Unit II

Conditional Program Execution: if, if-else, and nested if-else statements, Switch statements, Restrictions on switch values, Use of break and default with switch, Comparison of switch and if-else.

Loops and Iteration: for, while and do-while loops, Multiple loop variables, Nested loops, Assignment operators, break and continue statement.

Functions: Introduction, Types, Declaration of a Function, Function calls, Defining functions, Function Prototypes, Passing arguments to a function Return values and their types, Writing multifunction program, Calling function by value, Recursive functions.

Calling function by value, Recursive functions.

Control flow statements in C are used to manage the order in which statements are executed in a program. They allow you to make decisions, create loops, and control the flow of execution based on certain conditions.

C provides three types of control flow statements.

1. Decision Making statements.
    - if statements
        - I. if
        - II. if else
        - III. if else if ladder
        - IV. nested if
    - switch statement
2. Loop statements
    - do while loop
    - while loop
    - for loop
    - for-each loop
3. Jump statements
    - break statement
    - continue statement

Decision-making statements : As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in C, i.e., If statement and switch statement.

If Statement

The if statement is used to check some given condition and perform some operations depending upon the correctness of that condition. It is mostly used in the scenario where we need to perform the different operations for the different conditions. The syntax of the if statement is given below.

```
if(expression){
//code to be executed
}
```

```
#include <stdio.h>
int main()
{
   int x = 20;
   int y = 22;
   if (x<y)
   {
      printf("Variable x is less than y");
   }
   return 0;
}
```

Program to find the largest number of the three.

```
#include <stdio.h>
int main()
{
   int a, b, c;
    printf("Enter three numbers?");
   scanf("%d %d %d",&a,&b,&c);
   if(a>b && a>c)
   {
      printf("%d is largest",a);
   }
   if(b>a  && b > c)
   {
      printf("%d is largest",b);
   }
   if(c>a && c>b)
   {
      printf("%d is largest",c);
   }
   if(a == b && a == c)
   {
      printf("All are equal");
   }
}
```

Output

```
Enter three numbers?
12 23 34
34 is largest
```

If-else Statement

The if-else statement is used to perform two operations for a single condition. The if-else statement is an extension to the if statement using which, we can perform two different operations, i.e., one is for the correctness of that condition, and the other is for the incorrectness of the condition. Here, we must notice that if and else block cannot be executed simiulteneously. Using if-else statement is always preferable since it always invokes an otherwise case with every if condition. The syntax of the if-else statement is given below.

```
if(expression){
//code to be executed if condition is true
}else{
//code to be executed if condition is false
}
```

```
#include<stdio.h>
int main(){
int number=0;
printf("enter a number:");
scanf("%d",&number);
if(number%2==0){
printf("%d is even number",number);
}
else{
printf("%d is odd number",number);
}
return 0;
}
```

Output

```
enter a number:4
4 is even number
enter a number:5
5 is odd number
```

Program to check whether a person is eligible to vote or not.

```
#include <stdio.h>
int main()
{
   int age;
   printf("Enter your age?");
   scanf("%d",&age);
   if(age>=18)
   {
      printf("You are eligible to vote...");
   }
   else
   {
      printf("Sorry ... you can't vote");
   }
}
```

If else-if ladder Statement

The if-else-if ladder statement is an extension to the if-else statement. It is used in the scenario where there are multiple cases to be performed for different conditions. In if-else-if ladder statement, if a condition is true then the statements defined in the if block will be executed, otherwise if some other condition is true then the statements defined in the else-if block will be executed, at the last if none of the condition is true then the statements defined in the else block will be executed. There are multiple else-if blocks possible. It is similar to the switch case statement where the default is executed instead of else block if none of the cases is matched.

