What is programming?

When we want a computer to perform a specific task, such as generating a marks sheet or a salary slip, we have to create a sequence of instructions in a logical order that a computer can understand and interpret. This sequence of instructions is called a program. The process of writing programs is called *programming*.

Programming languages can be divided into three types. They are:

1. Machine language

This is the basic language understood by a computer. This language is made up of 0's and 1's. A combination of these two digits represents characters, numbers, and/or instructions. Machine language is also referred to as *binary language.*

2. Assembly language

This language uses codes such as ADD, MOV, and SUB to represent instructions. These codes are called mnemonics. Though these codes have to be memorized, assembly language is much easier to use than machine language.

3. High-level languages

High-level languages such as BASIC, FORTRAN, C, C++, and JAVA are very much easier to use than machine language or assembly language because they have words that are similar to English.

A quick comparison of programming languages

	Machine Language	Assembly Language	High-level Languages
Time to execute	Since it is the basic language of the computer, it does not require any translation, and hence ensures better machine efficiency. This means the programs run faster.	A program called an 'assembler' is required to convert the program into machine language. Thus, it takes longer to execute than a machine language program.	A program called a compiler or interpreter is required to convert the program into machine language. Thus, it takes more time for a computer to execute.
Time to develop	Needs a lot of skill, as instructions are very lengthy and complex. Thus, it takes more time to program.	Simpler to use than machine language, though instruction codes must be memorized. It takes less time to develop programs as compared	Easiest to use. Takes less time to develop programs and, hence, ensures better program efficiency.

		to machine language.	

Developing a computer program

Follow the steps given below to become a successful programmer:

1. *Define the problem*: Examine the problem until you understand it thoroughly.

2. *Outline the solution*: Analyze the problem.

3. Expand the outline of the solution into an *algorithm*: Write a step-by-step procedure that leads to the solution.

4. *Test the algorithm for correctness*: Provide test data and try to work out the problem as the computer would. This is a critical step but one that programmers often forget.

5. *Convert the algorithm into a program*: Translate the instructions in the algorithm into a computer program using any programming language.

6. *Document the program clearly*: Describe each line of instruction or at least some important portions in the program. This will make the program easy to follow when accessed later for corrections or changes.

7. *Run the program*: Instruct the computer to execute the program. The process of running the program differs from language to language.

8. *Debug the program*: Make sure that the program runs correctly without any errors or bugs as they are called in computer terminology. Finding the errors and fixing them is called *debugging*. Don't get depressed when bugs are found. Think of it as a way to learn.

C language

- C++ is derived from C Language. It is a Superset of C.
- Earlier C++ was known as C with classes.
- In C++, the major change was the addition of classes and a mechanism for inheriting class objects into other classes.
- Most C Programs can be compiled in C++ compiler.
- C++ expressions are the same as C expressions.
- All C operators are valid in C++.
- C was invented to write an operating system called UNIX.
- C is a successor of B language which was introduced around 1970
- The language was formalized in 1988 by the American National Standard Institute (ANSI).
- The UNIX OS was totally written in C by 1973.
- Today C is the most widely used and popular System Programming Language.
- Most of the state-of-the-art softwares have been implemented using C.
- Today's most popular Linux OS and RBDMS MySQL have been written in C.

CONTROL STRUCTURES

- The control flow statements of a language determine the order in which the statements are executed.
- We also need to be able to specify that a statement, or a group of statements, is to be carried out conditionally, only if some condition is true.
- Also we need to be able to carry out a statement or a group of statements repeatedly based on certain conditions.
-

These kinds of situations are described in C using C*onditional Control* and L*oop Control* structures.

