Module 1 Notes and Related Questions

1. Which programming needs do structures fulfill? Why does C language enable us to create structures? What are the limitations of structures?

The Need for Structures

Variables sometimes influence each other's values. A change in the value of one may necessitate a corresponding adjustment in the value of another. It is, therefore, necessary to pass these variables together in a single group to functions. Structures enable us to do this. Take the example of date. A date can be represented in C by three different integer variables taken together.

```
int d,m,y; //Here d, m, and y represent the day of the month, the month, and the year, respectively.
```

Although these three variables are not grouped together in the code, they actually belong to the same group. *The value of one variable may influence the value of the other two*. In order to understand this clearly, consider a function next_day() that accepts the addresses of the three integers that represent a date and changes their values to represent the next day.

```
void next_day(int *,int *,int *); //function to calculate the next day
```

If d=31;m=12; y=1999; //31st December, 1999 and we call the function as next_day(&d,&m,&y); 'd' will become 1, 'm' will become 1, and 'y' will become 2000. A change in the value of one may change the value of the other two. *But there is no language construct that actually places them in the same group*. Thus, members of the wrong group may be accidentally sent to the function as given below:

```
d1=28; m1=2; y1=1999; //28th February, 1999
d2=19; m2=3; y2=1999; //19th March, 1999
next_day(&d1,&m1,&y1); //OK
next_day(&d1,&m2,&y2); //What? Incorrect set passed!
```

There is nothing in the language itself that prevents the wrong set of variables from being sent to the function. Moreover, integer-type variables that are not meant to represent dates might also be sent to the function. The solution to this problem is to create a data type called date itself using structures.

Why does C language enable us to create structures?

Structure is a programming construct in C that allows us to put together variables that should be together.

Library programmers use structures to create new data types. Application programs and other library programs use these new data types by declaring variables of this data type. A struct in C is a complex data type declaration that defines a physically grouped list of variables to be placed under one name in a block of memory, allowing the different variables to be accessed via a single variable or pointer. The struct can contain many other complex and simple data types in an association, so is a natural organizing type for records like the mixed data types. For example

```
struct date { //a structure to represent dates
       int d, m, y;
};
```

Now, the next_day() function will accept the address of a variable of the structure date as a parameter. Accordingly, its prototype will be as follows:

```
void next_day(struct date *);
struct date d1;
d1.d=28;
d1.m=2;
d1.y=1999;
next_day(&d1);
```

'd1.d', 'd1.m', and 'd1.y' will be correctly set to 1, 3, and 1999, respectively. Since the function takes the address of an entire structure variable as a parameter at a time, there is no chance of variables of the different groups being sent to the function.

Limitations of Structures

- In C structure can't contain functions means only data members are allowed
- Associated functions that were designed by library programmer don't have rights to work upon the data stored in structures.
- They are not a part of structure definition itself because application program might modify the structure variables by some code inadvertently written in application program itself While distributing his/her application, application programmer can't be sure that program would run successfully.
- Data stored in structures is not secure and can be manipulated by any function/procedure. Therefore every new piece of code accessing structure variable will have to be inspected and tested again to ensure that it doesn't corrupt the members of structure.

2. **Discuss the issues of procedure oriented programming systems with respect of data security. If object oriented programming solves it then how?**
3. **What is object oriented programming? Bring out the salient features of procedure oriented programming and object oriented programming**. **OR** State important features of object oriented programming. Compare object oriented system with procedure oriented systems.
4. **List out the difference between procedure oriented and object oriented techniques**. **OR** Compare/Differentiate between procedure oriented and object oriented programming systems.
5. **Explain the various characteristics /features of object oriented programming (OOP). OR** Describe the following characteristics/features of object oriented programming: i) Encapsulation ii) Polymorphism iii) Inheritance

Procedure oriented programming: In the **procedure oriented approach**, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. The focus of **procedure oriented programming** is to break down a programming task into a collection of functions, variables and data structures. The primary focus is on functions. A number of functions are written to accomplish these tasks. Conventional programming using high level languages such as FORTRAN and C, is commonly known as procedure oriented programming (POP).

Characteristics/Features of Procedure Oriented Programming System

- Large programs are divided into smaller programs known as functions
- Focus is on functions.
- Functions can share global data
- Data can move freely around the system from one function to other function. Data (contained in structure variables) is passed from one function to another to be read from or written into.
- Functions are associated with the data but they are not a part of it. Instead they receive structure variables / their addresses and work upon them.
- Employs top-down approach in program design.

Issues/Disadvantages of Procedure Oriented Programming System

- Data is not secure and can be manipulated by any function/procedure.
- Associated functions that were designed by library programmer don't have exclusive rights to work upon the data.
- They are not a part of structure definition itself because application program might modify the structure variables by some code inadvertently written in application program itself.
- While distributing the application, application programmer can't be sure that program would run successfully.
- Every new piece of code accessing structure variable will have to be inspected and tested again to ensure that it doesn't corrupt the members of structure.
- Compilers that implement procedure oriented programming systems don't prevent unauthorized functions from accessing / manipulating the structure variables. Lack of data security of procedure oriented programs has led to Object Oriented Programming Systems

How does object oriented programming solves the problem of data security?

In order to solve the problem of data security, one of the most important feature provided by the object oriented systems is "***data hiding***" facility. OOP gives lots of importance to data. Programmer can hide the really important core data from external world using OOP method. A class is a feature in OOP which facilitates to pack together different data types along with different functions that manipulate these data members of class. Data members can be declared as **private** or **public** inside a class. To hide the data from external world, a programmer does it by ***declaring the data as private***. ***Class allows to pack together data with its associated functions***. In OOP we use another entity named **'object'** to access both data and functions inside a class. We call data or function inside a class as '**class member**'. ***A class member can be accessed from external world (outside the class) using an object of the class only!***. This feature of data hiding is called as "**Data Encapsulation**". Thus one of the major flaws of POP is solved in OOP. OOP ties the data closely to a particular class and its objects. There is no need of "global data types" as in POP and hence data will not flow freely around the program. ***This makes sure there will be no 'accidental modification' of critical data.***

Object oriented programming: OOP allows decomposition of a problem into a number entities called objects and then builds data and functions around these objects. The data of an object can be accessed only by the functions associated with that object. However, functions of one object can access the functions of other objects. Object oriented programming breaks down a programming task into objects that expose behavior (methods) and data (members or attributes) using interfaces.

Characteristics of Object Oriented Programming System

- Emphasis is on data rather than procedures or functions.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up approach in program design.

