**Q5(a). Compare object-oriented programming with procedure-oriented programming.**

**Ans. Difference Between Procedure Oriented Programming (POP) & Object Oriented Programming (OOP) :**

| | Procedure Oriented Programming | Object Oriented Programming |
|---|---|---|
| **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 functions use 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. |

**Q5(b). Write a program which reads marks of a student and display whether the student got first division (marks ≥ 60), second division (marks ≥ 50), third division (marks ≥ 35) or fail (mark < 35).**

**Ans.**

```
#include <iostream.h>
#include <conio.h>
void main( )
{
  int p, m1, m2, m3, m4, m5;
  cout << "Enter marks of students in 5 subjects";
  cin >> m1 >> m2 >> m3 >> m4 >> m5 ;
  p = (m1 + m2 + m3 + m4 + m5)/500 * 100 ;
    if(p > = 60)
    cout << "Student has got 1st division";
  else {
    if(p > = 50)
    cout << "Student has got 2nd division";
  else {
    if(p > = 35)
    cout << "Student has got 3rd division";
```

```
else
  }
  }
  cout << "Student is failed";
getch( );
}
```

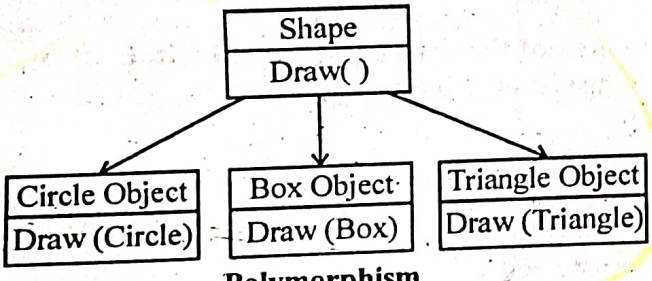
**Q6(a). What is polymorphism? Explain the static and dynamic binding in this.**

**Ans. Polymorphism :**

The term 'polymorphism' is made up of two words - poly and morphism, were '**poly**' means '**many**' and '**morphism**' means '**forms**'. So, collectively the term means many forms or the ability to take many forms. When operators and methods are used in different ways, depending on what they are operating on, it is called Polymorphism (one thing with several distinct forms).

It permits us to create multiple definitions for operators and functions.

It refers to identically named methods (member functions) that have different behaviour depending on the type of object they refer to. It is the process of defining a number of objects of different classes into a group and calls the methods to carry out the operation of the objects using different function calls. In other words, polymorphism means to carry out differing processing steps by function having same messages. It refers to the runtime finding to a pointer to a method.

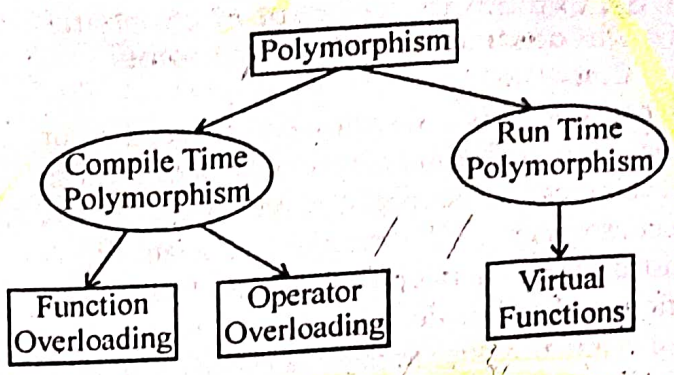



**Polymorphism**

By inheritance, every object will have this procedure. Its algorithm is, however unique to each object and so the draw procedure will be redefined in each class that defines the object. At run time, the code matching the object under current reference will be called.

Polymorphism is of two types.

1. Compile time polymorphism.
2. Run time polymorphism.




**1. Compile Time Polymorphism :**

In compile time polymorphism, during compilation time, the C++ compiler determines which function is to be executed based on the parameters passed to the function or the function return type. The compiler then substitutes the correct function for each invocation. It is also known as '**static binding**' or '**early binding**' or '**static linkage**'.

The compile time polymorphism is achieved using– **operator overloading** and **function overloading**. The process of making an operator to exhibit different behaviour in different instances is called **operator overloading**.

The process of using a single function name to perform different types of tasks in known as **function overloading**.

**2. Run Time Polymorphism :**

In run time polymorphism, the function that is to be chosen for execution is done during execution time. The appropriate member function is selected while the program is running. The object of a class must be declared either as a pointer to a class or reference to a class. It is also known as **late binding** or **dynamic linkage** or **dynamic binding**.

The runtime polymorphism is achieved using **virtual function**. The virtual function is a function that does not exists in reality, but appears real to some parts of a program. The role of keyword '**virtual**' in polymorphism is mainly to achieve run time polymorphism by making the member function of a base class as a virtual function by placing keyword in front of member function name. When a function is made virtual, C++ determine which function to use at run time based on the type of object pointed to by the base pointer rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the virtual function.

**Q6(b). Explain the concept of constructors and destructors with examples.**

**Ans. Constructors :**

A constructor is a special member function for automatic initialization of an object. Whenever an object is created, the special member function, i.e., the constructor will be executed automatically. A constructor function is different from all other non-static member functions in a class, because it is used to initialize the variables of whatever instance being created.

**Syntax rules for writing constructor functions :**

1. A constructor name must be the same as that of its class name.
2. It is declared with no return type.
3. It cannot be declared const or volatile, but a constructor can be invoked for a const and volatile objects.
4. It may not be static.
5. It may not be virtual.
6. It should have public or protected access within the class and only in rare circumstances, it should be declared private.

**Syntax of the constructor :**

```
class user_name
{
 private :
  _______
  _______
 protected :
  _______
  _______
 public :
  user_name( );
  _______
  _______
};
user_name : : user_name( )
{
 _______
 _______
}
```

**For Example,** Constructor declaration in a class definition

```
Class employee
{
 private :
  char name[20];
  int ecode;
  char address[20];
 public :
  employee( );
  void getdata( );
  void display( );
};
employee( ) : : employee( );  // constructor
{
 _______
 _______
}
```

**Destructor :**

A destructor is a function that automatically executes, when an object is destroyed. A destructor function gets executed whenever, an instance of the class to which it belongs goes out of existence. The primary usage of the destructor function is to release space on the heap. A destructor function may be invoked explicitly.