Conditional & loop structures

A conditional structure can be implemented in C using

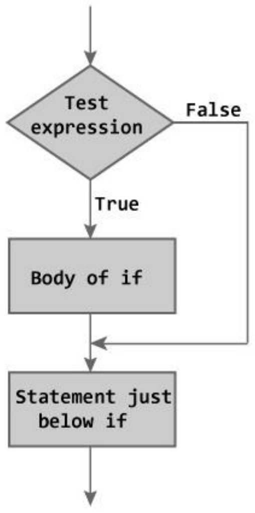
- The if statement
- The if-else statement

- The nested if-else statement
- The switch statement.
- break statement
- continue statement
- goto statement

Where as loop control structures can be implemented in C using

- while loop
- do-while loop
- for statement

The if statement



The if statement is used to control the flow of execution of statements. The general form of if statement is

```
if (test expression)
statement;
```

Suppose if it is required to include more than one statement, then a compound statement is used, in place of single statement. The form of compound statement is

```
if (condition)
{
statement1;
statement2;
}
```

If the condition is true, then the statement/statements will be executed.
If the condition is false, then the statement/statements will not be executed.

Program

/* Write a C program to print the number entered by user only if the number entered is negative. */

```
#include <stdio.h>
    int main()
{
    int num;
    printf("Enter a number to check.\n");
    scanf("%d",&num);
    if(num<0)
       {     /* checking whether number is less than 0 or not. */
        printf("Number = %d\n",num);
       }
/*If test condition is true, statement above will be executed, otherwise it will not be executed */
    printf("The if statement in C programming is easy.");
return 0;
}
```

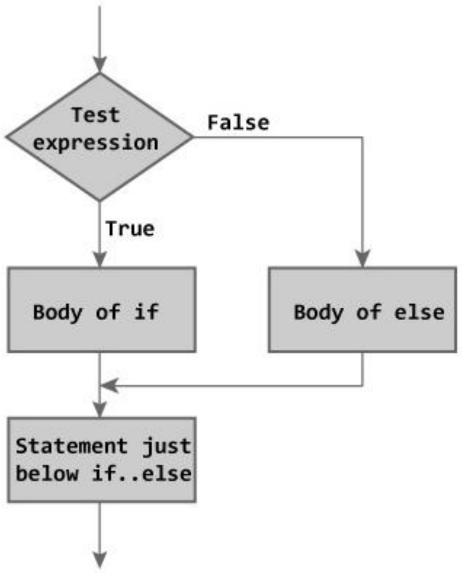
Output :

```
Enter a number to check.
-2
Number = -2
The if statement in C programming is easy.
```

○ if-else statement

The general form of if-else statement is…

```
if (test expression)
{
   statement1 to be executed if test expression is true;
}
else
 {
   statement2 to be executed if test expression is false;
}
```



If the condition is true, then statement1 is executed. Otherwise if the condition is false, then the statement2 is executed. Here statements statement1 and statement2 are either simple statements or compound statements.

Example:

/* Write a C program to check whether a number entered by user is even or odd */

```
#include <stdio.h>
int main(){
    int num;
    printf("Enter a number you want to check.\n");
    scanf("%d",&num);
    if((num%2)==0)        //checking whether remainder is 0 or not.
        {
       printf("%d is even.",num);
        }
    else
        {
       printf("%d is odd.",num);
        }
    return 0;
}
```

Output 1

```
Enter a number you want to check.
25
25 is odd.
```

Output 2

```
Enter a number you want to check.
2
2 is even.
```

Nested if-else Statements

When a series of conditions are involved, we can use more than one if-else statement in nested form.
This form is also known as *if-else if-else* statements. The general form of if-else if-else statement is

```
if (test expression1){
   statement/s to be executed if test expression1 is true;
```

```
    }
    else if(test expression2) {
        statement/s to be executed if test expression1 is false and 2 is true;
    }
    else if (test expression 3) {
       statement/s to be executed if text expression1 and 2 are false and 3 is true;
    }
       .
       .
       .
    else {
          statements to be executed if all test expressions are false;
      }
```

Note that a program contains number of else if statements and must be ended with else statement.

