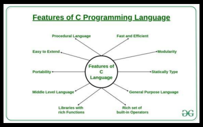
Q1. What are the features of the C language?
Ans. Features of C Programming Language
C is a procedural programming language. It was initially developed by Dennis Ritchie in the year 1972. It was mainly developed as a system programming language to write an operating system. The main features of C language include low-level access to memory, a simple set of keywords, and a clean style, these features make C language suitable for system programming's like an operating system or compiler development.
Features of C Programming Language:

1. Procedural Language
2. Fast and Efficient
3. Modularity
4. Statically Type
5. General-Purpose Language
6. Rich set of built-in Operators
7. Libraries with rich Functions
8. Middle-Level Language
9. Portability
10. Easy to Extend



1. **Procedural Language:** In a procedural language like C step by step predefined instructions are carried out. C program may contain more than one function to perform a particular task. New people to programming will think that this is the only way a particular programming language works. There are other programming paradigms as well in the programming world. Most of the commonly used paradigm is an object-oriented programming language.
2. **Fast and Efficient:** Newer languages like java, python offer more features than c programming language but due to additional processing in these languages, their performance rate gets down effectively. C programming language as the been middle-level language provides programmers access to direct manipulation with the computer hardware but higher-level languages do not allow this. That's one of the reasons C language is considered the first choice to start learning programming languages. It's fast because statically typed languages are faster than dynamically typed languages.
3. **Modularity:** The concept of storing C programming language code in the form of libraries for further future uses is known as modularity. This programming language van does very little on its own most of its power is held by its libraries. C language has its own library to solve common problems like in this we can use a particular function by using a header file stored in its library.
4. **Statically Type:** C programming language is a statically typed language. Meaning the type of variable is checked at the time of compilation but not at run time. This means each time a programmer type a program they have to mention the type of variables used.
5. **General Purpose Language:** From system programming to photo editing software, the C programming language is used in various applications. Some of the common applications where it's used are as follows:
   - Operating systems: Windows, Linux, iOS, Android, OXS
   - Databases: PostgreSQL, Oracle, MySQL, MS SQL Server etc.
6. **Rich set of built-in Operators:** It is a diversified language with a rich set of built-in operators which are used in writing complex or simplified C programs.
7. **Libraries with rich Functions:** Robust libraries and functions in C help even a beginner coder to code with ease.
8. **Middle-Level Language:** As it is a middle-level language so it has the combined form of both capabilities of assembly language and features of the high-level language.
9. **Portability:** C language is lavishly portable as programs that are written in C language can run and compile on any system with either none or small changes.
10. **Easy to Extend:** Programs written in C language can be extended means when a program is already written in it then some more features and operations can be added to it.

Q2. What do you mean by dynamic memory allocation in 'C'?

Ans. **The mechanism by which storage/memory/cells can be allocated to variables during the run time is called Dynamic Memory Allocation (not to be confused with DMA).** So, as we have been going through it all, we can tell that it allocates the memory during the run time which enables us to use as much storage as we want, without worrying about any wastage. **Dynamic memory allocation is the process of assigning the memory space during the execution time or the run time.**

Advantage of allocating memory dynamically:

1. When we do not know how much amount of memory would be needed for the program beforehand.
2. When we want data structures without any upper limit of memory space.
3. When you want to use your memory space more efficiently. *Example:* If you have allocated memory space for a 1D array as array[20] and you end up using only 10 memory spaces then the remaining 10 memory spaces would be wasted and this wasted memory cannot even be utilized by other program variables.
4. Dynamically created lists insertions and deletions can be done very easily just by the manipulation of addresses whereas in case of statically allocated memory insertions and deletions lead to more movements and wastage of memory.
5. When you want you to use the concept of structures and linked list in programming, dynamic memory allocation is a must.

Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

malloc() function in C

The malloc() function allocates single block of requested memory. It doesn't initialize memory at execution time, so it has garbage value initially. It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

- ptr=(cast-type*)malloc(byte-size)

```
#include<stdio.h>
#include<stdlib.h>
int main(){
 int n,i,*ptr,sum=0;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
   if(ptr==NULL)
```

```
    {
        printf("Sorry! unable to allocate memory");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
return 0;
}
```

Output

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

calloc() function in C

The calloc() function allocates multiple block of requested memory.
It initially initialize all bytes to zero.
It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

```
ptr=(cast-type*)calloc(number, byte-size)
```

```
#include<stdio.h>
#include<stdlib.h>
int main(){
 int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));  //memory allocated using calloc
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate memory");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
return 0;
}
```

Output

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

realloc() function in C

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the syntax of realloc() function.

```
ptr=realloc(ptr, new-size)
```

free() function in C

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

```
free(ptr)
```

C Functions

In c, we can divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the C program. In other words, we can say that the collection of functions creates a program. The function is also known as *procedure* or *subroutine* in other programming languages.