**Syntax rules for writing a destructor function :**

1. A destructor function name is the same as that of the class it belongs, except that the first character of the name must be a **tilde (~).**
2. It is declared with no return types, since it cannot ever return a value.
3. It cannot be declared static, const or volatile.
4. It takes no arguments and therefore cannot be overloaded.
5. It should have public access in the class declaration.

**Syntax of the destructor function in C ++ is :**

```
Class user_name
{
 private :
   //data variables
   //methods
 protected :
   //data
 public :
   user_name( );  // constructor
   ~user_name( );  // destructor
   //methods
};
```

**Example of destructor function declaration :**

```
class employee
{
 private :
  char name[20];
```

```
int ecode;
char address[20];
public :
 employee( );
 ~ employee( );
  void getdata( );
  void display( );
};
```

**Q7. What is inheritance? What are the different types of inheritance? Explain how multiple inheritance is implemented in C++ with example.**

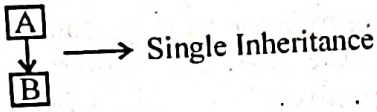
**Ans. Inheritance :**

It is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification.

In OOP, the concept of inheritance provides the idea of **reusability**. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. The real power of the inheritance mechanism is that it allows the programmer to reuse a class that is almost, but not exactly and to tailor the class in such a way that, it does not introduce any undesirable side-effects into the rest of the classes.

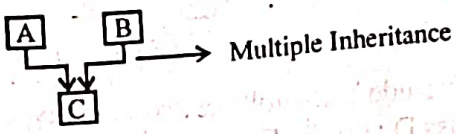
**Types of Inheritance :**

**(i). Single Inheritance :** A derived class with only one base class **m** is called single Inheritance.



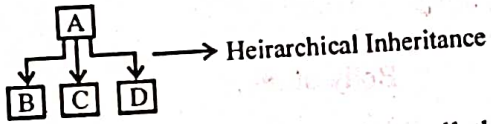

```
class B
{
   int a ;
  public :
    int a ;
    void get_ab( ) ;
};
class D : public B.
{
   int c ;
  public :
    void mul(void) ;
    void display(void) ;
} ;
```

**(ii). Multiple Inheritance :** A derived class with several base classes is called multiple inheritance. Multiple inheritance allows us to combine the feature of several existing classes as a starting point for defining new classes.



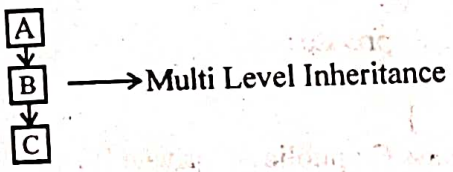

```
class B
{
    int a ;
   public :
    int a ;
    void get( ) ;
}
class C
 {
    int c ;
   public :
     void mul( ) ;
     void display( ) ;
 }
     class D : public B, public C
     { ≡
     }
```

**(iii). Hierarchal Inheritance :** The traits of one class may be inherited by more than one class.



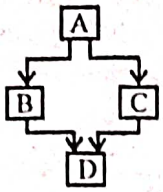

In C++, the base class will include all the features that are common to the subclasses. A subclass can be constructed by inheriting the properties of the base class.

**(iv). Multi Level Inheritance :** The mechanism of deriving a class from another 'derived class' is known as multilevel inheritance. **Class A** serve as a base class for the derived **Class B** which in turn serve as a base class for the derived **Class C**.




**(v). Hybrid Inheritance :** Hybrid inheritance support the concept of hierarchical inheritance and

multiple inheritance. It can be shown with the help of following figure :




It include both multiple and multilevel inheritance,

```
class D : public B, public C
{
}
```

**Multiple Inheritance :**

A class can inherit the attributes of two or more classes. This is known as multiple inheritance. It allows us to combine the features of several existing classes as a starting point for defining new classes.

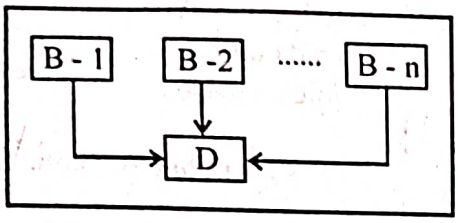



**Fig : Multiple Inheritance**

The syntax of a derived class with multiple base classes is as follows :

```
Class D : Visibility B-1, Visibility B-2
  {
    .....
      Body of D
    .....
  };
```

Where, visibility may be either **public** or **private**.

Multiple Inheritance have **tree-like** structure

```
class A
  {
  protected :
    int a ;
  }
class B : public A
  {
    private :
    int a ;
  }
class C : public A , private B
  { protected :
    int a ;
  }
```

**Q8(a). What is a virtual function? Illustrate with an example, the usage of virtual functions.**

**Ans. Virtual Function :**

It is a function qualified by **Virtual** Keyword. Virtual functions implement polymorphism, whereby objects belonging to different classes can respond to the same message in different ways.

When we use the same function name in both the base and derived classes, the function in base class is declared as **virtual** using the keyword **virtual** preceding its normal declaration. When a function is made virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointers to point to different objects, we can execute different versions of the virtual function.

**An example,** where virtual functions are implemented in practice. Consider a book shop which sells both books and video tapes. We can create a class known as **media** that stores the **title** and **price** of a publication. We can then create two derived classes, one for storing the number of pages in a book and another for storing the playing time of a tape.

The classes are implemented in Program. A function display( ) is used in all the classes to display the class contents. The function **display()** has been declared **virtual** in **media**, the base class.

In the **main** program, we create a heterogeneous list of pointers of type media as shown below :

```
media * list[2] = {&book1, &tape1};
```

The base pointers list[0] and list[1] are initialized with the addresses of objects **book1** and **tape1** respectively.