Write a C program to relate two integers entered by user using = or > or < sign.

```
#include <stdio.h>
int main()
{
    int numb1, numb2;
    printf("Enter two integers to check\n");
    scanf("%d %d",&numb1,&numb2);
    if(numb1==numb2) //checking whether two integers are equal.

           printf("Result: %d = %d",numb1,numb2);
    else if(numb1>numb2) //checking whether numb1 is greater than numb2.
          printf("Result: %d > %d",numb1,numb2);
            else
        printf("Result: %d > %d",numb2,numb1);
return 0;
}
```

Output 1

```
Enter two integers to check.
5
3
Result: 5 > 3
```

Switch case statement

The Switch statement is an extension of the if-else if-else statement. The switch makes one selection when there are several choices to be made. The direction of the branch taken by the switch statement is based on the value of any int (or int compatible) variable or expression.
The general form of Switch statement is shown below.

```
switch (expression)
{
        case constant1:statement 1;
        case constant 2:statement 2;
        case constant 3:statement 3;
        .
        .
        .
        .
        case value n:statement n;
        default :statement;
}
```

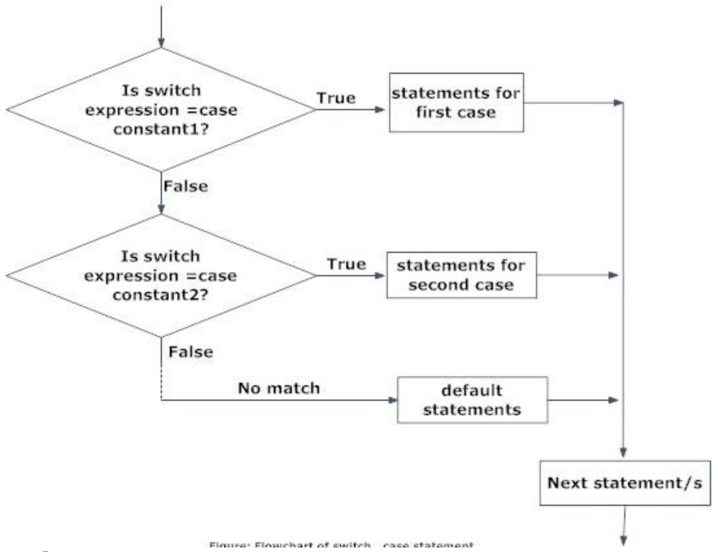

Is switch expression =case constant1?
True
statements for first case
False
Is switch expression =case constant2?
True
statements for second case
False
No match
default statements
Next statement/s

Figure: Flowchart of switch...case statement

Rules

- The switch expression must be an integral type.
- Case labels must be constants or constants expressions.
- Case labels must be unique. No two labels can have the same value.
- Case labels must end with colon.
- The break statement transfers the control out of the switch statement.
- The break statement is optional. i.e. two or more case labels may belong to the same statements
- The default label is optional. If present, it will be executed when the expression does not find a matching case label.
- There can be at most one default label.
- It is permitted to nest switch statements

Example:

/* C Program to create a simple calculator for addition, subtraction, multiplication and division

```
# include <stdio.h>
int main()
 {
   char o;
   float num1,num2;
   printf("Select an operator either + or - or * or / \n");
   scanf("%c",&o);
   printf("Enter two operands: ");
```

```
    scanf("%f%f",&num1,&num2);
    switch(o) {
        case '+':
            printf("%.1f + %.1f = %.1f",num1, num2, num1+num2);
            break;
        case '-':
            printf("%.1f - %.1f = %.1f",num1, num2, num1-num2);
            break;
        case '*':
            printf("%.1f * %.1f = %.1f",num1, num2, num1*num2);
            break;
        case '/':
            printf("%.1f / %.1f = %.1f",num1, num2, num1/num2);
            break;
        default:
            /* If operator is other than +, -, * or /, error message is shown */
            printf("Error! operator is not correct");
            break;
    }
    return 0;
}
```

Output

```
Enter operator either + or - or * or /
*
Enter two operands: 2.3
4.5
2.3 * 4.5 = 10.3
```