Important features/concepts of OOPS

- **Object:** Object is a collection of attributes/entities. Objects take up space in the memory. Objects are instances of classes. When a program is executed, the objects interact by sending messages to one another. Each object contain data and code to manipulate the data. Objects can interact without having to know details of each other's data or code.
- **Class:** Class is a collection of objects of similar type. Objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. Eg: grapes bananas and orange are the member of class fruit. Example:
  Fruit orange; // Here object orange is an instance of class fruit.
- **Encapsulation:** Combining data and functions into a single unit called **class** and the process is known as **Encapsulation**. Data encapsulation is important feature of a class. Class contains both data and functions. Data is not accessible from the outside world and only those function which are present in the class can access the data.
- **Data Hiding and Abstraction:** The insulation of the data from direct access by the program is called **data hiding**. Hiding the complexity of program is called **Abstraction** and only essential features are represented. In short we can say that internal working is hidden.
- **Inheritance:** it is the process by which object of one class acquire the properties or features of objects of another class. The concept of inheritance provide the idea of reusability means we can add additional features to an existing class without modifying it. This is possible by driving a new class from the existing one. The new class will have the combined features of both the classes
  Example: Sparrow is a part of the class flying bird which is again a part of the class bird.
- **Polymorphism:** A Greek term means ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation. Examples
  - Function overloading & Operator Overloading
  - Static & Dynamic Polymorphism
- **Message Passing:** The process by which one object can interact with other object is called message passing.
- **Dynamic Binding:** Refers to linking of function call with function definition is called binding and when it is take place at run time called dynamic binding.

Difference between Procedure Oriented Programming (POP) & Object Oriented Programming (OOP)

	Procedure Oriented Programming (POP)	**Object Oriented Programming (OOP)**
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP, Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are: C, VB, FORTRAN, Pascal.	Example of OOP are: C++, JAVA, VB.NET, C#.NET.

6. Compare C++ with C?

C	C++
C language compiler cannot execute C++ programs	C++ language compiler can execute C programs
In C, you may or may not include function prototypes	In C++, you must Include function prototypes
C doesn't allow for default arguments	C++ lets you to specify default arguments in function prototype
Declaration of the variables must be at the beginning of the scope	Declaration of the variables can be anywhere before using
If a C program uses a Local variable that has Same name as global variable, then C uses the value of a local variable.	In C++, you can instruct program to use value of global variable with scope resolution Eg- cout << "I am global var :" << ::I;
Function overloading is not there	Function overloading exists
Function inside the structure is not allowed	Function inside the structure is allowed
Object initialization doesn't exist	Object initialization (constructor) exist
Data hiding, data abstraction and data encapsulation feature doesn't exist	Data hiding, data abstraction and data encapsulation exists in C++

7. What is cin and cout? Explain with examples.

Console Output (cout): Let us consider an example for outputting to console.

```
#include<iostream.h>
void main() {
       int x;
       x=10;
       cout<<x; //outputting to the console
}
```

Output

```
10
```

The third statement in the main() function, cout stands for the **console output** device, that is, the monitor. The **<< symbol** in the given context operates as the **insertion operator**. It takes two operands. The operand on its left must be some object of the ostream class. The operand on its right must be a value of some fundamental data type. The value on the right side of the insertion operator is 'inserted' into the console output device on the left. Consequently, the value of 'x' is displayed on the monitor. Another object endl allows us to insert a new line into the output stream. Example.

```
       cout<<endl; //inserting a new line by endl
```

Insertion operator is that it does not need the format specifiers that are needed in the printf(). It is also possible to cascade the insertion operator. Example: `cout<<x<<endl<<y; //cascading the insertion operator`

Console Input (cin): Let us consider an example for console input in C++.

```
#include<iostream.h>
void main() {
       int x;
       cout<<"Enter a number: ";
       cin>>x; //console input in C++
       cout<<"You entered: "<<x;
}
```

Output

```
Enter a number: 10<enter>
You entered: 10
```

The third statement in the main(), **cin** stands as alias for the **console input** device, that is, the keyboard. The **>> symbol**, in the given context, operates as the **extraction operator**. It is a binary operator and takes two operands. Left operand must be some object of the istream_withassign class. Right operand must be a variable of some fundamental data type. The value for the variable on right side of the extraction operator is extracted from the input stream device on the left. Consequently, the value of 'x' is obtained from the keyboard.

8. Why C++ introduced reference variable? Explain with example.

9. What are the features of reference variable?

Reference Variable: A reference variable is nothing but a reference for an existing variable. It is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, it shares the memory location with that variable. Either the variable name or the reference name may be used to refer to the variable.

Following are some of the features/characteristics of reference variables:

- You cannot have NULL references. A reference is always connected to a legitimate piece of storage.
- A reference must be initialized at the time of declaration.
- Once a reference is initialized to an object, it cannot be changed to refer to another object.
- All operations supposedly performed on the reference (alias) are actually performed on the original variable.

Some of the reasons why C++ introduced reference variables:

- C++ references allow you to create a second name (which can be shorter or a more suitable name) for the variable that you can use to read or modify the original data stored in that variable.
- If we use reference for function arguments, object copy can be avoided which reduces the memory space requirement.
- If function arguments are references, there is a way to modify the original data objects passed into the function.
- It is simpler to use references over pointers. It makes the program code more readable.

An example of use of reference variable is given below:

```
#include<iostream.h>
void main() {
       int x;
       x=10;
       cout<<x<<endl;
       int & iRef=x; //iRef is a reference to x
       iRef=20; //same as x=20;
       cout<<x<<endl;
       x++; //same as iRef++;
       cout<<iRef<<endl;
}
```

Output

```
10
20
21
```

10. Discuss the following with an example i) Call by reference ii) Return by reference

i) Call by Reference / Passing reference as function argument: A reference variable can be a function argument and thus changes the value of the parameter that is passed to it in the function call. An example is given below.

```
#include<iostream.h>
void increment(int &); //formal argument is a reference to the passed parameter
void main(){
       int x = 10;
       increment(x);
       cout<<x<<endl;
}
void increment(int & r) {
       r++; //same as x++;
}
```

Output

```
11
```

ii) Return by Reference: Functions can return by reference also. Consider the following example:

```
#include<iostream.h>
int & larger(int &, int &);
int main() {
       int x = 10,y = 20;
       int & r=larger(x,y);
       r=-1;
       cout<<x<<endl<<y<<endl;
}
int & larger(int & a, int & b) {
       if(a>b) //return a reference to the larger parameter
              return a;
       else
              return b;
}
```

Output

```
10
-1
```

In the foregoing listing, ‘a’ and ‘x’ refer to the same memory location while ‘b’ and ‘y’ refer to the same memory location. From the larger() function, a reference to ‘b’, that is, reference to ‘y’ is returned and stored in a reference variable ‘r’. The larger() function does not return the value ‘b’ because the return type is int& and not int. Thus, the address of ‘r’ becomes equal to the address of ‘y’. Consequently, any change in the value of ‘r’ also changes the value of ‘y’.

11. Discuss function prototyping with an example. Also write its advantages.

Function Prototyping: Function prototyping is a function declaration statement that is necessary in C++. A prototype describes the function’s interface to the compiler. It tells the compiler the return type of the function as well as the number, type, and sequence of its formal arguments.

Advantages of function prototype:

- A function must be defined before calling it. But prototyping allows a function to be called before defining it.
- With prototyping, compiler ensures following: The return value of a function is handled correctly and correct number and type of arguments are passed to a function. Function prototyping guarantees protection from errors arising out of incorrect function calls. It helps the compiler in determining whether a function is called correctly or not. Each time when a function is called, its calling statement is compared with its prototype. In case of any mismatch, compiler reports an error.
- A function prototyping produces automatic-type conversion wherever appropriate. For example, if the compiler expects an int type value, but a type of double value is wrongly passed, C++ compiler converts double type to int type automatically , this this because of the function prototype before the function call.