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```

Program to calculate the grade of the student according to the specified marks.

```
#include <stdio.h>
int main()
{
   int marks;
   printf("Enter your marks?");
   scanf("%d",&marks);
   if(marks > 85 && marks <= 100)
   {
      printf("Congrats ! you scored grade A ...");
   }
   else if (marks > 60 && marks <= 85)
   {
      printf("You scored grade B + ...");
   }
   else if (marks > 40 && marks <= 60)
   {
      printf("You scored grade B ...");
   }
   else if (marks > 30 && marks <= 40)
   {
      printf("You scored grade C ...");
   }
   else
   {
      printf("Sorry you are fail ...");
   }
}
```

Output

```
Enter your marks?10
Sorry you are fail ...
Enter your marks?40
You scored grade C ...
Enter your marks?90
Congrats ! you scored grade A ...
```

Nested if else statement in C

One of the fundamental constructs in programming is ***conditional statements***. They allow a program to take different paths based on the values of certain conditions. In C, conditional statements are implemented using ***if-else statements***. In more complex scenarios, ***nested if-else statements*** can be used to make more sophisticated decisions. This blog post will provide an in-depth explanation of nested if-else statements in C, including syntax, example, and output.

Syntax:

A ***nested if-else statement*** is an ***if statement*** inside another ***if statement***. The general syntax of nested if-else statement in C is as follows:

```
if (condition1) {
   /* code to be executed if condition1 is true */
   if (condition2) {
      /* code to be executed if condition2 is true */
   } else {
      /* code to be executed if condition2 is false */
   }
} else {
   /* code to be executed if condition1 is false */
}
```

```
#include<stdio.h>

void main()
{
    int x=20,y=30;

    if(x==20)
    {
        if(y==30)
        {
            printf("value of x is 20, and value of y is 30.");
        }
    }
}
```

C Switch Statement

The switch statement in C is an alternate to if-else-if ladder statement which allows us to execute multiple operations for the different possibles values of a single variable called switch variable. Here, We can define various statements in the multiple cases for the different values of a single variable.

The syntax of switch statement in c language is given below:

```
switch(expression){
case value1:
 //code to be executed;
 break;  //optional
case value2:
 //code to be executed;
 break;  //optional
......

default:
 code to be executed if all cases are not matched;
}
```

Rules for switch statement in C language

1) The *switch expression* must be of an integer or character type.

2) The *case value* must be an integer or character constant.

3) The *case value* can be used only inside the switch statement.

4) The *break statement* in switch case is not must. It is optional. If there is no break statement found in the case, all the cases will be executed present after the matched case. It is known as *fall through* the state of C switch statement.

Let's try to understand it by the examples. We are assuming that there are following variables.

```
1. int x,y,z;
2. char a,b;
3. float f;
```

Valid Switch	Invalid Switch	Valid Case	Invalid Case
switch(x)	switch(f)	case 3;	case 2.5;
switch(x>y)	switch(x+2.5)	case 'a';	case x;
switch(a+b-2)		case 1+2;	case x+2;
switch(func(x,y))		case 'x'>'y';	case 1,2,3;

```
#include<stdio.h>
int main(){
int number=0;
printf("enter a number:");
scanf("%d",&number);
switch(number){
case 10:
printf("number is equals to 10");
break;
case 50:
printf("number is equal to 50");
break;
case 100:
printf("number is equal to 100");
break;
default:
printf("number is not equal to 10, 50 or 100");
}
return 0;
}
```

Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

1. for loop
2. while loop
3. do-while loop

Let's understand the loop statements one by one.

for loop in C

The **for loop in C language** is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like the array and linked list.