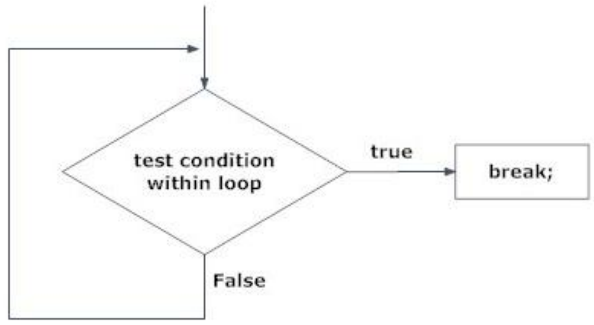
Break & continue statements

There are two statements built in C programming, break; and continue; to alter the normal flow of a program. Loops perform a set of repetitive task until text expression becomes false but it is sometimes desirable to skip some statement/s inside loop or terminate the loop immediately without checking the test expression. In such cases, break and continue statements are used. The break; statement is also used in switch statement to exit switch statement.

break Statement

In C programming, break is used in terminating the loop immediately after it is encountered. The break statement is used with conditional if statement.

Syntax of break statement
break;



Write a C program to find average of maximum of n positive numbers entered by user. But, if the input is negative, display the average(excluding the average of negative input) and end the program.

```
# include <stdio.h>
int main()
{
   float num,average,sum;
   int i,n;
   printf("Maximum no. of inputs\n");
   scanf("%d",&n);
   for(i=1;i<=n;++i){
       printf("Enter n%d: ",i);
       scanf("%f",&num);
       if(num<0.0)
       break;                    //for loop breaks if num<0.0
       sum=sum+num;
}
  average=sum/(i-1);
  printf("Average=%.2f",average);
  return 0;
}
```

Output

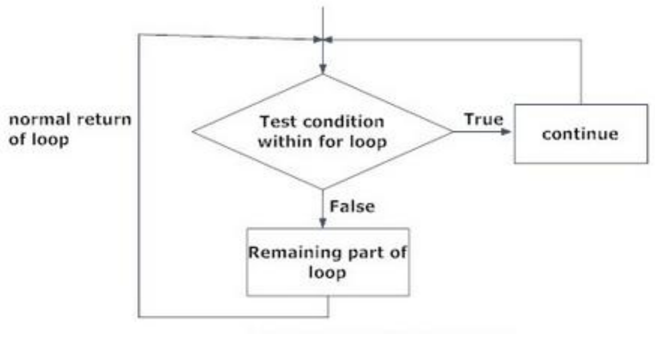
```
Maximum no. of inputs
4
Enter n1: 1.5
Enter n2: 12.5
Enter n3: 7.2
Enter n4: -1
Average=7.07
```

○ ***Continue statement***

It is sometimes desirable to skip some statements inside the loop. In such cases, continue statements are used.

Syntax of continue Statement

```
continue;
```



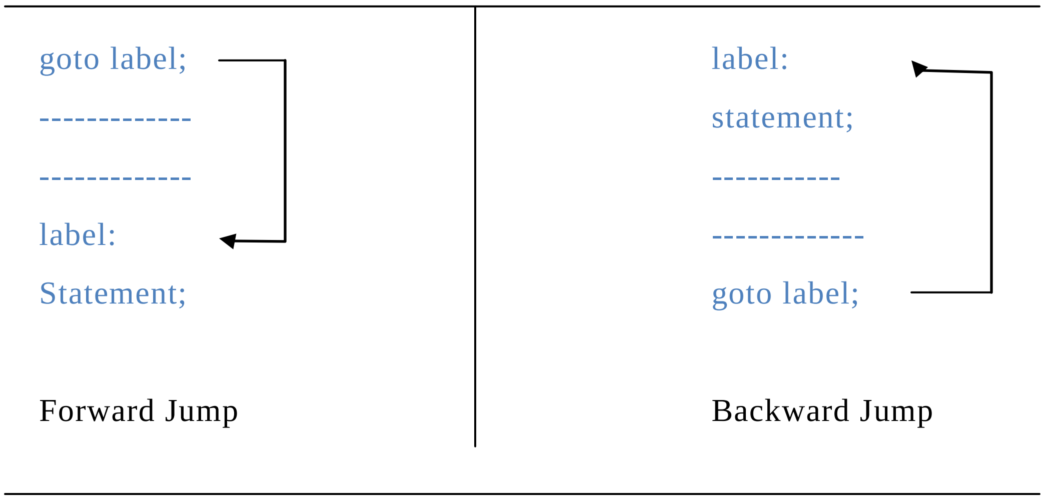
```
//program to demonstrate the working of continue statement in C programming
# include <stdio.h>
int main()
{
    int i,num,product;
 for(i=1,product=1;i<=4;++i){
        printf("Enter num%d:",i);
        scanf("%d",&num);
        if(num==0)
            continue;
/ *In this program, when num equals to zero, it skips the statement product*=num and
continue the loop. */
        product*=num;
}
    printf("product=%d",product);
return 0;
}
```