Example: Consider the following prototype statement: `int add(int, int);`
The prototype tells the compiler that the add() function returns an integer-type value. Thus, the compiler knows how many bytes have to be retrieved from the place where the add() function is expected to write its return value and how these bytes are to be interpreted. In the absence of prototypes, the compiler will have to assume the type of the returned value. Suppose, it assumes that the type of the returned value is an integer. However, the called function may return a value of an incompatible type (say a structure type). Now, suppose an integer-type variable is equated to the call to a function where the function call precedes the function definition. In this situation, the compiler will report an error against the function definition and not the function call. This is because the function call abided by its assumption, but the definition did not. However, if the function definition is in a different file to be compiled separately, then no compile-time errors will arise. Instead, wrong results will arise during run time as shows below:

```
// Absence of function prototype produces weird results:
/*Beginning of def.c*/
struct abc {
       char a;
       int b;
       float c;
};
struct abc test() { //*function definition*/
       struct abc a1;
       a1.a='x';
       a1.b=10;
       a1.c=1.1;
       return a1;
}
/*End of def.c*/
/*Beginning of driver.c*/
void main() {
       int x;
       x=test(); //no compile time error!!
       printf("%d",x);
}
/*End of driver.c*/
```

Output

```
1688
```

A compiler that does not enforce prototyping will definitely compile the above program. But then it will have no way of knowing what type of value the test() function returns. Therefore, erroneous results will be obtained during run time as the output clearly shows.

12. **What is function overloading? Write a program in C++ to overload the function add(S1, S2) where S1 and S2 are integers and floating point values**. **OR** What is function overloading? Illustrate function overloading through add function which as adds two integers and two float numbers. **OR** What is function overloading explain with an example.

Function Overloading: When same function name is used to create the functions that performs variety of different tasks, it is called **function overloading** (static polymorphism). C++ allows two or more functions to have the same name. In order to distinguish amongst the functions with the same name, the compiler expects their signatures to be different. Signature of a function means the number, type, and sequence of formal arguments of the function. Depending upon the type of parameters that are passed to the function call, the compiler decides which of the available definitions will be invoked. For this, function prototypes should be provided to the compiler for matching the function calls. Accordingly, the linker, during link time, links the function call with the correct function definition. Consider the following example:

Function overloading Example

```
#include<iostream.h>
int add(int,int); //first prototype
int add(int,int,int); //second prototype
void main() {
       int x,z;
       float y;
       x=add(10,20); //matches first prototype
       y=add(30,40,50); //matches second prototype
       cout<<x<<endl<<y<<endl;
}
int add(int a,int b) {
       return(a+b);
}
int add(int a,int b,int c) {
       return(a+b+c);
}
```

Output

```
30
120
```

13. **Write a C++ program to create a class complex and implement the following overloaded function ADD that returns a complex number: i) ADD(c1, c2) where both c1 and c2 are complex no. ii) ADD(x, c1) where x is real and c1 is complex no.**

```
#include<iostream.h>
class complex {
  int r, i;
  public:
  void read();
  void print();
  friend complex add(int a,complex c);
  friend complex add(complex c1,complex c2);
};
void complex::read() {
  cout<<"Enter real and imaginary\n";
  cin>>r>>i;
}
void complex::print() {
  cout<<r<<"+i"<<i<<endl;
}
complex add(int a,complex c) {
  complex t;
  t.r=a+c.r;
  t.i=c.i;
  return t;
}
complex add(complex c1,complex c2) {
  complex t;
  t.r=c1.r+c2.r;
  t.i=c1.i+c2.i;
  return t;
}
```

```
void main() {
   int a=2;
   complex s1,s2,s3;
   s1.read();
   cout<<endl<<"s1 : ";
   s1.print();
   s2=add(a,s1);
   cout<<"s2 : 2+s1"<<endl;
   cout<<"   : "; s2.print();
   s3=add(s1,s2);
   cout<<"s3=s1+s2"<<endl;
   cout<<"s1 : "; s1.print();
   cout<<"s2 : "; s2.print();
   cout<<"s3 : "; s3.print();
}
```

14. **Why should default values be given to function argument in function prototype? Explain with example**. OR **Discuss about "Default argument" with an example**
15. **Why should default values to be given to function arguments in function prototype and not in function definition? Write a program to add three numbers using function which has one or more default values**.

Default Values for Formal Arguments of Functions: It is possible to specify default values for some or all of the formal arguments of a function. If no value is passed for an argument when the function is called, the default value specified for it is passed. If parameters are passed in the normal fashion for such an argument, the default value is ignored. Consider the following example.

```
// Default values for function arguments
#include<iostream>
int add(int,int,int c=0); //third argument has default value
void main() {
       int x,y;
       x=add(10,20,30); //default value ignored
       y=add(40,50); //default value taken for the third parameter
       cout<<x<<endl<<y<<endl;
}
int add(int a,int b,int c) {
       return (a+b+c);
}
```

Output

```
60
```

Why should default values to be given to function arguments in function prototype and not in function definition?

Default values must be specified in function prototypes alone. They should not be specified in the function definitions. Since function prototypes are placed in header files, which are included in both the library files that contain the function's definition and the client program files that contain calls to the functions. While compiling the library file that contains the function definition, the compiler will obviously read the function prototype before it reads the function definition. Suppose the function definition also contains default values for the arguments. Even if the same default values are supplied for the same arguments, the compiler will think that you are trying to supply two different default values for the same argument. This is obviously unacceptable because the default value can be only one in number. Thus, default values must be specified in the function prototypes and not again in the function definitions.

17. **What is inline function? Explain its general syntax, merits and demerits**. OR
    What is an inline function? What are advantages of having function inline? OR
    Define inline function. Explain with an example program. What are the conditions when inline functions are not expanded? OR, **Explain the working of inline function with an example.**

What is inline function?

An inline function is a function whose compiled code is 'in line' with the rest of the program. That is, the compiler replaces the function call with the corresponding function code. For specifying an inline function, you must:

- prefix the definition of the function with the *inline* keyword and
- Define the function before all functions that call it, i.e. define it in the header file itself.

Example of inline function: Inline function to find the cube of a number
Write a C++ program to find cube of a number using inline function. OR

```
// program to find cube of a number using inline function.

#include<iostream.h>
inline double cube(double x) { return x*x*x; }
void main() {
       double a,b;
       a=cube(5.0);
       b=cube(4.5+7.5);
       cout<<a<<endl;
       cout<<b<<endl;
}
Output
125
1728
```

What are merits/advantages of inline function?

- With inline code, the program does not have to jump to another location to execute the code and then jump back. Inline functions, thus, run a little faster than regular functions.
    - It does not require function calling overhead.
    - It also save overhead of variables push/pop on the stack, while function calling.
    - It also save overhead of return call from a function.
- It increases locality of reference by utilizing instruction cache.

What are demerits/disadvantages of inline function?

- If an inline function is called repeatedly, then multiple copies of the function definition appear in the code Thus, the executable program itself becomes so large that it occupies a lot of space in the computer's memory during run time.
- Consequently, the program runs slow instead of running fast. Thus, inline functions must be chosen with care.