**Example :**

```
#include <iostream.h>
#include <cstring>
using namespace std;
class media
{ protected :
    char title[50];
    float price;
  public :
    media(char*s, float a)
    {
        strcpy(title, s);
        price = a;
    }
```

```
  virtual void display() {}  // empty virtual function
};
class book : public media
{
  int pages;
  public:
    book(char*s, float a, int p) : media(s, a)
    {
        pages = p ;
    }
  void display( );
};
class tape : public media
{
  float time ;
  public:
    tape(char*s, float a, float t) : media(s, a)
    {
        time = t;
    }
  void display( );
}
void book :: display( )
{
  cout << "\n Title :" <<title;
  cout << "\n Page :" <<pages;
  cout << "\n Price :" <<price;
}
void tape : : display( )
{
  cout << "\n Title :" <<title;
  cout << "\n Play time :" <<time<< "mins";
  cout << "\n Price :" <<price;
}
int main( )
{ char *title = new char[30];
  float price, time;
  int pages;
  // Book details
        cout << "\n ENTER BOOK DETAILS \n";
        cout << "Title :" ; cin >> title ;
        cout << "Price :" ; cin >> price ;
        cout << "Pages :" ; cin >> pages ;
  book book1(title, price, pages);
  // Tape details
        cout << "\n ENTER TAPE DETAILS \n";
        cout << "Title :" ; cin >> title ;
        cout << "Price :" ; cin >> price ;
        cout << "Play time(mins) :" ; cin >> time ;
  tape tape1(title, price, time);
        media*list[2];
        list[0] = &book1;
        list[1] = &tape1;
  cout << "\n MEDIA DETAILS";
  cout << "\n .....BOOK....." ;
  list[0]--> display( );     //display book details
  cout << "\n .....TAPE....." ;
  list[1] --> display( );    //display tape details
  return 0;
}
```

**Output :**

```
ENTER BOOK DETAILS
Title : Programming_In_ANSI_C
Price : 88
Pages : 400
ENTER TAPE DETAILS
Title : Computing_Concepts
Price : 90
Play time (mins) : 55
MEDIA DETAILS
.....Book.....
Title : Programming_In_ANSI_C
Pate : 400
Price : 88
.....Tape.....
Title : Computing_Concepts
Play time : 55 mins
Price : 90
```

**Q8(b). Write short notes on the following:**

**(i) Operator overloading**
**(ii) Generalization**
**(iii) Meta data**

**Ans. (i) Operator Overloading :**

One of the most interesting features of C++ is that it makes the user-defined data types behave as inbuilt data type. In this context, it should be permitted to perform operations on user-defined data types (objects) just like they are performed on basic data types. This is possible, if special meaning is given to the operator. Thus, the mechanism in which an operator performs additional task, apart from doing its normal operation is known as Operator Overloading. It is important to remember the following points about operator overloading :

**(a).** Only the 'semantics' of an operator are extended, its 'syntax' cannot be changed. The syntax includes the grammatical rules that

govern the use of operator such as the number of operands, precedence and associativity.

**(b).** The operators which cannot be overloaded, are class member access operator (.and.*), scope resolution operator (::), size operator (sizeof) and conditional operator (?).

**(c).** The original meaning of operator is not lost **for example,** the operator + cannot be used to multiply two objects.

**Ans. (ii) Generalization :**

Generalization is the relationship between a class and one or more refined versions of it. The class being refined is called the **super class** & each refined version is called **sub-class.** Attributes and operations common to a group of subclasses are attached to the superclass and shared by each subclass. Each subclass is said to inherit the features of its superclass.

It is a useful construct for both conceptual modeling and implementation. Generalization is the "**a-kind-of**" relationship because each instance of a subclass is an instance of the superclass as well. It relates to classes. A generalization tree is composed of classes that describe an object. It is sometimes called an "**or-relationship**".

This symbol is used for generalization.

**For example,** Pump inherits attributes manufacturer, weight and cost from equipment. Generalization is sometimes called "**is-a**" relationship because each instance of a subclass is an instance of superclass also.

**Ans. (iii) Meta Data :**

Meta data is data that describes other data. **Example,** the definition of a class is meta data. Models are inherently meta data, since they describe the things being modeled. Many real-world applications have metadata, such as parts catalog, blue prints and dictionaries. Computer language implementations also use metadata heavily.

Relational database management systems also use meta data. A person can define database tables for storing information. Similarly, a relational DBMS has several meta tables that store table definitions.

Thus, a data table may store the fact that the capital of Japan is Tokyo, the capital of Thailand is Bangkok and the capital of India is New Delhi. A metatable would store the fact that a country has a capital city.

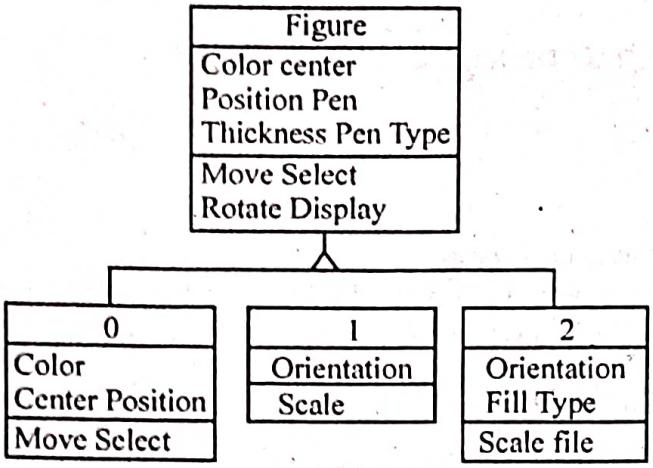



**Figure : Generalization**

**Classification:** Classification signifies that object with the same attribute and behaviors are grouped together into a class. Each class defines a possibly infinite set of individual object. Each object is said to be instance of its class. Each instance of a class has its own value for each attribute but share the same attribute names, behaviors with other instance of that class. Such as monitor on desk is an instance of monitor class with its own values of attribute such as name, size, type. An object contain an implicit reference to its own class.

### Q2(a). What is object oriented programming? What are its advantages and application area?

**Ans. Object-Oriented Programming :**

Object-oriented programming (OOP) is a programming paradigm that represents the concept of "objects" that have data fields and associated procedures known as methods. Objects, which are usually instance of classes, are used to interact with one another to design applications and computer programs. Object-oriented programming is an approach to designing modular, reusable software systems. Object oriented programming may be seen as a collection of co-operating objects as opposed to a traditional view in which a program may be seen as a collection of functions, or simply as a list of instructions to the computer. In OOP each object is capable of receiving messages, processing data and sending messages to other objects.