- **goto statement**

In C programming, goto statement is used for altering the normal sequence of program execution by transferring control to some other part of the program.

Syntax of goto statement

```
goto label;
.............
.............
.............
label:
statement;
```



- The goto label requires a label in order to identify the place where the branch is to be made.
- A label is any valid variable name , & must be end with colon.
- The label is placed immediately before the statement where the control is to be transferred.

Example of goto statement

```
/*  C program to demonstrate the working of goto statement. */
/* This program calculates the average of numbers entered by user. */
/* If user enters negative number, it ignores that number and
   calculates the average of number entered before it.*/

# include <stdio.h>
int main(){
   float num,average,sum;
   int i,n;
   printf("Maximum no. of inputs: ");
   scanf("%d",&n);
   for(i=1;i<=n;++i){
       printf("Enter n%d: ",i);
       scanf("%f",&num);
       if(num<0.0)
       goto jump;             /* control of the program moves to label jump */
       sum=sum+num;
}
jump:
  average=sum/(i-1);
  printf("Average: %.2f",average);
  return 0;
}
```

Output:

```
Maximum no. of inputs: 4
Enter n1: 1.5
Enter n2: 12.5
Enter n3: 7.2
Enter n4: -1
Average: 7.07
```

○ ***LOOPS***

A portion of program that is executed repeatedly is called a *loop*.

Loops causes program to execute the certain block of code repeatedly until some conditions are satisfied, i.e., loops are used in performing repetitive work in programming.

Suppose you want to execute some code/s 10 times. You can perform it by writing that code/s only one time and repeat the execution 10 times using loop.

The C programming language contains three different program statements for program looping. They are

- For loop
- While loop
- Do-While loop

○ **For loop**

The for loop is used to repeat the execution statement for some fixed number of times.

The general form of for loop is

```
for(initialization statement; test expression; update statement)
{
      code/s to be executed;
}
```

How for loop works in C programming?

The initialization statement is executed only once at the beginning of the for loop. Then the test expression is checked by the program. If the test expression is false, for loop is terminated. But if test expression is true then the code/s inside body of for loop is executed and then update expression is updated. This process repeats until test expression is false.

where the statement is single or compound statement.

- *Initialization expression:* is the initialization expression, usually an assignment to the loop-control variable. This is performed once before the loop actually begins execution.
- *Test-conditions* is the test expression, which evaluated before each iteration of the loop, which determines when the loop will exist.
- *Increment/upatde expression:* is the modifier expression, which changes the value of loop control variable. This expression is executed at the end of each loop.

1. **<u>multiple initialization & update expressions:</u>**

A for loop may contain multiple initialization and /or multiple update expressions. These multiple expressions must be separated by commas.
Example
For(i=1;sum=0;i<=n; sum+=I,++i)

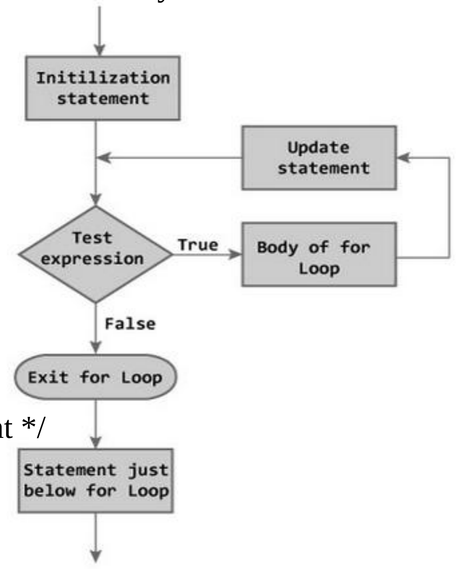
<u>2.Optional expressions</u>

In a for loop ,initialization expression, test expression& update expression are optional i.e you can skip any or all of these expressions.
Even if you Skip initialization –expression, but the ; following it must be present.