What are the conditions when inline functions are not expanded?
Under following circumstances, the compiler may not expand the function inline. Instead, it will issue a warning that the function could not be expanded inline and then compile all calls to such functions in the ordinary fashion. Those conditions are:

- The function is recursive.
- There are looping constructs in the function.
- There are static variables in the function.

18. **What are two ways of defining inline functions? Give examples**. OR
    Explain two different ways of defining member functions with example.

Member functions are made inline by either of the following two methods:

1. By defining the function within the class itself as given below

```
class A {
        public:
        void show() {/* definition of show inside the class */} //
};
```

2. By only prototyping and not defining the function within the class. The function is defined outside the class by using the scope resolution operator. The definition is prefixed by the inline keyword. The definition of the inline function must appear before it is called. Hence, the function should be defined in the same header file in which its class is defined. Consider the following example:

```
class A {
        public:
        void show();
};
inline void A::show() { //definition in header file itself
        //definition of A::show() function
}
```

Introduction to Classes and Objects

16. **What is a class? Explain the structure of a class with the help of an example. Differentiate between a class and a structure**. OR **What is a class? How is it created?**

What is a class? Explain the structure of a class with the help of an example.

A class is a mechanism for creating user-defined data types. It is similar to the C language structure data type. A class not only holds data variables, but can also hold functions. These variables and functions are members of a class. The variables are called data members and functions are called member functions. A class can be declared using the keyword class. Each class type represents a unique set of class members including data members, member functions, and other type names. Accessibility of the members of the class is specified using the access specifier keywords: private, public or protected. With these specifiers, the access right of the members that will follow are set. The members of a class private by default.

- private members of a class are only accessible by other private members of the same class.
- protected members are accessible by members of the same class or derived classes.
- public members are accessible from anywhere where the object is used.

The general form/strucure of a class looks like this:

```
class class_name {
       access_specifier:
              member1;
       access_specifier:
              member2;
};
```

An example showing structure of a class is given below:

```
class Rectangle {
       int width;// private by default
       int length; // private by default
public:
       void setwidth(int w) {width=w;}
       void setlength(int l) {length=l;}
       int area() { return length*width; }
};
```

Differentiate between a class and a structure

Structure	**Class (class)**
Declared using keyword struct	Declared using keyword class
Member of a structure are by default public	Member of a class are by default private
C structures contain only data members	Class contains data member and member function both
C structures cannot be inherited	Classes can be inherited
No data hiding features comes with C structures	Classes support data hiding using private and protected keywords
C structures do not support encapsulation	Classes support Encapsulation
C structure can't be abstract	Class can be abstract

17. **Compare the struct and class keywords of C++.**
    First two points in the above table.
18. **Write a C++ codeto create a class employee with data members name, age and salary. Display at least 5 Employee info.**

```
#include<iostream>
using namespace std;
class Employee {
        char  name[20];
        float age;
        float salary;
public:
        void getEmpDetails();
        void showEmpDetails();
};
void Employee :: getEmpDetails() {
        cout<<endl<<"Enter employee name:"<<endl; cin>>name;
        cout<<endl<<"Enter employee age:"<<endl; cin>>age;
        cout<<endl<<"Enter employee salary:"<<endl; cin>>salary;
}
```

```
void Employee :: showEmpDetails() {
        cout<<endl<<endl<<endl<<"Details of   :  "<<name;
        cout<<endl<<"Age:      "<<age;
        cout<<endl<<"Salary:      "<<salary; cout<<endl;
}
int main() {
        Employee emp[10];
        int i,num;
        system("cls");
        cout<<endl<<"Enter number of employees"<<endl;
        cin>>num;
        for(i=0;i<num;i++)
                emp[i].getEmpDetails();
        for(i=0;i<num;i++)
                emp[i].showEmpDetails();
        system("pause");
        return 0;
}
```

19. Write a C++ program to define a class called TIME with hour, minute and second as the data members and read(), display() and add() as the member functions.

```
#include <iostream>
using namespace std;
class time {
private:
    int hours, minutes, seconds;
public:
    time(){
           hours=0; minutes=0; seconds=0;
    }
    time(int a,int b,int c) {
           while(c>=60) {
                  c-=60;
                  minutes+=1;
           }
           minutes=minutes+1;
           seconds=c;
           while(b>=60) {
                  b-=60;
                  hours+=1;
           }
           hours=hours+1;
           minutes=b;
           hours=a;
    }
    void add(time t1,time t2) {
           seconds=t1.seconds+t2.seconds;
           while(seconds>=60) {
                  seconds-=60;
                  minutes+=1;
           }
           minutes=t1.minutes+t2.minutes;
           minutes=minutes+1;
           while(minutes>=60) {
                  minutes-=60;
                  hours+=1;
           }
             hours=t1.hours+t2.hours;
             hours=hours+1;
       }
       void showtime() {
             cout<<"time 1 is 12:28:32 and time 2 is 7:44:55 ";
             cout<<"\nSum of times is "<<hours<<":"<< minutes<< ":"<< seconds<< ".\n";
        }
};
```

```
int main() {
    time t1(12,28,32);
    time t2(7,44,55);
    time t3;
    t3.add(t1,t2);
    t3.showtime();
}
```

19. Explain class objects. With the help of example explain how data hiding and encapsulation characteristics are achieved in C++.

Objects / class Objects: Variables of the class are known as objects. Class is mere a blueprint or a template. No storage is assigned when we define a class. Objects are instances of class, which holds the data variables declared in class and the member functions work on these class objects.

- Objects take up space in the memory. Each object has different data variables.
- Objects are initialized using special class functions called Constructors.
- When a program is executed, the objects interact by sending messages to one another. Objects can interact without having know details of each others data or code.

Following statements declare two objects of class Box:

```
Box Box1;          // Declare Box1 of class Box
Box Box2;          // Declare Box2 of class Box
```

For data hiding and encapsulation, refer next question.

20. What is data abstraction? How is it implemented in C++? Explain with an example. OR
What is data hiding? How is it achieved in C++? OR
With the help of example explain how data hiding and encapsulation characteristics are achieved in C++.

The insulation of the data from direct access by the program is called **data hiding**. Data hiding is a technique specifically used in object-oriented programming (OOP) to hide internal object details (data members). Data hiding ensures exclusive data access to class members and protects object integrity by preventing unintended or intended changes. Data hiding also reduces system complexity for increased robustness by limiting interdependencies between software components. Hiding the complexity of program is called Abstraction and only essential features are represented. In short we can say that internal working is hidden.

In C++, data hiding is achieved through **Encapsulation**. Encapsulation is the process of combining data and function into a single unit called **Class**. Data Hiding is the mechanism where the details of the class are hidden from the user. The user can perform only a restricted set of operations in the hidden member of the class. All the essential properties of the object that are to be created are encapsulated within the class.