**Advantages :**

**1. Code Reuse and Recycling:** Objects created for object oriented programs can easily be reused in other programs.

**2. Design Benefits:** Large programs are very difficult to write. Object oriented program force designers to go through an extensive planning phase, which makes for better designs with less flaws. In addition once a program reaches a certain size, object oriented programs are actually easier to program than non – object oriented ones.

**3. Encapsulation:** Once an object is created, knowledge of its implementation is not necessary for its use. Objects have the ability to hide certain parts of themselves from programmers. This prevent programmers from tempering with values they shouldn't.

**4. Software Maintenance:** Programs are not disposable. Legacy code must be dealt with on a daily basis, either to be improved upon (for a new version of on exist piece of software) or made to work with newer computers and softwares. An object – oriented program is much easier to modify and maintain than a non – object oriented program. So although a lot of work is spent before the program is written, less work is needed to maintain it overtime.

**Application area of OOP :**

Main application areas of OOP are :

- User interface design such as windows, menu.
- Real time systems.
- Simulation and Modeling
- Object oriented databases.
- AI and Expert system.
- Neural Networks and parallel programming
- Decision support and office automation system etc.

### Q2(b). How are the data and functions organized in an object oriented program?

**Ans.** Object oriented programming requires to apply the good programming practices of modularity, data encapsulation and data abstraction. Object oriented programming then becomes a process of creating these objects and defining how the objects interact with each other.

The objects are used to create classes. A class is organized almost exactly like a data structure in C. A class contain any number of variables like a data structure, but it also contains all of the functions that work on that data.

To define a class you must create two files. The first is a header file, called the interface or specification file. This interface file contains the definition of the class, using syntax similar to that for defining a data structure. It contains all of the

class variables, and the prototypes for the functions that are part of the class. The second file, called the implementation contains all of the functions that are members of that class. BTW: the variables in a class are referred to as member variables and the functions in a class are referred to as member functions.

**Example:**

```
#include<iostream.h>
#include<conio.h>
void main ( )
{
int x, y;
int mul (int a , int b): //function prototype
clrscr ( );
cout << "enter the value of first no.;
cin >>x;
cout <, enter the value of second no.;
cin >> y;
cout <<"the multiplication is: <<endl;
int m = mul (x, y); // calling function.
cout << m << endl;
getch( );
}
int mul (int a , int b)
{
int temp = a * b;
return (temp);
}
```

### Q3(a). What are the advantages of using new operator as compared to the function malloc()?

**Ans.**

The basic differences between the two is that malloc exist in C- language while new is a specific feature of the C++ language. Malloc is also a function (which for programmers requires some time to execute) while new is an operator program (which cuts the execution time). This is a benefit from the operator new because programmers doing real time programming have a need to choose the fastest method to use.

Operator new is type – safe unlike the function malloc. Function malloc is library function for the C – language. All it does is allocate memory and return a pointer to it. On the other hand, operator new is a language – level construct, and its job is to instantiate an object by allocating memory and calling the appropriate constructors.

The two also have different ways in dealing with failure and memory exhaustion. It function malloc fails, it returns with a NULL pointer. Operator new never returns a NULL pointer but indicates a failure by throwing an exception instead. This is an advantage for the operator new since the computer programmer doesn't have to check the returned pointer every time it calls new nor the computer programmer will miss such an exception.

Function malloc doesn't construct an object (called constructor of object) but operator new does. Another advantage of new is that the operator can be overloaded while function malloc cannot – operator new requires a specific number of objects to allocate while function malloc required specifying the total number of bytes to allocate.

Operator new has leverage over function malloc by returning the exact data type while the function returns void*. Function malloc cannot be overloaded as compared to operator new wherein overloading is possible. Another feature of operator new is that there are less chances of making mistakes with the operator compared to function malloc. Function malloc and free can be used in C++, but it cannot be used directly. It is advisable to use new and delete instead.

### Q3(b). Why is any array called a derived data type?

**Ans.** An array is a derived data type because it cannot be defined on its own, it is a collection of basic data types usually, such as integers, doubles, floats, booleans etc. In object oriented language we can have our own class which can be basis of an array. An array type can be formed from any valid completed type. Completion of an array type requires that the number and type of array members be explicitly or implicitly specified. The member types can be complete in the same or a different compilation unit. Array cannot be of void or function type, since the void type cannot be completed and function types are not object types requiring storage typically, arrays are used to perform operations on same homogenous set of values. The size of the array type is determined by the data type of the array and the member of elements in the array. Each element in an array has the same type.

**For example :**

```
char x[ ] = "Hi!"        /*Declaring an array x */;
```

The size of array is determined by its initialization. An array is allocated contiguously in memory and cannot be empty. An array can have only one dimension. To create an array of "two dimension", declare an array of arrays and so on.

**Q3(c). How many data types and operators are used in C + +? Explain in brief.**

**Ans. Data types in C++:** Basic data types in C++ are :

**1. Integer Type:** Integers are numbers with no fractional part such as 2, 98, - 528 and 0. There are lots of integers, assuming we consider an infinite number to be a lot, so no finite amount of computer memory can represent all possible integers.

| TYPE | BYTE | RANGE |
|---|---|---|
| int | 2 | - 32768 to 32767 |
| Unsigned int | 2 | 0 to 65535 |
| Signed int | 2 | - 32768 to 32767 |
| Short int | 2 | - 32768 to 32767 |
| Unsigned short int | 2 | 0 to 65535 |
| Signed short int | 2 | - 32768 to 32767 |
| Long int | 4 | - 214748 3648 to 21474 83647 |
| Signed long int | 4 | - 214748 3648 to 2147 483647 |
| Unsigned long int | 4 | 0 to 42949667295 |

Variable can be declared as: - int a, b; long int x, y; unsigned long int z;

**2. Floating Type:** Those numbers, which contains fractional part, also called as floating type data. This type of data can be represented in mantissa and exponent form. Example: $51.45$ , $21\ e-7$

| TYPE | BYTE | RANGE |
|---|---|---|
| Float | 4 | $3.4\ E-38$ to $3.4E+38$ |
| Double | 8 | $1.7\ E-308$ to $1.7\ E+308$ |
| Long double | 10 | $3.4\ E-4932$ to $1.1\ E+4932$. |

Variables can be declared as: float a; double x; long double y, z;

