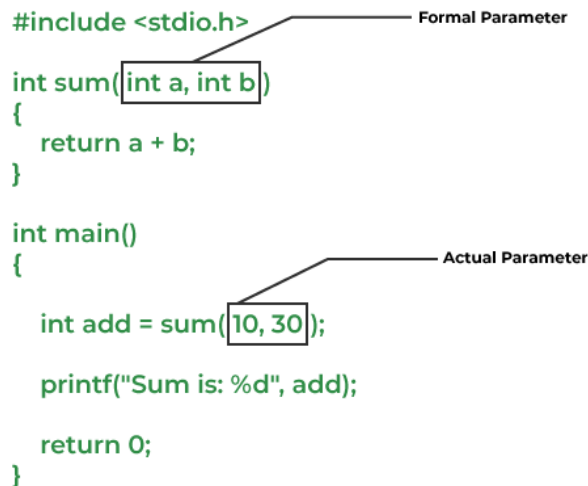
In the function call below, 5 and 3 are actual parameters:

```
int result = add(5, 3);
```

In the below program, 10 and 30 are known as actual parameters. Formal Parameters are the variable and the data type as mentioned in the function declaration. Here **a** and **b** are known as formal parameters.

```
#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;
}
```



Parameter passing Mechanism

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

1. Pass by Value
2. Pass by Reference

1. Pass by Value

In this method, it copies the value from actual parameters into formal parameters. The function works with this copy, thus any modifications to the parameter inside the function do not affect the original variable.

Example:

```
#include <stdio.h>
void swap(int var1, int var2)
{
    int temp = var1;
    var1 = var2;
    var2 = temp;
}

int main()
{
    int var1 = 3, var2 = 2;
    printf("Before swap Value of var1 and var2 is: %d, %d\n",
```

```

    var1, var2);
swap(var1, var2);
printf("After swap Value of var1 and var2 is: %d, %d",
    var1, var2);
return 0;
}

```

Output

Before swap Value of var1 and var2 is: 3, 2

After swap Value of var1 and var2 is: 3, 2

2. Pass by Reference

In pass by reference, instead of passing a copy of the actual parameter, the address (reference) of the variable is passed to the function. This means the function can modify the original variable directly.

Example:

```

#include <stdio.h>
void swap(int *var1, int *var2)
{
    int temp = *var1;
    *var1 = *var2;
    *var2 = temp;
}
int main()
{
    int var1 = 3, var2 = 2;
    printf("Before swap Value of var1 and var2 is: %d, %d\n",
        var1, var2);
    swap(&var1, &var2);
    printf("After swap Value of var1 and var2 is: %d, %d",
        var1, var2);
    return 0;
}

```

Output

Before swap Value of var1 and var2 is: 3, 2

After swap Value of var1 and var2 is: 2, 3

Types of Functions based on Return Types and Arguments

1. Function with no arguments and no return value
2. Function with no arguments and with return value
3. Function with argument and with no return value
4. Function with arguments and with return value

Function with no arguments and no return value

A function with no arguments and no return value can be defined using void. Such functions are useful for tasks that do not require input parameters and do not need to return a result.

Example

```

#include <stdio.h>
void greet() //Function definition with no argument and no return type.

```

```

{
    printf("Hello, welcome to the program!\n");
}
int main()
{
    greet();
    return 0;
}

```

Function with No Arguments and With Return Value

A function that does not take any parameters but returns a value. The return type is specified before the function name.

```

#include <stdio.h>
int printSum()
{
    int a=10,b=5,c;
    c=a+b;
    return c;
}
int main()
{
    int sum;
    sum=printSum();
    printf(" Sum is %d ",sum);
    return 0;
}

```

Function with Arguments and With No Return Value

A function that takes one or more parameters but does not return a value. The return type is void.

```

#include <stdio.h>
void printSum(int a, int b)
{
    printf("The sum is: %d\n", a + b);
}
int main()
{
    printSum(5, 3);
    return 0;
}

```

3. Function with Arguments and With Return Value

A function that takes one or more parameters and returns a value. The return type is specified before the function name.

Example:

```
#include <stdio.h>
// Function that takes two integers and returns their sum
int add(int a, int b)
{
    return a + b;
}

int main()
{
    // Call the add function with two arguments and store the returned value
    int result = add(5, 3);
    printf("The sum is: %d\n", result); // Output: The sum is: 8

    return 0;
}
```

Working of C Function

1. **Declaring a function:** Define the return types and parameters of the function.
2. **Defining a function:** The block of code that performs a specific task.
3. **Calling the function:** Calling the function by passing the arguments in the function.
4. **Executing the function:** Run all the statements inside the function to get the final result.
5. **Returning a value:** The calculated value after the execution of the function is returned.

```
#include <stdio.h>

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

int main() {
    int result = add(5, 3); // Function call
    printf("Sum: %d\n", result); // Output: Sum: 8
    return 0;
}

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

Advantages of Functions in C

1. The function can reduce the repetition of the same statements in the program.
2. The function makes code readable by providing modularity to our program.
3. There is no fixed number of calling functions it can be called as many times as you want.
4. The function reduces the size of the program.
5. Once the function is declared you can just use it without thinking about the internal working of the function.

Disadvantages of Functions in C

1. Cannot return multiple values.

2. Memory and time overhead due to stack frame allocation and transfer of program control.