For(;test expression;update-experssion(s))
Loop body

3. Infinte loop- although any loop statement can be used to create an infinite(endless loop)
   For(; ;)
   Printf("endless");

/* Write a program to find the sum of first n natural numbers where n is entered by user. Note: 1,2,3... are called natural numbers.*/



```
#include <stdio.h>
int main()
{
   int n, count, sum=0;
   printf("Enter the value of n.\n");
   scanf("%d",&n);
   for(count=1;count<=n;++count)  //for loop terminates if count<n
   {
      sum+=count;    /* this statement is equivalent to sum=sum+count */
   }
   printf("Sum=%d",sum);
   return 0;
}
```

Output

```
Enter the value of n.
19
Sum=190
```

○ Nested for loop

The syntax for a **nested for loop** statement in C is as follows:

```
for ( init; condition; increment )
{
   for ( init; condition; increment )
   {
      statement(s);
   }
   statement(s);

}
```

program uses a nested for loop to find the prime numbers from 2 to 100:

```
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int i, j;

   for(i=2; i<100; i++) {
      for(j=2; j <= (i/j); j++)
        if(!(i%j)) break; // if factor found, not prime
      if(j > (i/j)) printf("%d is prime\n", i);
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
```

```
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```

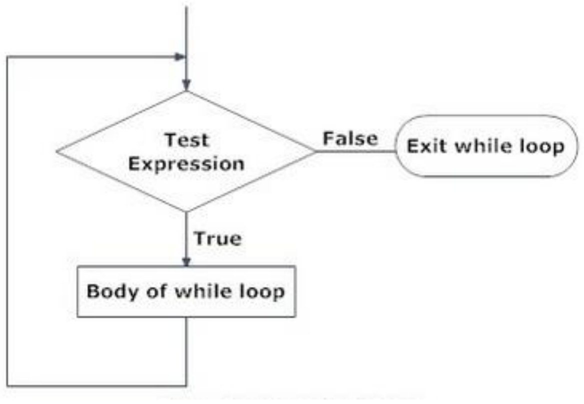
○ The While Loop

The while loop checks whether the test expression is true or not. If it is true, code/s inside the body of while loop is executed, that is, code/s inside the braces { } are executed. Then again the test expression is checked whether test expression is true or not. This process continues until the test expression becomes false

The general form of while loop is

```
while(conditional-expression)
{
statement;
increment/decrement;
}

Or
While(expression)
Loop body
```



where the statement (body of the loop) may be a single statement or a compound statements. The expression (test condition) must result zero or non-zero.

Example:

```
/*C program to demonstrate the working of while loop*/

#include <stdio.h>
    int main(){
    int number,factorial;
    printf("Enter a number.\n");
    scanf("%d",&number);
    factorial=1;
    while (number>0){      /* while loop continues util test condition number>0 is true */
          factorial=factorial*number;
          --number;
}
printf("Factorial=%d",factorial);
return 0;
}
```

Output

```
Enter a number.
5
Factorial=120
```

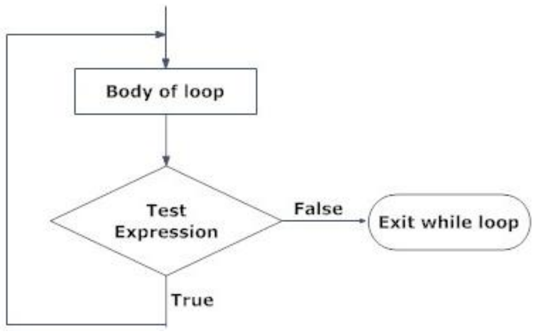
○ The do-while loops

In c,do while loop is very similar to while loop, only difference between these two loops is that, in while loops test expression is checked at first but, in do while code is executed at first then the condition is checked. So the code are excuted at least once in do. While loops

The structure of do-while loop is similar to while loop. The difference is that in case of do-while loop the expression is evaluated after the body of loop is executed. In case of while loop the expression is evaluated before executing body of loop.

The general form of do-while statement is