Within a class members can be declared as either **public**, **protected** or **private** in order to explicitly enforce encapsulation. The private members are accessible only to the methods of the class. The protected members are accessible only to the methods of the class and derived classes. The public members (data/function) are accessible to all the user of the class. The data is hidden inside the class by declaring it as private inside the class. Thus private data cannot be directly accessed by the object.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes. We already have studied that a class can contain private, protected and public members. By default, all items defined in a class are private. For example:

```
class Box {
   public:
      double getVolume(void) {
         return length * breadth * height;
      }
   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

The variables length, breadth, and height are private. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

21. Write a general form of C++ program

A general form of C++ program is given below:

```
// Introductory comments
// file name, programmer, when written or modified
// what program does
#include <iostream.h>
void main() {
       constant declarations
       variable declarations
       executable statements
}
```

22. Explain i) this operator ii) arrow operator OR
Define this pointer. Indicate the steps involved in referring to members of the invoking object.

this Pointer / this operator:

Every object in C++ has access to its own address through an important pointer called this pointer. The pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object. The C++ compiler creates and calls member functions of class objects by using a unique pointer called *this* pointer. It is always a constant pointer. The *this* pointer always points at the object with respect to which the function was called.

```
// An example of use of this pointer
int Box::compare(Box box) {
         return this->Volume() >  box.Volume();
}
```

Steps involved in referring to members of the invoking object using this pointer:

Once compiler ascertains that no attempt has been made to access the private members of an object by non-member functions, it converts the C++ code into an ordinary C language code as follows:

1. It converts the class into a structure with only data members as follows:

```
// Before
class Distance {
       int iFeet;
       float fInches;
       public:
       void setFeet(int); //prototype only
       int getFeet(); //prototype only
       void setInches(float); //prototype only
       float getInches(); //prototype only
};
// After
struct Distance {
       int iFeet;
       float fInches;
};
```

2. It puts a declaration of this pointer as a leading formal argument in the prototypes of all member functions as follows.

```
// Before
void setFeet(int);
// After
void setFeet(Distance * const, int);
```

3. It puts the definition of the this pointer as a leading formal argument in the definitions of all member functions as follows. It also modifies all the statements to access object members by accessing them through this pointer using the pointer-to-member access operator (->).

```
// Before
void Distance::setFeet(int x) {
       iFeet=x;
}
// After
void setFeet(Distance * const this, int x) {
       this->iFeet=x;
}
```

4. It passes the address of invoking object as a leading parameter to each call to the member functions as follows:

```
// Before
d1.setFeet(1);
// After
setFeet(&d1,1);
```

Arrow Operator

Member functions can be called with respect to an object through a pointer pointing at the object. The arrow operator (->) does this. The definition of the arrow (->) operator has been extended in C++. It takes not only data members on its right as in C, but also member functions as its right-hand side operand. Consider the following example:

```
#include <iostream.h>
#include <Distance.h>
void main () {
   Distance d, *dptr;
   dptr= &d;
   dptr->setFeet(2);
   dptr->setInches(10);
}
```

If the operand on its right is a data member, then the arrow operator behaves just as it does in C language. However, if it is a member function of a class, then the compiler simply passes the value of the pointer as an implicit leading parameter to the function call. For example: dptr->setFeet(2) after conversion become: setFeet(dptr,2). Now , the 'dptr' value is copied to 'this' pointer

23. Demonstrate C++ program for i) passing objects to functions ii) Returning objects

Following example illustrates a class Class representing a complex number. The method `addComplexNumbers` takes a Complex object comp2 as a parameter and adds the real and imaginary parts of comp2 and this complex object stores the result into a temporary complex object temp which is finally returned by the function.

```
#include <iostream>
using namespace std;
class Complex {
    private:
       int real;
       int imag;
    public:
       Complex(): real(0), imag(0) { }
       void readData() {
           cout << "Enter real and imaginary number respectively:"<<endl;
           cin >> real >> imag;
        }
        Complex addComplexNumbers(Complex comp2) {
            Complex temp;
            temp.real = real+comp2.real;
            temp.imag = imag+comp2.imag;
            return temp;
        }
        void displayData() {
            cout << "Sum = " << real << "+" << imag << "i";
        }
};
int main(){
    Complex c1, c2, c3;
    c1.readData();
    c2.readData();
    c3 = c1.addComplexNumbers(c2);
    c3.displayData();
    return 0;
}
```

24. Write a general form of a function. Explain the different types of argument passing techniques with example.

The General Form of a Function

The general form of a function is:

```
ret-type function-name(parameter list) {
       body of the function
}
```

The ret-type specifies the type of data that the function returns. A function may return any type of data except an array. The parameter list is a comma-separated list of variable names and their associated types that receive the values of the arguments

Different types of argument passing techniques

In C++ there are two ways of passing the function arguments: **i) pass/call by value and ii) pass/call by reference**.

1. Passing by value means that a copy of the object is made on the callee's stack and altering the object means altering a local copy so the caller's object is unchanged when the function returns. In the example below, an integer is passed by value to function foo(), which increments it. Because a copy is made the variable in main remains unchanged and the program's output is 0 1 0.

```
void foo(int i) {
  i++;
  cout << i << endl;
}
int main() {
  int i = 0;
  cout << i << endl;
  foo(i);
  cout << i << endl;
  return 0;
}
```

2. Passing by reference means that the address of the object is sent (a reference holds an address but behaves like an object) so that the callee can directly alter the original object. In this example an integer is passed to foo() by reference, which means that altering it will directly affect the object in main and the output is 0 1 1.

```
void foo(int &i) {
  i++;
  cout << i << endl;
}
int main() {
  int i = 0;
  cout << i << endl;
  foo(i);
  cout << i << endl;
  return 0;
}
```

25. Explain constant member functions and mutable data members with example.

Constant Member Functions: A function becomes constant when const keyword is used in function's declaration. The idea of constant functions is not allow them to modify the object on which they are called. Let us consider this situation. The library programmer desires that one of the member functions of the class should not be able to change the value of member data. This function should be able to merely read the values contained in the data members, but not change them. However, while defining the function one might accidentally write the code to do so. In order to prevent this, if the function is declared as a constant function, then for any attempt to change the value of a data member through the function, the compiler throws an error. An example class Distance with constant member functions is shown below:

```
/* Distance class containing constant member functions*/
class Distance {
       int iFeet;
       float fInches;
       public:
       void setFeet(int);
       int getFeet() const; //constant function
       void setInches(float);
       float getInches() const; //constant function
};
void Distance::setFeet(int x) {
       iFeet=x;
}
int Distance::getFeet() const  { //constant function
       return iFeet++; //ERROR!!
}
void Distance::setInches(float y) {
       fInches=y;
}
float Distance::getInches() const  { //constant function
       return fInches=0.0; //ERROR!!
}
```

Mutable Data Members

A mutable data member is never constant. It can be modified inside constant functions also. Prefixing the declaration of a data member with the keyword mutable makes it mutable. In the following example variable y is declared as a mutable data member. Please note that even if we try to modify y inside a constant member function abc(), the compiler does not give an error and allows the mutable data member in constant member function.