Advantage of functions in C

There are the following advantages of C functions.

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call C functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions.
- However, Function calling is always a overhead in a C program.

Function Aspects

There are three aspects of a C function.

- **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.

- **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.

- **Function definition** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

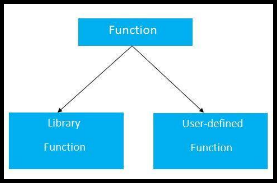
SN	**C function aspects**	**Syntax**
1	Function declaration	return type function name (argument list);
2	Function call	function_name (argument_list)
3	Function definition	return_type function_name (argument list) {function body;}

The syntax of creating function in c language is given below:

```
1. return_type function_name(data_type parameter...){
2. //code to be executed
3. }
```

Types of Functions

There are two types of functions in C programming:



1. **Library Functions**: are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.

2. **User-defined functions**: are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.

Return Value

A C function may or may not return a value from the function. If you don't have to return any value from the function, use void for the return type.

Example without return value:

```
void hello(){
printf("hello c");
}
```

If you want to return any value from the function, you need to use any data type such as int, long, char, etc. The return type depends on the value to be returned from the function.

Example with return value:

```
int get(){
return 10;
}
```

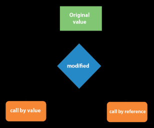
In the above example, we have to return 10 as a value, so the return type is int. If you want to return floating-point value (e.g., 10.2, 3.1, 54.5, etc), you need to use float as the return type of the method.

```
float get(){
return 10.2;
}
```

Now, you need to call the function, to get the value of the function.

Different aspects of function calling

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 aspects 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
- 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*.

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.

```c
#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
```

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

```c
#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 and 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.

```
#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 and b in main
    swap(&a,&b);
    printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual parameters 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 = 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 = 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

Q4. Write a program to print Fibonacci series up to 100.

Ans. Fibonacci Series in C: In case of fibonacci series, *next number is the sum of previous two numbers* for example 0, 1, 1, 2, 3, 5, 8, 13, 21 etc. The first two numbers of fibonacci series are 0 and 1.

There are two ways to write the fibonacci series program:

- Fibonacci Series without recursion
- Fibonacci Series using recursion

Fibonacci Series in C without recursion

Let's see the fibonacci series program in c without recursion.

```
#include<stdio.h>
int main()
{
 int n1=0,n2=1,n3,i,number;
 printf("Enter the number of elements:");
 scanf("%d",&number);
 printf("\n%d %d",n1,n2);//printing 0 and 1
 for(i=2;i<number;++i)//loop starts from 2 because 0 and 1 are already printed
 {
  n3=n1+n2;
  printf(" %d",n3);
  n1=n2;
  n2=n3;
 }
  return 0;
 }
```

Output:

OOPs Concepts in Java

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

Fibonacci Series using recursion in C

Let's see the fibonacci series program in c using recursion.

```
 #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
```

Section - B

Q1. List and explain at least six pre-processor directives along with examples.

Ans. The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP. All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column.

- A program which processes the source code before it passes through the compiler is known as **preprocessor.**
- The commands of the preprocessor are known as **preprocessor directives.**
- It is placed before the main().
- It begins with a # symbol.
- They are never terminated with a semicolon.

Sr.No.	Directive & Description
1	**#define** Substitutes a preprocessor macro.
2	**#include** Inserts a particular header from another file.
3	**#undef** Undefines a preprocessor macro.
4	**#ifdef** Returns true if this macro is defined.
5	**#ifndef** Returns true if this macro is not defined.
6	**#if** Tests if a compile time condition is true.

https://docs.microsoft.com/en-us/cpp/preprocessor/preprocessor-directives?view=msvc-160

Q2. What are the different data types used in C? Give example.
Ans. Each variable in C has an associated data type. Each data type requires different amounts of memory and has some specific operations which can be performed over it. Let us briefly describe them one by one:
Following are the examples of some very common data types used in C:

- **char:** The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.
- **int:** As the name suggests, an int variable is used to store an integer.
- **float:** It is used to store decimal numbers (numbers with floating point value) with single precision.
- **double:** It is used to store decimal numbers (numbers with floating point value) with double precision.