```
do
{
          Body of loop
}
while(expression);
```



At first codes inside body of do is executed. Then, the test expression is checked. If it is true, code/s inside body of do are executed again and the process continues until test expression becomes false(zero).

Notice, there is semicolon in the end of while (); in do...while loop.

Write a C program to add all the numbers entered by a user until user enters 0.

```
/*C program to demonstrate the working of do...while statement*/
#include <stdio.h>
int main()
```

```
{
  int sum=0,num;
  do           /* Codes inside the body of do...while loops are at least executed once. */
  {
      printf("Enter a number\n");
      scanf("%d",&num);
      sum+=num;
  }
  while(num!=0);
  printf("sum=%d",sum);
return 0;
}
```

Output

```
Enter a number 3
Enter a number -2
Enter a number 0
sum=1
```

Subprogram

In comp. science ,subprogram (also called subroutine,procedure,function or method) is a portion of code within the larger program that perform a specific task.

Uses of C functions:

- C functions are used to avoid rewriting same logic/code again and again in a program.
- There is no limit in calling C functions to make use of same functionality wherever required.
- We can call functions any number of times in a program and from any place in a program.
- A large C program can easily be tracked when it is divided into functions.
- The core concept of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality and to improve understandability of very large C programs.

C function declaration, function call and function definition:

There are 3 aspects in each C function. They are,

- Function declaration or prototype - This informs compiler about the function name, function parameters and return value’s data type.
- Function call – This calls the actual function
- Function definition – This contains all the statements to be executed.

S.no	**C function aspects**	**syntax**
1	function definition	return_type function_name (arguments list) { Body of function; }
2	function call	function_name (arguments list);
3	function declaration	return_type function_name (argument list);

Example:

```
#include<stdio.h>
// function prototype, also called function declaration
float square ( float x );
// main function, program starts from here
int main( )
{

        float m, n ;
        printf ( "\nEnter some number for finding square \n");
        scanf ( "%f", &m ) ;
        // function call
        n = square ( m ) ;
        printf ( "\nSquare of the given number %f is %f",m,n );

}

float square ( float x )   // function definition
{
        float p ;
        p = x * x ;
        return ( p ) ;
}
```

Output:

```
Enter some number for finding square
2
Square of the given number 2.000000 is 4.000000
```

In programming, a function is a segment that groups code to perform a specific task. A C program has at least one function main( ). Without main() function, there is technically no C program.

Types of C functions

There are two types of functions in C programming:

- Library function
- User defined function

Library function

Library functions are the in-built function in C programming system. For example:

main()

- The execution of every C program starts from this main() function.

  printf()-prinf() is used for displaying output in C.
  scanf()-scanf() is used for taking input in C.

User defined function-

C allows programmer to define their own function according to their requirement. These types of functions are known as user-defined functions. Suppose, a programmer wants to find factorial of a number and check whether it is prime or not in same program. Then, he/she can create two separate user-defined functions in that program: one for finding factorial and other for checking whether it is prime or not.

Syntax

```
return_type function_name( parameter list )

{

  body of the function

}
```

A function definition in C programming language consists of a *function header* and a *function body*. Here are all the parts of a function:

- **Return Type**: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers

to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

- **Function Body:** The function body contains a collection of statements that define what the function does.

Advantages of user defined functions

1. User defined functions helps to decompose the large program into small segments which makes programmer easy to understand, maintain and debug.
2. If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
3. Programmer working on large project can divide the workload by making different functions.

Scope:

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable cannot be accessed. There are three places where variables can be declared in C programming language:

1. Inside a function or a block which is called **local** variables,
2. Outside of all functions which is called **global** variables.
3. In the definition of function parameters which is called **formal** parameters.

Let us explain what are **local** and **global** variables and **formal** parameters.

Local Variables

- Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables. Here all the variables a, b and c are local to main() function.