```
// Program to demonstrate Mutable data members
class A {
    int x; //non-mutable data member
    mutable int y; //mutable data member
    public:
    void abc() const { //a constant member function
           x++; //ERROR: cannot modify a non-constant data
           //member in a constant member function
           y++; //OK: can modify a mutable data member in a constant member function
    }
    void def() { //a non-constant member function
           x++; //OK: can modify a non-constant data member
           //in a non-constant member function
           y++; //OK: can modify a mutable data member in a non-constant member function
    }
};
```

26. **What are friend functions? Why it is required? Explain with the help of suitable example**. OR
    Explain the need of friend function in C++. OR
27. **What are friend non-member functions and friend member functions? Explain with suitable examples.**

Define Friend Functions

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions of this class. A friend can be a non-member function, member function of some other class, or another class in which case the entire class and all of its members are friends. To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend.

Friend non-member functions

A friend function is a non-member function that has special rights to access private/protected data members of any object of the class of whom it is a friend. A friend function is prototyped within the definition of the class of which it is intended to be a friend. The prototype is prefixed with the keyword friend. Since it is a non-member function, it is defined without using the scope resolution operator. Moreover, it is not called with respect to an object. An illustrative example is shown below:

```
// Program to illustrate Friend non-member functions
class A {
       int x;
       public:
       friend void abc(A&); //prototype of the friend function
};
void abc(A& a) { //definition of the friend function
       a.x++; //accessing private members of the object
}
void main() {
       A a1;
       abc(a1);
}
```

28. **What are the points to remember about friend function?** OR **What are the characteristics of friend functions explain?**

A few points about the friend functions that we must keep in mind are as follows:

- friend keyword should appear in the prototype only and not in the definition.
- Since it is a non-member function of the class of which it is a friend, it can be prototyped in either the private or the public section of the class.
- A friend function takes one extra parameter as compared to a member function that performs the same task. This is because it cannot be called with respect to any object. Instead, the object itself appears as an explicit parameter in the function call.
- We need not and should not use the scope resolution operator while defining a friend function.

Need of friend function/Why friend functions required?
There are situations where a function that needs to access the private data members of the objects of a class, cannot be called with respect to an object of that class. For example consider the following example depicting a function friendly to two classes.

```
class  ABC;//Forward declaration
class XYZ {
       int x;
       public:
       void setvalue(int i){ x=i;}
       friend void max(XYZ,ABC);
};
class ABC {
       int a;
       public:
       void setvalue(int i){a=i;}
       friend void max(XYZ,ABC);
};
void max(XYZ m, ABC n) { //definition of friend function max
       If(m.x>=n.a)
              cout<<m.x;
       else
              cout<<n.a;
}
```

29. What are friend classes explain with example. OR **What is a friend class? Illustrate friends as bridges.**

Friend Classes: A class can be a friend of another class. Member functions of a friend class can access private data members of objects of the class of which it is a friend. If class B is to be made a friend of class A, then the statement

```
friend class B;
```

should be written within the definition of class A. Following example illustrates this.

```
// Declaring friend classes
class A {
friend class B; //declaring B as a friend of A
       /* rest of the class A */
};
```

It does not matter whether the statement declaring class B as a friend is mentioned within the private or the public section of class A. Now, member functions of class B can access the private data members of objects of class A.

Friend member functions: Some specific member functions of one class can be made friendly to another class. For making only `B::test_friend()` function a friend of class A, in the declaration of the class following statement is added

```
friend void B::test_friend();
```

The modified definition of the class A is

```
class A {
       /* rest of the class A */
       friend void B::test_friend();
};
```

In order to compile this code successfully, the compiler should first see the definition of the class B. Otherwise, it does not know that `test_friend()` is a member function of the class B. Therefore we should put the definition of class B before the definition of class A. However, a pointer of type A * is a private data member of class B. So, the compiler should also know that there is a class A before it compiles the definition of class B. This problem of circular dependence is solved by forward declaration of class A before the definition of class B as shown in the following example.

```
// Forward declaring a class that requires a friend
class A; //Declaration only! Not definition!!
class B {
       A * APtr;
       public:
       void Map(const A * const);
       void test_friend(const int=0);
};
class A
{
       int x;
       public:
       friend void B::test_friend(const int=0);
};
```

Friends as bridges

Friend functions can be used as bridges between two classes. Suppose there are two unrelated classes whose private data members need a simultaneous update through a common function. This function should be declared as a friend to both the classes. See the following example.

```
// Friends as bridges
class B; //forward declaration
class A {
       /* rest of the class A */
       friend void ab(const A&, const B&);
};
class B {
/* rest of the class B */
       friend void ab(const A&, const B&);
};
```

30. Define friend functions. Explain with a C++ program to add two complex numbers.

```
// C++ program to add two complex numbers using friends
#include<iostream>
using namespace std;
class complex {
    float x;
    float y;
public:
    void input(float real, float img) {
              x = real;
              y = img;
     }
    friend complex sum(complex, complex);
    void show(complex);
};

complex sum(complex c1, complex c2) {
    complex c3;
    c3.x = c1.x + c2.x;
    c3.y = c1.y + c2.y;
    return(c3);
}

void complex :: show(complex c) {
    cout << c.x << " + j" << c.y << "\n";
}

int main() {
    complex A, B, C;
    A.input(5.1, 5.25);
    B.input(9.95, 6.2);
    C = sum(A,B);
    cout << "A = "; A.show(A);
    cout << "B = "; B.show(B);
    cout << "C = "; C.show(C);
    return 0;
}
```

31. What are the points to remember about friend function? Device a class MATRIX with a constructor, method to read and display the matrix.

```
// Matrix multiplication using friend function
#include<iostream.h>
class Matrix{
private:
       int a[3][3];
public:
       void getMatrix();
       void printMatrix();
       friend Matrix Multiply(Matrix A,Matrix B);
};
```

```
void Matrix::getMatrix()
{
	cout<<"Enter the Matrix one by one\n";
	for(int i=0;i<3;i++){
		for(int j=0;j<3;j++){
			cin>>a[i][j];
		}
	}
}
void Matrix::printMatrix(){
	cout<<"Given Matrix is\n";
	for(int i=0;i<3;i++) {
		for(int j=0;j<3;j++) {
			cout<<a[i][j]<<"";
		}
		cout<<"\n";
	}
}
Matrix Multiply(Matrix A, Matrix B) {
	Matrix T;
	for(int i=0;i<3;i++){
		for(int j=0;j<3;j++){
			T.a[i][j]=0;
			for(int k=0;k<3;k++){
				T.a[i][j]=T.a[i][j]+A.a[i][k]*B.a[k][j];
			}//end of k loop
		}//end of j loop
	}//end of i loop
	return T;
}
void main() {
	Matrix P,Q,R;
	P.getMatrix();
	Q.getMatrix();
	R=Multiply(P,Q);
	R.printMatrix();
}
```

Output:

```
Enter the Matrix one by one
111
111
111
Enter the Matrix one by one
222
222
222
Given Matrix is
666
666
666
```

32. Write a C++ program to convert temperature on F to C and vice versa using friend functions.

To be written as assignment.

Static Data Member

33. What is a static data member? Explain its general syntax rules and examples.

Static data members of a class hold global data that is common to all objects of that class. Examples of such global data are

- count of objects currently present,
- common data accessed by all objects, etc.

We can define class members static using static keyword. When we declare a data member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static data member. A static data member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class by re-declaring the static variable, using the scope resolution operator :: to identify which class it belongs to.

Let us consider a class Count to count the number of objects of this class currently present in the system. We want every objects of this class to increment the count when it is created and decrement the count when it is destroyed. Therefore, this count data should be globally available to all objects of this class. This data cannot and should not be a member of the objects themselves. Otherwise, multiple copies of this data will be embedded within the objects taking up unnecessary space. Same value would have to be maintained for this data in all objects. This is very difficult. Thus, this data cannot be stored in a member variable of class Count. At the same time, this data should not be stored in a global variable. Then the data is liable to be changed by even non-member functions. It will also potentially lead to name conflicts. This means storing the data in a static variable of the class resolves this conflict. Static data members are members of the class and not of any object of the class, that is, they are not contained inside any object. **Consider the following example to count the number of objects of certain class using static data member (Refer next question)**

34. **Write a C++ program to keep track of number of objects created by a particular class without using extern variable**. OR **Write a C++ program to count the number of objects of certain class.**