**3. Character type:** Character variable are declared by using type specifier char. Variable can be declared as char ch;

| TYPE | BYTE | RANGE |
|---|---|---|
| Char | 1 | - 128 to 127 |
| Unsigned char | 1 | 0 to 255 |
| Signed char | 1 | - 128 to 127 |

**Operator In C++ :** In C++ operator can be classified as:

1. **Arithmetic operator:** Arithmetic operators are those mathematical operator, which are used for adding, subtracting, multiplying, dividing and other operations. **Example:** *, /, +, -, %.
2. **Comparison and relational operator:** These are used for comparing two operands (data). The operator given the result false or true. **Example:** <, >, <=, >=, ==, !=.
3. **Logical operator:** Logical operators are symbol that are used to combine or negate expression containing relational operator. **Example:** &&, !! ,!
4. **Bitwise Operator:** Bitwise operator perform operation on the bit level. Bitwise operator is one of the salient features of C++. Bitwise operators are as: &, !, >>, <<, ~.
5. **Assignment operator:** We can assign a value or variable to another variable in C++ there are so many assignment operator but most important one is = assignment operator. **Example:** a = 30 means we are assigning 30 to a.
6. **Special operator:** In C++ there are some special types of operator which perform particular type of operation.

- ***(Pointer Operator):** is used to get the content of the address operator pointing to a particular variable.

- **& (Address operator):** The address operator is used to get address of any variable.
- **:: (Scope operator):** It help to access a global variable within a particular function.
- **Size of Operator:** Size of operator calculates the size of a variable.

**Q4(a). What is function? What are the advantages of function prototypes in C++ ? When will you make a function Inline? Why?**

**Ans. Function:** A function groups a number of program statement into a unit and give its name. This unit can then invoked from other part of program.

In C++, the main ( ) itself is a function. The main function invokes the other functions to perform its various tasks. The function may or may not transfer back, value to a called function block.

**Structure of a function:** The general syntax of the function definition is

```
function_return_type function_name (data_type arg 1, datatype arg 2....)
{  —
   body of function
   —
   return (value);
}
```

- **Function_return_type:** In place of this we write the type of value which return by the function to the calling portion of the program for example int, float, char etc. If functions return nothing to the calling portion of the program then we write void in place of function return type.
- **Function_name:** Function name can be any name conforming to the syntax rules of identifier.
- **Data_type and Arg:** Arg means argument. The data type of argument depends on the application of function. An argument is a data passed from a program to a function. The data_type of arg tell about the type of arg.
- **Body of function:** The body of function is same like the body of main ( ) program. Statements or a block of statements enclosed between two braces; { and }
- **Return statement:** The return is a reserve word. It is used to terminate function and return a value to its caller.

**Advantage of Function Prototype :**

The prototype describes the function interface to the compiler by giving such as the number and type of argument and type of return values in C++ prototype is must.

For example in place of 'int mul (int a, int b); we can write

'int mul (int, int); . This means the name of argument is optional in Prototype declaration but the type is must. Each argument variable must be declared independently inside parentheses.

The general syntax of prototype declaration is :

type function_name (argument list);

**Inline Function :**

When we call a function control jump to function and come back to caller program then execution of function is completed. This takes lot of time. One solution of this problem is to use macro. Usual error checking does not occur during compilation in macro because they are not function. In C++ a different solution of above is also possible, that is inline function. When we call inline function compiler replace function call with the corresponding function code but control not jump to function.

**Declaration of inline function:**

Inline function declare like follows :

```
inline function_name (argument list)
{
  —
}
```

**Q4(b). Write a program to calculate the factorial of a given number.**

**Ans.**

```
# include < iostream.h>
#include < conio.h>
 oid main ( )
{
    clrscr ( ); // clear the screen
    int no, fact = 1, i; // declare variable type int
    cout << "enter the number. <<endl;
    cin >> no;
    // calculate the factorial
    i = 1
    while (i < = no)
    {
       fat = fact * i;
       i++;
    }
    cout << "the factorial is" << fact;
    getch ( );
}
```

**Q5(a). What is the difference between Class and Structure? What is class and object and how does it accomplish data hiding?**

**Ans. Difference between Class & Structure :**

Classes have a variety of permission customizations that help user define the level of permission that they would like to retain with their objects and object data. A class is defined by default as private, with a slight modification. The program body within which it is defined can access it, but access to the variable is forbidden. A class can additionally be defined as public, private or protected, which changes the level of permission applied to it. Protected classes can be accessed from within classes in the same package only.

Structure, on the other hand, when created, are always public. This applies even to the variables within them. What this means is that access to both the structure, as well as its contents is possible from any program body, even one that is different from the current one. In a way, that makes the variables a bit exposed.

**Class :**

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions with a class are called members of the class.

When you define a class, you define a blueprint for a data type. A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declaration.

For example, we defined the Box data type using the keyword class as follows:

```
Class Box
{
      public:
      double length; // length of box
      double breadth; // breadth of box
      double height; // height of box
};
```

The keyword public determines the access attributes of the members of the class that follow it.

**Objects :**

When class is defined, only specification for object is defined. Object has same relationship to class as variable has with the data type. A class provides the blue prints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types.

Following statements declare two objects of class Box:

```
Box Box1; // declare Box1 of type Box.
Box Box2; // declare Box2 of type Box.
```

Both of the objects Box1 and Box2 will have their own copy of data members.

In C++, data hiding is enforced in classes using private data member. It means that we can't directly access the private data members of a class.

**Q5(b). What is a friend function? What are the merits and demerits of using friend functions?**

**Ans. Friend Function :**

A friend functions are special functions which can access the private members of a class. 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.

A friend can be a function, function template or member function, or a class or class template, 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 as follows:

```
class box
{
        double width;
  public:
        double length;
        friend voidprint width (Box box);
        void setwidth (double wid);
};
```

**Friend functions have the following properties:**

1. Friend of the class can be member of some other class.
2. Friend of one class can be friend of another class or all the classes in one program such a friend is known as Global friend.
3. Friends are non – members hence do not get "this" pointer.
4. Friends, can be friend of more than one class, hence they can be used for message passing between the classes.
5. Friend can be declared anywhere (in public, protected or private section) in the class.

The advantage of friend function is that they provide a degree of freedom in the interface design options.

The disadvantage of friend functions is that they require an extra line of code when you want dynamic binding.