Syntax of for loop in C

The syntax of for loop in c language is given below:

```
for(Expression 1; Expression 2; Expression 3){
//code to be executed
}
```

C for loop Examples

Let's see the simple program of for loop that prints table of 1.

```
#include<stdio.h>
int main(){
int i=0;
for(i=1;i<=10;i++){
printf("%d \n",i);
}
return 0;
}
```

```
// Program to calculate the sum of first n natural numbers
// Positive integers 1,2,3...n are known as natural numbers

#include <stdio.h>
int main()
{
    int num, count, sum = 0;

    printf("Enter a positive integer: ");
    scanf("%d", &num);

    // for loop terminates when num is less than count
    for(count = 1; count <= num; ++count)
    {
        sum += count;
    }

    printf("Sum = %d", sum);

    return 0;
}
```

Output

```
Enter a positive integer: 10
Sum = 55
```

While loop in c :

In C programming, a **while** loop is a control flow statement that repeatedly executes a block of code as long as a specified condition is true. The general syntax of a **while** loop is as follows:

The syntax of the `while` loop is:

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

Example 1: while loop

```
// Print numbers from 1 to 5

#include <stdio.h>
int main() {
  int i = 1;

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

  return 0;
}
```

Do while loop in C :

In C programming, the **do-while** loop is a control flow statement that executes a block of code repeatedly as long as a specified condition is true. The key feature of a **do-while** loop is that the condition is checked after the execution of the loop block, ensuring that the block is executed at least once. The general syntax of a **do-while** loop is as follows:

```
do {

// Code to be executed

} while (condition);
```

```c
// Program to add numbers until the user enters zero

#include <stdio.h>
int main() {
  double number, sum = 0;

  // the body of the loop is executed at least once
  do {
    printf("Enter a number: ");
    scanf("%lf", &number);
    sum += number;
  }
  while(number != 0.0);

  printf("Sum = %.2lf",sum);

  return 0;
}
```

Output

```
Enter a number: 1.5
Enter a number: 2.4
Enter a number: -3.4
Enter a number: 4.2
Enter a number: 0
Sum = 4.70
```

In the C programming language, jump statements are used to control the flow of a program by transferring control to different parts of the code. There are three main types of jump statements in C: goto, break, and continue. Additionally, the return statement can be considered a type of jump statement as it transfers control out of a function.

Types of Jump Statement in C

There are 4 types of Jump statements in C language.

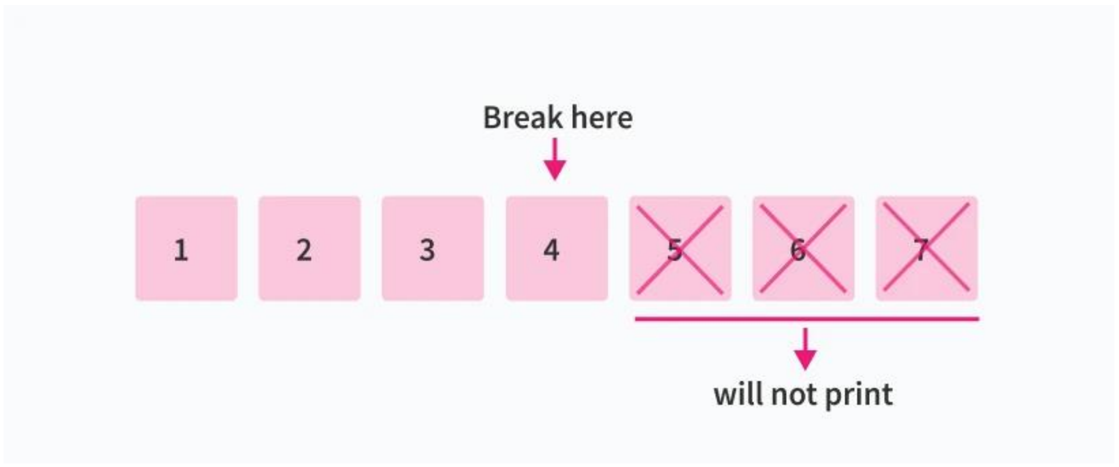
1. Break Statement
2. Continue Statement
3. Goto Statement
4. Return Statement.

Break Statement in C

Break statement exits the loops like for, while, do-while immediately, brings it out of the loop, and starts executing the next block. It also terminates the switch statement.