```
#include <stdio.h>
 int main ()
{
  /* local variable declaration */
  int a, b;
  int c;
  /* actual initialization */
 a = 10;
  b = 20;
  c = a + b;
 printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
  return 0;
}
```

Global Variables

- Global variables are defined outside of a function, usually on top of the program. The global variables will hold their value throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.
- A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables:

```
#include <stdio.h>

/* global variable declaration */
int g;

int main ()
{
  /* local variable declaration */
  int a, b;

  /* actual initialization */
  a = 10;
  b = 20;
  g = a + b;

  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

  return 0;
}
```

- A program can have same name for local and global variables but value of local variable inside a function will take preference. Following is an example:

```
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main ()
{
  /* local variable declaration */
  int g = 10;

  printf ("value of g = %d\n",  g);

  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of g = 10
```

- **Formal Parameters**

Function parameters, formal parameters, are treated as local variables with-in that function and they will take preference over the global variables. Following is an example:

```
#include <stdio.h>

/* global variable declaration */
int a = 20;

int main ()
{
  /* local variable declaration in main function */
  int a = 10;
  int b = 20;
  int c = 0;

  printf ("value of a in main() = %d\n",  a);
  c = sum( a, b);//actual parameters
  printf ("value of c in main() = %d\n",  c);

  return 0;
}

/* function to add two integers */
int sum(int a, int b)//formal parameters
{
    printf ("value of a in sum() = %d\n",  a);
    printf ("value of b in sum() = %d\n",  b);

    return a + b;
}
```

Actual parameters-Arugments which used in function call.

Function Arguments:

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by reference	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Call by value

```
include <stdio.h>
 /* function declaration */
            void swap(int x, int y);
             int main ()
            {
            /* local variable definition */
            int a = 100;
            int b = 200;
            printf("Before swap, value of a : %d %d\n", a,b );
            /* calling a function to swap the values */
             swap(a, b);
            printf("after swap, value of a : %d %d\n", a,b );
}
  void swap(int a,int b)
{
            int tmp;
            a=b;
            b=tmp;
            printf("After swap, value of a : %d%d\n", a,b );
             return 0;
}
```

Let us put above code in a single C file, compile and execute it, it will produce the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

Call by reference

```
include <stdio.h>
 /* function declaration */
void swap(int &x, int &y);
 int main ()
{
             /* local variable definition */
             int a = 100;
            int b = 200;
            printf("Before swap, value of a : %d %d\n", a,b );
            /* calling a function to swap the values */
            swap(&a, &b);
            printf("after swap, value of a : %d %d\n", a,b );
             void swap(int &a,int &b)
            {
            int tmp;
            a=b;
            b=tmp;
            printf("After swap, value of a : %d%d\n", a,b );
            return 0;
            }
```

Recursion

A function that calls itself is known as recursive function and this technique is known as recursion in c programming.

Example of recursion in C programming

/* Factorial by Recursion */

```
#include<stdio.h>
int fact(int);
int main()
{
  int n;
  printf("Type any value : ");
  scanf("%d",&n);
  n=fact(n);
  printf("\nFactorial : %d ",n);
  return 0;
}

int fact(int x)
{
  if(x==1)
    return(x);
  else
   x=x*fact(x-1);
}
```

/* Fibonacci by Recursion */

```
#include<stdio.h>
int fib(int);

int main()
{
 printf("Type any value : ");
 printf("\nNth value: %d",fib(getche()-'0'));
 return 0;
}

int fib(int n)
{
 if(n<=1)
  return n;
 return(fib(n-1)+fib(n-2));
}
```

/* Reverse by Recursion */

```
#include<stdio.h>
int rev(int,int);

int main()
{
 int a;
 printf("Type a value : ");
 scanf("%d",&a);
 printf("Reverse: %d",rev(a,0));
 return 0;
}

int rev(int i,int r)
{
 if(i > 0)
  return rev(i/10,(r*10)+(i%10));
 return r;
}
```