**Q6. What is constructor and destructor? Is it mandatory to use constructor in a class? Can we have more than one constructors in a class? If yes, explain with program, and if not, give the reason.**

**Ans. Constructor :**

A constructor is a special member function whose task is to initialize the object of its class. The name of the member function is the same as the class name. The name given to this initializing function is constructor because it constructs the value of the data member of a class. Constructor is creation and initialization of the object. It is special type of member function for automatic initialization of object. Whenever an object is created the constructor will executed automatically.

The name of constructor must be same as that of its class. The constructor is declared with no return type, not even void. Constructor may not be static and virtual. Constructor should be declared in the public section, it can be declared within the protected and in some rare case within the private. The object with a constructor cannot be used as a member of union.

```
class abc
{
   int x, y;
   public:
   abc( )
   {
      x = 10; y = 20;
   }
   void output ( )
   {
      cout << a <<b << endl;
   }
}
```

In the above example abc is class, the constructor is

```
abc ( )
{
      x = 10; y = 20;
}
```

**Characteristics of a Constructor :**

- It is declared in the public section of the class.
- It automatically initializes the data members of the object.
- It does not return any value.
- Constructor cannot be inherited, through a derived class can call the base class constructor.

**Destructor :**

A destructor as the name implies is used to destroy the object that has been created by a constructor. It is a member function like constructor whose name is the same as the class name but is preceded by tilde.

```
~construct ( ) { }
```

**Characteristics of Destructor :**

- It never takes any arguments
- It does not return any value.
- It is invoked implicitly by the compiler upon exit from the program.

To allocate memory new operator is used. Similarly to free the memory we use delete operator.

**Q2(b). What do you mean by variable and constant? How many datatypes of operators are used in C++? Explain in brief. 7**

**Ans. Variable :**

*A variable is a temporary container to store information.* It is a named location in computer memory where varying data like numbers and characters can be stored & manipulated during the execution of the program.

There are two types of variables :

1. Local variables.
2. Global variables.

**Example :**

```
int my_Age;
int my_Name;
```

**Constant :**

A **constant** is an unchanging value. It's value remains constant throughout the execution of program.

**Types of Constant :**

1. Integer constant
2. String constant
3. Character constant

**Basic built in data types are :**

1. Boolean
2. Character
3. Integer
4. Floating point
5. Double floating point
6. Valueless (void)

***Several of the basic data types can be modified using one of or more of these type modifiers :***

1. Signed 2. Unsigned
3. Short 4. Long

***The types of operators are as follows :***

1. Arithmetic operators.
2. Relational operators.
3. Logical operators.
4. Bitwise operators.
5. Assignment operators.

*For More Information*

*[Please Refer Q3(c). 2013, Page-108]*

**Q3(a). What is Constructor and Destructor? Explain some characteristics of constructor. Also explain the types of constructor with proper syntax. 7**

**Ans. Constructor :**

*A class constructor is a special member function of a class that is executed whenever we create new objects of that class.*

A constructor will have the same name as the class and it does not have any return type at all not even void. Constructors are useful in initialization for certain member variables.

```
class A
{
        int x;
        public:
        A( );
};
```

**Characteristics of Constructor :**

*[Please Refer Q6. 2013, Page-111]*

**Types of constructors :**

1. Default constructor.
2. Parameterized constructor.
3. Copy constructor
4. Dynamic Constructor

*[Please Refer Q13. Unit-V, Page-68]*

**Destructors :**

*Destructors are special class function which destroys the object as soon as the scope of the object ends. The destructor is called automatically by the compiler.* Destructor will never have any arguments.

```
class.name( )
{
        ~destructor( ); // same name as class name.
};
```

**Characteristics of Destructor :**

*[Please Refer Q6. 2013, Page-111]*

**Q3(b). Write a program of constructor overloading an also use the destructor. 7**

**Ans.**

```
class student
{
        int roll_no;
        string name;
        public:
        student(int x)
        {
                roll_no = x;
                name = "NONE";
        }
```

```
    student(int x, string str)
    {
        roll_no = x;
        name = str;
    }
    ~ student( )
    {
        cout<<"Destructor called";
    }
};
int main( )
{
    student a(10);
    student b(11, "Ram");
    return 0;
}
```

**Q4. Write short notes on any four of the following : 3½ each**

**(a) Scope resolution operator**
**(b) Functional model**
**(c) Dynamic model**
**(d) Storage classes**
**(e) Function overloading**
**(f) Static data member and static member function**

**Ans(a). Scope Resolution Operator :**

Scope resolution operator (::) is used to define a function outside a class or when we want to use global variable but also has local variable with the same name.

**Example :**

```
class A
{
    public:
    void output( );
}
void A :: output( )
{
    cout<<"ABC";
}
```

**(b) Functional Model :**

Functional modelling gives the process perspective of the object oriented analysis model and an overview of what the system is supposed to do. It defines the function of the internal processes in the system with the aid of data flow diagrams. It depicts the functional derivation of the data values without indicating how they are derived when they are computer or why they need to be computed.

**(c) Dynamic Modelling :**

It represents the time dependent aspects of the system. It is concerned with the temporal changes in the states of the objects in a system. The main concepts are :

1. State
2. Transition
3. Event
4. Action
5. Concurrency of transactions

**(d) Storage Classes :**

A storage class defines the scope and lifetime of variables and/or functions within a C++ program. These specifies precede the type that they modify. Types are :

1. **Auto :** It is the default storage class for all local variables.
2. **Register :** It is used to define local variables that should be stored in a register instead of RAM.
3. **Static :** It instructs the compiler to keep local variables in existence during the lifetime of the program instead of creating & destroying it each time it comes & goes out of scope.
4. **Extern :** It is used to give reference of a global variable that is visible to all program files.

**(e) Function Overloading :**

We can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list.

**Example :**

```
class A
{    public:
        void print(int i)
        {
            cout<<i;
        }
        void print(double f)
        {
            cout<<f;
        }
};
```

**(f) Static Data Member & Static Member Function :**

**Static Data Member :** When we declare a member of a class static it means no matter how

many objects of the class are created, there is only one copy of the static member. A static member is shared by all objects of the class.

**Static Member Function :** By declaring member function as static we make it independent of any particular object of the class. It can be called even if no objects of the class exist & they are accessed using class name.