If we use the break statement in the nested loop, then first, the break statement inside the inner loop will break the inner loop. Then the outer loop will execute as it is, which means the outer loop remains unaffected by the break statement inside the inner loop.

The diagram below will give you more clarity on how the break statement works.



Syntax of Break Statement

The break statement's syntax is simply to use the break keyword.

Syntax:

```
//specific condition
break;
```

Example: How does Break Statement in C Works?

In a while loop:

The while loop repeatedly executes until the condition inside is true, so the condition in the program is the loop will execute until the value of a is greater than or equal to 10. But inside the while loop, there is a condition for the break statement: if a equals 3, then break. So the code prints the value of a as 1 and 2, and when it encounters 3, it breaks(terminates the while loop) and executes the print statement outside the while loop.

Code:

```
#include<stdio.h>

void main() {

  int a = 1; //initialize value to a

  while (a <= 10) // run loop unless the value is 10
  {
    if (a == 3) // if the value is 3
      break; //break the loop
    printf("Print=%d \n", a);
    a++;
  }

  printf("Outside loop"); //print statement outside the loop
}
```

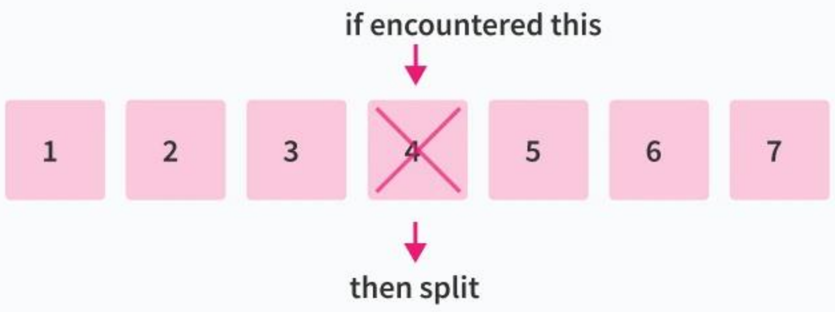
Output:

```
 Print=1
 Print=2
 Outside loop
```

Continue Statement in C

Continue in jump statement in C skips the specific iteration in the loop. It is similar to the break statement, but instead of terminating the whole loop, it skips the current iteration and continues from the next iteration in the same loop. It brings the control of a program to the beginning of the loop.

Using the continue statement in the nested loop skips the inner loop's iteration only and doesn't affect the outer loop.



Syntax of continue Statement

The syntax for the continue statement is simple continue. It is used below the specified continue condition in the loop.

Syntax:

```
continue;
```

Example: How does Continue Statement in C Works?

In the below program, when the continue condition gets true for the iteration i=3 in the loop, then that iteration is getting skips, and the control goes to the update expression of for loop
where i becomes 4, and the next iteration starts.

Code:

```
#include<stdio.h>

int main()
{
   int i;
   for ( i = 1; i <= 7; i++ )
   {
      if( i == 3) //continue condition
      {
         continue; //continue statement
      }

      printf("Value = %d \n",i);
   }
}
```

Output:

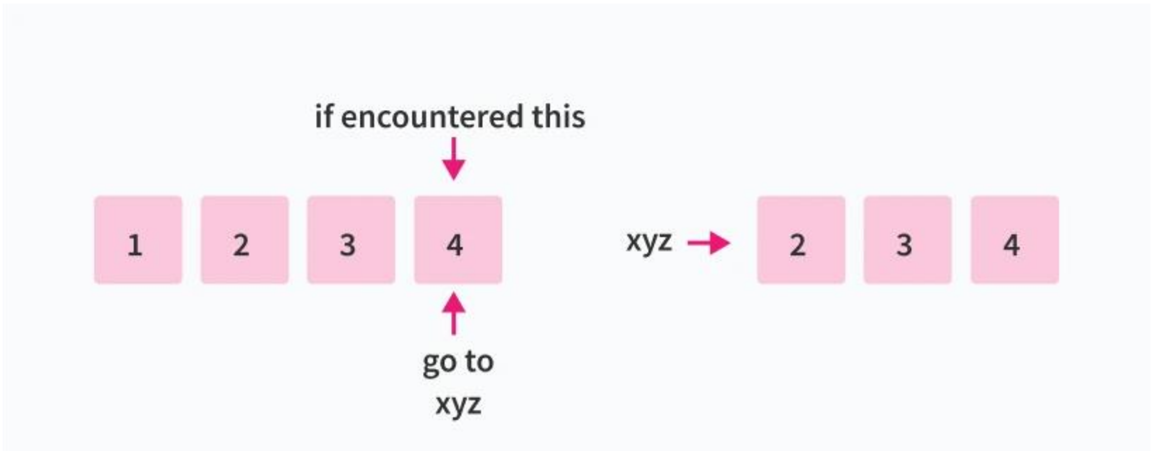
```
Value = 1
Value = 2
Value = 4
Value = 5
Value = 6
Value = 7
```

Goto Statement in C

Goto statement is somewhat similar to the continue statement in the jump statement in C, but the continue statement can only be used in loops, whereas Goto can be used anywhere in the program, but what the continue statement does it skips the current iteration of the loop and goes to the next iteration, but in the goto statement we can specify where the program control should go after skipping.

The concept of **label** is used in this statement to tell the program control where to go. The jump in the program that goto take is within the same function.

Below is the diagram; the label is XYZ, and it will be more clear when we'll see the syntax.



Syntax of goto Statement

Syntax:

```
goto label;
.
.
label:
--code--
--code--
```

Example: How does Goto Statement in C Works?

Check odd or even using goto:

In the program, the if statement has a logic of an even number. When it is satisfied, it will go to the Even label; otherwise, Odd Label.

Code:

```
#include<stdio.h>

int main() {

  int a;
  printf("\nEnter a Positive int:");
  scanf("%d", & a);

  if (a % 2 == 0) //logic of even no
    goto Even; //goto statement 1
  else
    goto Odd; //goto statement 2

  Even: // label 1
    printf("Number is Even\n");
  exit(0);

  Odd: //label2
    printf("Number is Odd\n");

  return 0;
}
```

Output:

```
Enter a Positive int:4
Number is Even
```

```
int i = 0;
start:
    if (i < 5) {
        printf("%d ", i);
        i++;
        goto start;
    }
```

Output:

```
0 1 2 3 4
```

Return Statement in C

The return statement is a type of jump statement in C which is used in a function to end it or terminate it immediately with or without value and returns the flow of program execution to the start from where it is called.

The function declared with void type does not return any value.
Syntax of Return Statement

Syntax:

```
return expression;
or
return;
```

Functions

Functions in C are the basic building blocks of a C program. A function is a set of statements enclosed within curly brackets ({}) that take inputs, do the computation, and provide the resultant output. You can call a function multiple times, thereby allowing reusability and modularity in C programming. It means that instead of writing the same code again and again for different arguments, you can simply enclose the code and make it a function and then call it multiple times by merely passing the various arguments.

Why Do We Need Functions in C Programming?

We need functions in C programming and even in other programming languages due to the numerous advantages they provide to the developer. Some of the key benefits of using functions are:

- Enables reusability and reduces redundancy
- Makes a code modular
- Provides abstraction functionality
- The program becomes easy to understand and manage
- Breaks an extensive program into smaller and simpler pieces

Basic Syntax of Functions

The basic syntax of functions in C programming is:

return_type function_name(arg1, arg2, … argn){

Body of the function //Statements to be processed

}

In the above syntax:

- return_type: Here, we declare the data type of the value returned by functions. However, not all functions return a value. In such cases, the keyword void indicates to the compiler that the function will not return any value.
- function_name: This is the function’s name that helps the compiler identify it whenever we call it.
- Body: The function’s body contains all the statements to be processed and executed whenever the function is called.