```
// program to count the number of objects of certain class using static data member
#include<iostream>
using namespace std;
class Count {
        private:
        static int count; // declared as a static data member

        void printCount() {
                cout<<"There are "<<count<<" Count objects"<<endl;
        }
        public:
        Count() {
                cout<<"Constructor ";
                count++;
                printCount();
        }
        ~Count() {
                cout<<"Destructor ";
                count--;
                printCount();
        }
};
int Count::count = 0; // re-declared and initialized outside the class using :: operator
int main() {
        Count c1, c2, c3;
}
```

35. **Write a C++ program to implement stack of integers having member functions push, pop, display and also a constructor.**
36. **Write a C++ program to define a class called box with length, breadth and height as the data members and input(), print() and volume() as the member function.**
    To be written as assignment.

37. **Explain the term Namespace and Namespace pollution**. OR **How do Namespace help in preventing pollution of the global namespace?** OR **Write a note on Namespaces**.

A **namespace** is a region that provides a scope to the identifiers (the names of types, functions, variables etc.) inside it. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries. All identifiers at namespace scope are visible to one another without qualification. Identifiers outside the namespace can access the members by using the fully qualified name for each identifier, for example `std::cout` or using Directive for all the identifiers in the namespace:

```
using namespace std;
```

Namespace pollution means that code that should really live in separate namespaces is added to a common namespace or global namespace. Namespaces enable the C++ programmer to prevent pollution of the global namespace that leads to name clashes. The term 'global namespace' refers to the entire source code. It also includes all the directly and indirectly included header files. By default, the name of each class is visible in the entire source code, that is, in the global namespace. This can lead to problems. Consider the following example. Suppose a class with the same name is defined in two header files.

```
/*Beginning of A1.h*/
class A { // code };
/*End of A1.h*/

/*Beginning of A2.h*/
class A { // code }; //a class with an existing name
/*End of A2.h*/

#include"A1.h"
#include"A2.h"
void main() {
       A AObj; //ERROR: Ambiguity error due to multiple definitions of A
}
```

If we include both these header files in a program, while creating an object of class A, compiler gives an Ambiguity error due to the multiple definitions of A. Enclosing the two definitions of the class A in separate namespaces overcomes this problem.

```
/*Beginning of A1.h*/
namespace A1 { //beginning of a namespace A1
       class A {
       };
} //end of a namespace A1
/*End of A1.h*/
/*Beginning of A2.h*/
namespace A2 { //beginning of a namespace A2
       class A {
       };
} //end of a namespace A2
/*End of A2.h*/
```

Now, the two definitions of the class A are enveloped in two different namespaces. Thus, the ambiguity encountered in the above listing can be overcome. A revised definition of the main() function is given below.

```
// Enclosing classes in namespaces prevents pollution of the global namespace
#include"A1.h"
#include"A2.h"
void main() {
       A1::A AObj1; //OK: AObj1 is an object of the class
       //defined in A1.h
       A2::A AObj2; //OK: AObj2 is an object of the class
       //defined in A2.h
}
```

38. Explain with an example to illustrate the different features of keyword "Namespace" and "using".

Example showing use of keywords **"Namespace"** and **"using"**

Qualifying the name of the class with that of the namespace can be cumbersome. The using directive enables us to make the class definition inside a namespace visible so that qualifying the name of the referred class by the name of the namespace is no longer required. Consider the following example

```
// The using directive makes qualifying of referred class names by namespaces unnecessary
#include"A1.h"
#include"A2.h"
void main() {
       using namespace A1;
       A AObj1; //OK: AObj1 is an object of the class
       //defined in A1.h
       A2::A AObj2; //OK: AObj2 is an object of the class
       //defined in A2.h
}
```

39. What is a nested class? What is its use? Give an example and explain.

What is nested class?

A class can be defined inside another class. Such a class is known as a nested class. The class that contains the nested class is known as the enclosing class. Nested classes can be defined in the private, protected, or public portions of the enclosing class. In the following example class B is defined in the private section of class A.

```
class A {
    class B {
    /* definition of class B */
    };
    /* definition of class A */
    };
```

What is its use?

A nested class is created if it does not have any relevance outside its enclosing class. By defining the class as a nested class, we avoid a name collision. In the example above, even if there is a class B defined as a global class, its name will not clash with the nested class B. Also the size of objects of an enclosing class is not affected by the presence of nested classes. Nested classes are used in situations where the nested class has a close conceptual relationship to its surrounding class.

Give an example and explain.

For example, with the class String, a class iterator is required which provides all characters that are stored in the string. This Iterator class can be defined as nested class in the class String. In the following example class Iterator is defined in the public section of class String. The member function next() of class Iterator is defined outside the class String.

```
// Defining nested classes
class String {
   public:
   class Iterator {
          public:
          void next(); //prototype only
   };

   /* definition of class String */
};

void String::Iterator::next()
{
   //definition of String::Iterator::next() function
}
/* definitions of the rest of the functions of class Iterator */
```

Constructors and Destructors

40. What are constructors? Explain different types of constructors with suitable examples. OR
    What is a constructor? Explain different types of constructors.
41. How is constructor different from a member function? Illustrate with an example.
42. What are constructors? When are they called? What is there use? Define a suitable parameterized constructor with default values for the class TIME with data members hr, min and sec.
43. What is a copy constructor explain with an example. OR Write a short note copy constructor.
44. What are constructors and destructors? Explain the different types of constructors in C++ with different examples.

What are constructors? When are they called? What is there use?

Constructors are special class functions which performs initialization of every object. **The constructor gets called** automatically by the compiler whenever an object is created. Constructors initialize values to object members after storage is allocated to the object. It appears as member function of each class, whether it is defined or not. It has the same name as that of the class. It may or may not take parameters. It does not return any value.

What is there Use? : Constructors perform the initialization of member data of a class and domain constraints on the values of data members can also be implemented via constructors. The prototype of a constructor is

```
<class name> (<parameter list>);
```

How is constructor different from a member function? Illustrate with an example.

- Constructor name must be same as class name but functions cannot have same name as class name.
- Constructor does not have return type whereas functions must have.
- Member function can be virtual, but, there is no concept of virtual constructor in C++.
- Constructors are invoked at the time of object creation automatically and cannot be called explicitly but functions are called explicitly using class objects.

Example:

```
class Car{
public:
	Car(){ // Constructor
		cout << "Car's Constructor\n";
	}
	void CarAvailable(){
		cout << "Car's Function\n";
	}
};
int main() {
	//Constructor will be invoked automatically during object creation.
	Car obj;
	//Functin can be called using class object, no automatic
	obj.CarAvailable();
	return 0;
}
```

Explain different types of constructors: Constructors are of three types:

1. Zero argument constructors/ Default constructors
2. Parameterized constructors
3. Copy constructors

- **Zero argument/Default constructor:** The constructor that does not take any argument is called zero argument constructors. The constructor provided by default by the compiler also does not take any arguments. The terms 'zero-argument constructor' and 'default constructor' are used interchangeably.