**Q5(a). What is operator overloading? Give some limitation of operator overloading. Write a program to overload a preincrement '++' operator by using member function. 8**

**Ans. Operator Overloading :**

One of the most interesting features of C++ is that it makes the user-defined data types behave as inbuilt data type. In this context, it should be permitted to perform operations on user-defined data types (objects) just like they are performed on basic data types. This is possible if special meaning is given to the operator. Thus, the mechanism in which an operator performs additional task, apart from doing its normal operation, is known as **Operator Overloading.**

We can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user defined types as well. Overloaded operators are function with special names the keyword operator followed by the symbol for the operator being defined. Overloaded operator has a return type & a parameter list.

```
Box operator + (const Box&);
```

*For More Information*
*[Please Refer Q28. Unit-V, Page-80]*

**Limitations of Operator Overloading :**

1. Invention of new operators is not allowed.
2. Neither the precedence nor the number of arguments of an operator may be altered.
3. The following operators cannot be overloaded:
   (a) Direct member access operator.
   (b) Scope resolution operator (::)
   (c) Conditional operator (?:)
   (d) Size of operator (sizeof)

**Program to overload a preincrement '++' operator by using member function :**

```
class MyIncClass
{     public:
        int m_nCounter;
      public:
        MyIncClass & operator ++( )
        {
              ++this → m_nCounter;
              return + this;
        }
};
void main( )
{
      MyIncClass counter;
      ++ counter;
}
```

**Q5(b). What is Friend Function? Explain the process of declaring friend function. Also write a program of friend function. 6**

**Ans. Friend Function :**

***A friend functions are special functions which can access the private members of a class. 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.

***If a function is defined as a friend function then the private and the protected data of class can be accessed from that function.***

Friend functions are actually not class member function. They are made to give private access to non-class functions.

**Declaration of friend function :**

The declaration of the friend function should be made inside the body of the class starting with keyword friend.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend as follows :

```
class box
{
      double width;
  public:
      double length;
      friend voidprint width (Box box);
      void setwidth (double wid);
};
```

**Program containing friend function :**

```
#include<iostream>
using namespace std;
class Distance
```

**Q6(a). What is the difference between while and do-while loop? Explain.** 7

**Ans. Difference between 'while' and 'do-while loop' :**

1. In **while** loop the condition is tested first and then the statements are executed if the condition is true. In **do while** the statements are executed for the first time & then the conditions are tested, if the condition is true then the statements are executed again.
2. A **do-while** runs atleast once even though the condition given is false, **while loop** does not run in case the condition given is false.
3. **While loop** is entry control loop whereas do **while** is exit control loop.

**This can be explained in tabular form as follows :**

| | While Loop | Do...While Loop |
|---|---|---|
| 1 | The while is an entry controlled loop statement | 1 The do...while is an exit controlled loop statement |
| 2 | **Its Syntax :**<br>while (test condition)<br>{<br>body of the loop<br>} | 2 **Its Syntax :**<br>do<br>{<br>body of the loop<br>} while (test condition); |
| 3 | In while loop the test condition is evaluated and if condition is true then body of the loop is executed. | 3 On reaching the do statement the program proceeds to evaluate the body of the loop first. At the end of the loop, the test condition in while statement is executed. |
| 4 | **Example:**<br>sum =0 ; n =1;<br>While (n < =10)<br>{<br>sum = sum + n*n ;<br>n =n + 1;<br>}<br>cout <<"sum = " << sum; | 4 **Example:** do<br>{<br>cout << "Input no. \n" ;<br>number = getnum( ) ;<br>}<br>while ( number > 0 ); |

**Q8. What do you mean by Recursion? Discuss the advantage(s) of recursion. Show, how recursion is carried out for calculating the value of 4! in steps.**

**Ans. Recursion :**

Recursion is a process by which a function calls itself recursively, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a previous result. It is also called 'Circular definition'. It is a control technique that is convenient for a variety of problems that would be difficult to solve using iterative constructs such as for, while and do-while loops. Recursion means self-reference and a recursive function is a function whose definition is based upon itself, i.e., a function containing either a call statement to itself or a call statement to another function that may eventually may result in a call statement back to the original function. Then that function is called a recursive function.

**Advantages of Recursion :**

1. It defines the process in a simple way to understand.
2. It is convenient for problems that would be difficult to solve using iterative constructs.
3. It reduces the size of a program using recursive functions.

**Steps of recursion process to calculate 4! :**

The computation of 4! Requires following steps :

1. 4! = 4.3!
2. 3! = 3.2!
3. 2! = 2.1!
4. 1! = 1.0!
5. 0! = 1
6. 1! = 1.1 = 1
7. 2! = 2.1 = 2
8. 3! = 3.2 = 6
9. 4! = 4.3 = 24

The steps can be described as :

1. This step defines 4! in terms of 3!, so to compute 4!, we need to compute first 3!. Thus we have to postpone the computation of 4! Until we compute 3!. This postponement is shown by next step.
2. This step defines 3! in terms of 2!, so we postpone computing 3!, until we compute 2!.
3. This step defines 2! in terms of 1!, so we postpone 2!, until we compute 1!.
4. This step defines 1! in terms of 0!, so we postpone 1!, until we compute 0!.
5. This step explicitly compute 0!, since 0 is the base value, and it equals 1.
6. Backtrack and compute 1! using the value of 0!.
7. Backtrack and compute 2! using the value of 1!.
8. Backtrack and compute 3! using the value of 2!.
9. Backtrack and compute 4! using the value of 3!.