Note: The function_name and parameters list are together known as the signature of a function in C programming.

Aspects of Functions in C Programming

Functions in C programming have three general aspects: declaration, defining, and calling. Let’s understand what these aspects mean.

1. Function Declaration

The function declaration lets the compiler know the name, number of parameters, data types of parameters, and return type of a function. However, writing parameter names during declaration is optional, as you can do that even while defining the function.

2. Function Call

As the name gives out, a function call is calling a function to be executed by the compiler. You can call the function at any point in the entire program. The only thing to take care of is that you need to pass as many arguments of the same data type as mentioned while declaring the function. If the function parameter does not differ, the compiler will execute the program and give the return value.

3. Function Definition

It is defining the actual statements that the compiler will execute upon calling the function. You can think of it as the body of the function. Function definition must return only one value at the end of the execution.

Variable Scope

Variable scope means the visibility of variables within a code of the program.

In C, variables which are declared inside a function are local to that block of code and cannot be referred to outside the function. However, variables which are declared outside all functions are global and accessible from the entire program. Constants declared with a **#define** at the top of a program are accessible from the entire program. We consider the following program which prints the value of the global variable from both main and user defined function :

```
#include <stdio.h>
int global = 1348;
void test();
int main() {
  printf("from the main function : global =%d \n", global);
  test () ;
return 0;}

void test (){
printf("from user defined function : global =%d \n", global);}
```

Result:

```
from the main function : global =1348
from user defined function : global =1348
```

Parameter & Argument

1. **Parameter:**
    - A parameter is a variable or placeholder in a function's declaration or definition. It's a name used to represent a value that the function expects to receive when it's called.
    - Parameters are specified in the function's header (the function declaration or definition) and act as placeholders for the actual values that will be passed to the function when it's called.
    - Parameters define what kind of values a function can accept and use.

For example, in this function declaration, **a** and **b** are parameters:

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

2. **Argument:**

    - An argument is an actual value or expression that is passed to a function when it's called. Arguments are the real data that the function will work with.
    - Arguments are supplied in the function call, and they are assigned to the corresponding parameters within the function.
    - When a function is called, the arguments are used to populate the parameters defined in the function's declaration or definition.

For example, in this function call, **5** and **3** are arguments:

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

Different characteristics of functions :

A function may or may not accept any argument. It may or may not return any value. Based on these facts, There are four different characteristics of function calls.

- function without arguments and without return value
- function without arguments and with return value
- function with arguments and without return value
- function with arguments and with return value

Example for Function without argument and return value

Example 1

```
#include<stdio.h>
void printName();
void main ()  {
    printf("Hello ");
    printName();
}
void printName()  {
    printf("Mca Students");
}
```

Output

```
Hello Mca Students
```

Example 2

```
#include<stdio.h>
void sum();
void main()  {
    printf("\nGoing to calculate the sum of two numbers:");
    sum();
}
void sum()  {
    int a,b;
    printf("\nEnter two numbers");
    scanf("%d %d",&a,&b);
    printf("The sum is %d",a+b);
}
```

Output

```
Going to calculate the sum of two numbers:

Enter two numbers 10
24

The sum is 34
```

Example for Function without argument and with return value

Example 1

```
#include<stdio.h>
int sum();
void main()
{
   int result;
   printf("\nGoing to calculate the sum of two numbers:");
   result = sum();
   printf("%d",result);
}
int sum()
{
   int a,b;
   printf("\nEnter two numbers");
   scanf("%d %d",&a,&b);
   return a+b;
}
```

Output

```
Going to calculate the sum of two numbers:

Enter two numbers 10
24

The sum is 34
```