Different data types also have different ranges upto which they can store numbers. These ranges may vary from compiler to compiler. Below is list of ranges along with the memory requirement and format specifiers on 32 bit gcc compiler.

Data Type	Memory (bytes)	Range	Format Specifier
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu
long long int	8	-(2^63) to (2^63)-1	%lld
unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4	1.2E-38 to 3.4E+38	%f
double	8	2.3E-308 to 1.7E+308	%lf
long double	16	3.4E-4932 to 1.1E+4932	%Lf

We can use the sizeof() operator to check the size of a variable. See the following C program for the usage of the various data types:

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

```
{
    int a = 1;
    char b = 'G';
    double c = 3.14;
    printf("Hello World!\n");

    // printing the variables defined
    // above along with their sizes
    printf("Hello! I am a character. My value is %c and "
           "my size is %lu byte.\n",
           b, sizeof(char));
    // can use sizeof(b) above as well

    printf("Hello! I am an integer. My value is %d and "
           "my size is %lu  bytes.\n",
           a, sizeof(int));
    // can use sizeof(a) above as well

    printf("Hello! I am a double floating point variable."
           " My value is %lf and my size is %lu bytes.\n",
           c, sizeof(double));
    // can use sizeof(c) above as well

    printf("Bye! See you soon. :)\n");

    return 0;
}
```

Output:

```
Hello World!

Hello! I am a character. My value is G and my size is 1 byte.

Hello! I am an integer. My value is 1 and my size is 4  bytes.

Hello! I am a double floating point variable. My value is 3.140000 and my size i

s 8 bytes.

Bye! See you soon. :)
```

C Data Types are used to:

- Identify the type of a variable when it declared.
- Identify the type of the return value of a function.
- Identify the type of a parameter expected by a function.

ANSI C provides three types of data types:

1. **Primary(Built-in) Data Types**:
   void, int, char, double and float.
2. **Derived Data Types**:
   Array, References, and Pointers.
3. **User Defined Data Types**:
   Structure, Union, and Enumeration

Q3. What is an array? How it can be represented in memory? What are its various types? Explain in detail.

Ans. **Definition**

- Arrays are defined as the collection of similar type of data items stored at contiguous memory locations.
- Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc.
- Array is the simplest data structure where each data element can be randomly accessed by using its index number.
- For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variable for the marks in different subject. instead of that, we can define an array which can store the marks in each subject at a the contiguous memory locations.

The array **marks[10]** defines the marks of the student in 10 different subjects where each subject marks are located at a particular subscript in the array i.e. **marks[0]** denotes the marks in first subject, **marks[1]** denotes the marks in 2nd subject and so on.

Properties of the Array

1. Each element is of same data type and carries a same size i.e. int = 4 bytes.
2. Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
3. Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of data element.

Advantages of Array

- Array provides the single name for the group of variables of the same type therefore, it is easy to remember the name of all the elements of an array.
- Traversing an array is a very simple process, we just need to increment the base address of the array in order to visit each element one by one.
- Any element in the array can be directly accessed by using the index.

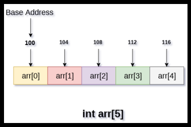
Memory Allocation of the array

- All the data elements of an array are stored at contiguous locations in the main memory. The name of the array represents the base address or the address of first element in the main memory. Each element of the array is represented by a proper indexing. The indexing of the array can be defined in three ways.
  1. 0 (zero - based indexing) : The first element of the array will be arr[0].
  2. 1 (one - based indexing) : The first element of the array will be arr[1].
  3. n (n - based indexing) : The first element of the array can reside at any random index number.

In the following image, we have shown the memory allocation of an array arr of size 5. The array follows 0-based indexing approach. The base address of the array is 100th byte. This will be the address of arr[0]. Here, the size of int is 4 bytes therefore each element will take 4 bytes in the memory.
In 0 based indexing, If the size of an array is n then the maximum index number, an element can have is **n-1**. However, it will be n if we use **1** based indexing.

Types of Arrays

- The various types of arrays are as follows.
    - One dimensional array
    - Multi-dimensional array



One-Dimensional Array

- A one-dimensional array is also called a single dimensional array where the elements will be accessed in sequential order. This type of array will be accessed by the subscript of either a column or row index.

Multi-Dimensional Array

- When the number of dimensions specified is more than one, then it is called as a multi-dimensional array. Multidimensional arrays include 2D arrays and 3D arrays.