```
// Use of zero argument constructor in class Cube
class Cube {
    int side;
public:
    Cube() {  side=10; }
};
int main() {
    Cube c;
    cout << c.side;
}
```

- **Parameterized Constructors:** These are the constructors with parameters. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument. The parameterized constructor is defined just like any other member function except for the fact that it does not return any value. If the parameterized constructor is provided and the zero-argument constructor is not provided, the compiler will not provide the default constructor. Default values given to parameters of a parameterized constructor make the zero-argument constructor unnecessary.

```
// Use of parameterized constructor in class Cube
class Cube {
    int side;
    public:
    Cube(int x) { side=x; }
};
int main() {
    Cube c1(10);
    Cube c2(20);
    cout << c1.side;
    cout << c2.side;
}
```

- **Copy Constructor:** The copy constructor is a special type of parameterized constructor which copies one object to another. It is called when an object is created and equated to an existing object at the same time. The copy constructor is called for the object being created. The pre-existing object is passed as a parameter to it. The copy constructor member-wise copies the object passed as a parameter to it into the object for which it is called. If the copy constructor for a class is not defined, the compiler defines it. Copy constructor is invoked under the following three circumstances:
  1. When an object is created and simultaneously equated to another existing object, the copy constructor is called for the object being created. The object to which this object was equated is passed as a parameter to the copy constructor. Example:

```
A A1;
A A2=A1; //copy constructor called
// or
A A3(A1) //copy constructor called
// or
A *ptr = new A(A1); //copy constructor called
```

  2. When an object is created as a non-reference formal argument of a function. The copy constructor is called for the argument object. The object passed as a parameter to the function is passed as a parameter to the copy constructor. Example:

```
void abc(A);
A A1;
abc(A1);      //copy constructor called
void abc(A A2) {
 // definition of abc()
}
```

  3. When an object is created and simultaneously equated to a call to a function that returns an object. The copy constructor is called for the object that is equated to the function call. The object returned from the function is passed as a parameter to the constructor

```
A abc(){
       A A1;
       //remaining definition of abc
       return A1;
}
A A2=abc();// //copy constructor called
```

45. **What is the benefit of copy constructor? Explain with example the necessity of defining our own copy constructor though default copy constructor exists.** OR **Write an example class where copy constructor is needed?**

What is the benefit of copy constructor?

The main benefit of the copy constructor is that, it is used to create a new object by duplicating the state of an existing object and this is done by copying the values of the data members of the existing object into the data members of newly created object.

Explain with example the necessity of defining our own copy constructor though default copy constructor exists.

If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member wise copy between objects. The compiler created copy constructor works fine in general. We need to define our own copy constructor only if an object has pointers or any run time allocation of resource like file handle, a network connection etc. Default copy constructor does only shallow copy. Deep copy is possible only with user defined copy constructor. In user defined copy constructor, we make sure that pointers (or references) of copied object point to new memory locations.

An example class where copy constructor is needed?
Following example demonstrates the use of copy constructor. In the String class, we must write copy constructor to make sure that pointers (or references) of copied object point to new allocated memory locations.

```
#include<iostream>
#include<cstring>
using namespace std;
class String {
private:
    char *s;
    int size;
public:
    String(const char *str = NULL); // constructor
    ~String() { delete [] s;  }// destructor
    String(const String&); // copy constructor
    void print() { cout << s << endl; } // Function to print string
    void change(const char *);  // Function to change
};
String::String(const char *str) {
    size = strlen(str);
    s = new char[size+1];
    strcpy(s, str);
}
void String::change(const char *str) {
    delete [] s;
    size = strlen(str);
    s = new char[size+1];
    strcpy(s, str);
}
String::String(const String& old_str) {
    size = old_str.size;
    s = new char[size+1];
    strcpy(s, old_str.s);
}
int main() {
    String str1("GeeksQuiz");
    String str2 = str1;
    str1.print(); // what is printed ?
    str2.print();
    str2.change("GeeksforGeeks");
    str1.print(); // what is printed now ?
    str2.print();
    return 0;
}
```

Output:

```
GeeksQuiz
GeeksQuiz
GeeksQuiz
GeeksforGeeks
```

46. Is overloading of constructors possible? Justify your answer with an example.

Just like other member functions, constructors can also be overloaded. In fact when you have both default and parameterized constructors defined in your class, you are having overloaded constructors, one with no parameter and other with parameter. You can have any number of constructors in a class that differ in parameter list. Following example shows the use of overloaded constructors.

```
// Use of overloaded constructors in class Cube
class Cube {
       int side;
public:
       Cube() {  side=10;  }
       Cube(int x) { side=x; } //overloaded
};
int main() {
       Cube c;
       cout << c.side;
       Cube c1(5);
       cout << c1.side;
}
```

Define a suitable parameterized constructor with default values for the class TIME with data members hr, min and sec.

```
#include<iostream>
using namespace std;
class Time {
        private :
                int hr;
                int min;
                int sec;
        public :
                Time(int h = 0, int m  = 0, int s = 0); //constructor with default value 0
                void print();//print description of object in hh:mm:ss
};
Time :: Time(int h, int m, int s) {
        hr = h;
        min = m;
        sec = s;
}
void Time :: print() {
                cout << setw(2) << setfill('0') << hour << ":"
                << setw(2) << setfill('0') << minute << ":"
                << setw(2) << setfill('0') << second << "\n";
}
int main() {
        Time t1(10, 50, 59);
        t1.print();  // 10:50:59
        Time t2;  //object created with default value
        t2.print();  // 00:00:00
        return 0;
}
```

Output :

```
10:50:59
00:00:00
```

47. What is a destructor?

Destructor: Destructor is the opposite of the constructor in the sense that it is invoked when an object ceases to exist. The definition of a destructor must obey the following rules:

- The destructor has the same name as the class but its name is prefixed by a tilde(~).
- The destructor has no arguments and no a return value.
- Destructors cannot be overloaded.

The destructor for the class Person is declared as follows:

```
// Destructor
class Person {
public:
    Person(); // constructor
    ~Person(); // destructor
};
```

48. Discuss the following with an example

a. Function prototyping
b. Call by reference
c. Default argument
d. Return by reference

Covered in previous questions.

49. Determine the output of the following snippets and comment

(i)

```
class A {
	int x = 10;
	void display() {
		cout << "The value of x = " << x;
	}
	void main() {
		A obj;
		obj.display();
	}
};
```

(ii)

```
class A {

	int pvt;
	int * ptr_pub;
	A() {
		pvt = 25;
		ptr_pub = &pvt;
	}
	void print_private() {
		cout << pvt << endl;
	}
	void main() {
		A objA;
		*objA.ptr_pub = 10;
		objA.print_private()
	}

};
```