Example 2: program to calculate the area of the square

```
#include<stdio.h>
int sum();
void main()
{
   printf("Going to calculate the area of the square\n");
   float area = square();
   printf("The area of the square: %f\n",area);
}
int square()
```

```
{
   float side;
   printf("Enter the length of the side in meters: ");
   scanf("%f",&side);
   return side * side;
}
```

Output

```
Going to calculate the area of the square
Enter the length of the side in meters: 10
The area of the square: 100.000000
```

Example for Function with argument and without return value

Example 1

```
#include<stdio.h>
void sum(int, int);
void main()
{
   int a,b,result;
   printf("\nGoing to calculate the sum of two numbers:");
   printf("\nEnter two numbers:");
   scanf("%d %d",&a,&b);
   sum(a,b);
}
void sum(int a, int b)
{
   printf("\nThe sum is %d",a+b);
}
```

Output

```
Going to calculate the sum of two numbers:

Enter two numbers 10
24

The sum is 34
```

Example 2: Program to calculate the average of five numbers using function without return type and with arguments.

```
#include<stdio.h>

    void average(int, int, int, int, int);
    void main()  {
       int a,b,c,d,e;
       printf("\nGoing to calculate the average of five numbers:");
       printf("\nEnter five numbers:");
       scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);
       average(a,b,c,d,e);
    }
    void average(int a, int b, int c, int d, int e)
    {
       float avg;
       avg = (a+b+c+d+e)/5;
       printf("The average of given five numbers : %f",avg);
    }
```

Output

```
Going to calculate the average of five numbers:
Enter five numbers:10
20
30
40
50
The average of given five numbers : 30.000000
```

Example for Function with argument and with return value

Example 1

```
    #include<stdio.h>
    int sum(int, int);
    void main()
    {
       int a,b,result;
       printf("\nGoing to calculate the sum of two numbers:");
       printf("\nEnter two numbers:");
       scanf("%d %d",&a,&b);
    result = sum(a,b);
    printf("\nThe sum is : %d",result);
}
int sum(int a, int b)
{
    return a+b;
}
```

Output

```
Going to calculate the sum of two numbers:
Enter two numbers:10
20
The sum is : 30
```

Example 2: Program to check whether a number is even or odd

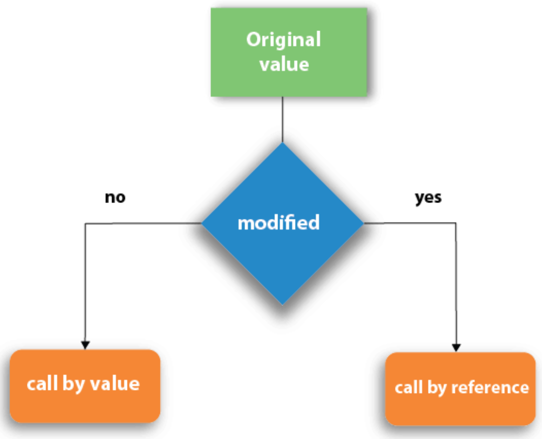
```
#include<stdio.h>
int even_odd(int);
void main()
{
 int n,flag=0;
 printf("\nGoing to check whether a number is even or odd");
 printf("\nEnter the number: ");
 scanf("%d",&n);
 flag = even_odd(n);
 if(flag == 0)
 {
   printf("\nThe number is odd");
 }
 else
 {
   printf("\nThe number is even");
 }
}
int even_odd(int n)
{
   if(n%2 == 0)
   {
      return 1;
   }
   else
   {
      return 0;
   }
}
```

Output

```
Going to check whether a number is even or odd
Enter the number: 100
The number is even
```

Call by value and Call by reference in C

There are two methods to pass the data into the function in C language, i.e., *call by value* and *call by reference*.



Let's understand call by value and call by reference in c language one by one.

Call by value in C

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we can not modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Let's try to understand the concept of call by value in c language by the example given below:

```
#include<stdio.h>
void change(int num) {
    printf("Before adding value inside function num=%d \n",num);
    num=num+100;
    printf("After adding value inside function num=%d \n", num);
}
int main() {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(x);//passing value in function
    printf("After function call x=%d \n", x);
return 0;
}
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

```
#include <stdio.h>

// Function that modifies the value through a pointer.
void modifyValue(int *x) {
    (*x)++;  // Increment the value at the memory location pointed to by x
}

int main() {
    int num = 5;

    printf("Before: %d\n", num);

    // Passing the address of 'num' to the function
    modifyValue(&num);

    printf("After: %d\n", num);

    return 0;
}
```

Call by Value Example: Swapping the values of the two variables

```
#include <stdio.h>
void swap(int , int); //prototype of the function
int main() {
    int a = 10;
    int b = 20;
    printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a an
d b in main
    swap(a,b);
    printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual parameters
 do not change by changing the formal parameters in call by value, a = 10, b = 20  }
void swap (int a, int b){
    int temp;
    temp = a;
    a=b;
    b=temp;
    printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a = 20,
b = 10
}
```

Output

```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 10, b = 20
```

Call by reference in C

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Consider the following example for the call by reference.

```
#include<stdio.h>
void change(int *num) {
    printf("Before adding value inside function num=%d \n",*num);
    (*num) += 100;
    printf("After adding value inside function num=%d \n", *num);
}
int main() {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(&x);//passing reference in function
    printf("After function call x=%d \n", x);
return 0;
}
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=200
```

Call by reference Example: Swapping the values of the two variables

```
#include <stdio.h>
void swap(int *, int *); //prototype of the function
int main() {
    int a = 10;
    int b = 20;
    printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a an
d b in main
    swap(&a,&b);
    printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual parameter
s do change in call by reference, a = 10, b = 20
}
void swap (int *a, int *b)
{
    int temp;
    temp = *a;
    *a=*b;
    *b=temp;
    printf("After swapping values in function a = %d, b = %d\n",*a,*b); // Formal parameters, a = 2
0, b = 10
}
```

Output

```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 20, b = 10
```

Difference between call by value and call by reference in c

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location.
4	Values of variables are passed by the simple technique.	Pointer variables are necessary to define to store the address values of variables.

What is Recursion in C?

Recursion is the process of calling a function itself repeatedly until a particular condition is met. A function that calls itself directly or indirectly is called a recursive function and such kind of function calls are called recursive calls.

Example of Recursion in C

```
#include <stdio.h>
int main() {
    // Write C code here
    printf("Hello world \n");
            main();
    return 0;
}
```

```
//Recursive function to calculate factorial.

#include <stdio.h>
int factorial(int n) {
    // Base case: factorial of 0 is 1
    if (n == 0 || n == 1) {
        return 1;
    } else {
        // Recursive case: n! = n * (n-1)!
        return n * factorial(n - 1);
    }
}

int main() {
    int num;
    printf("Enter a non-negative integer: ");
    scanf("%d", &num);

    // Check for negative input
    if (num < 0) {
        printf("Please enter a non-negative integer.\n");
    } else {
        printf("Factorial of %d = %d\n", num, factorial(num));
    }

    return 0;
}
```

//Recursive function to find fibonaci series..

```
 #include<stdio.h>
void printFibonacci(int n){
   static int n1=0,n2=1,n3;
   if(n>0){
       n3 = n1 + n2;
       n1 = n2;
       n2 = n3;
       printf("%d ",n3);
       printFibonacci(n-1);
   }
}
int main(){
   int n;
   printf("Enter the number of elements: ");
   scanf("%d",&n);
   printf("Fibonacci Series: ");
   printf("%d %d ",0,1);
   printFibonacci(n-2);//n-2 because 2 numbers are already printed
  return 0;
 }
```

Output:

```
Enter the number of elements:15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```