According to the factorial program, when recursive function fact( ) is called, what happens is:

fact (4) returns (4 times fact(3) ;
which returns (3 times fact(2),
which returns (2 times fact(1),
which returns (1))))

Thus, sequence of function calls are :

```
from main ( )
fact(4)
{
 if(n == 0)
   return (1) ;
 else
   return (4 * fact(4 – 1));
}
        to main( )
fact(3)
{
 if (n == 0)
   return (1) ;
 else
   return (3 * fact (3 – 1));
}
fact(2)
{
 if (n == 0)
   return (1) ;
 else
   return (2 * fact (2 – 1));
}
fact(1)
{
 if (n == 0)
   return (1) ;
 else
   return (1 * fact (1 – 1));
}
```

```
fact(0)
{
  if (n = = 0)
    return (1) ;
}
fact(1)
{
  if (n = = 0)
    return (1) ;
  else
    return (1 * fact (1 – 1));
}
```

**Program to calculate factorial of a number using recursion :**

```
#include <stdio.h>
#include <conio.h>
int fact (int) ;
void main( )
{
  int num ;
  clrscr( ) ;
  printf("Enter any number to calculate
  factorial:");
  scanf("%d", & num) ,
  printf("\n factorial of %d=%d", num,
fact(num));
  getch( ) ;
}
int fact(int n)
{
  if(n = = 0)
      return (1) ;
  else
      return(n * fact (n – 1)) ;
}
```

**Output :**

Enter any number to calculate factorial : 4
Factorial of 4 = 24

## Q9. What is a friend function?

**Ans. Friend Functions :**

The private members of a class cannot be accessed from outside the class. That is, non - member function cannot have an access to the private data of a class. However, there could be a situation where we would like two classes to share a particular function. In such situations, C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes. Such a function need not be a member of any of these classes.

To make an outside function "friendly" to a class, we have to simply declare this function as a friend of the class as shown below:

```
Class ABC
{  ....
   ....
public
   ....
   ....
   friend void xyz (void); // declaration
};
```

The function declaration should be preceded by the keyword friend. The function is defined elsewhere in the program like a normal C++ function. The function definition does not use either the keyword friend or the scope operator ::. The functions that are declared with the keyword friend are known as friend functions. A function can be declared as a friend in any number of classes. A friend function, although not a member function, has full access rights to the private members of the class.

**A friend function possesses certain special characteristics :**

- ⇨ It is not in the scope of the class to which it has been declared as friend.
- ⇨ Since it is not in the scope of the class, it cannot be called using the object of that class. It can be invoked like a normal function without the help of any object.
- ⇨ Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name e.g., a.x.
- ⇨ It can be declared either in the public or the private part of a class without affecting its meaning.
- ⇨ Usually, it has the objects as arguments.

**Example of Friend Function :**

```
#include <iostream.h>
class sample
{  int a;
   int b;
Public :
   void setvalue ( )    {a = 25;  b = 40;}
   friend float mean (sample s);
                              //friend declared
};
```

```
float mean (sample s)
{
    return float (s.a + s.b) / 2.0;
}
main ( )
{
    sample x;                    //object x
    x.setvalue ( );
   cout << "Mean value =" <<mean (x)<< "\n";
}
```

**Merit and Demerits of Friend Function :**

**Merits :**

1. To access private data of a class from a non member function.
2. To increase the versatility of overloaded operators.

**Demerits :**

1. It makes C++ not a pure object oriented language because it provides the facility to access private data from outside the class.
2. It lacks the security of data.

### Q10. Identify a situation in which a friend function will work and a member function will not.

**Ans.** If we want to exchange private members of two classes then we can use one common friend function. But it can not be achieved with the member function.

```
Class Class_A;
Class Class_B
{
  int value1;
  public:
   void indata (int a) {value1 = a;}
   void display (void) {cout << value1 << "\n";
   friend void exchange (class_1 &, class_2&);
};
class class_2
{  int value2;
  public:
    void indata (int a) {value2 = a;}
    void display (void){cout << value2 << "\n"; }
     friend void exchange (class_1 &, class_2&);
};
void exchange (class_1 & x, class_2 & y)
{       int temp = x.value1;
        x.value1 = y.value2;
        y.value2 = temp;
}
int main ( )
{
   class_1 C1;
   class_2 C2;
   C1.indata (100);
   C2.indata (200);
   cout << "Values before exchange"<< "\n"
   C1.display( );
   C2.display( );
   exchange(C1, C2);
   cout << "Values after exchange"<< "\n";
   C1.display( );
   C2.display( );
   return 0;
}
```

### Q11. What is the difference between a friend function and a member function?

**Ans.** Following are the differences between a friend function and a member function :

1. A friend function is not a member of a class. But still friend function can access private members of the class to which it has been declared as friend. But a member function should be declared in the class, then only member function can access the public, private & protected members of that particular class.
2. Because friend function is not in the scope of the class, it cannot be called using the object of the class. But a member function can be called only by the object of that particular class.

### Q12. What is an inline function? How is it advantageous over macros?

**Ans. Inline Function :** An inline function is a function that is expanded in line when it is invoked i.e., compiler replaces the function call with the corresponding function code (something similar to macro expansion).Inline functions are made by using the keyword inline in front of the function name. It merely sends a request and not a command to the compiler. The compiler may ignore this request if the function definition is too long or too complicated and compile the function as a normal function. The keyword inline gives a hint to the compiler that inline substitution of the function body is to be preferred to the usual function call implementation.

2. For functions not returning values, if a return statement exists.
3. If functions contains static variables.
4. If Inline functions are recursive.

**Difference between inline function and macros:**

| Inline Function | Macros |
|---|---|
| 1. It is a function provided by C++. | It is a preprocessor directive provided by C. |
| 2. It is declared by using the keyword inline before the function prototype. | It is declared by using the preprocessor directive # define before the actual statement. |
| 3. Expressions passed as arguments to inline functions are evaluated once. | In some cases, expressions passed as arguments to macros can be evaluated more than once. |
| 4. They are passed by the compiler. | They are expanded by the preprocessor at precompile time. |
| 5. Inline member functions can access the class's member data. | The preprocessor has no permission to access member data of a class & are thus useless even if used within a class. Thus they cannot be used as member function. |
| 6. Inline functions may or may not be expanded by the compiler. It is the compiler's decision whether to expand the function inline or not. | Macros are always expanded. |
| 7. It can be defined inside or outside the class. | It can not be defined inside the class. |

## Q14. What is structure? How structure is declared and how structure variable is declared?

**Ans.** Structure is a collection of heterogeneous type of data i.e. different types of data. The various individual data components in a structure can be accessed and processed separately. A structure is a collection of variables referenced under one name, providing a convenient means of keeping related information together. A structure declaration forms a template that may be used to create structure objects (that is, instances of a structure).

A structure definition contains a keyword **struct** along with structure name which is followed by the curly braces, which contains different components/ elements/ member of the structure, of various data types. The general format of a simple structure declaration is:

```
<storage_class struct user_defined_name>
{
        datatype    member1;
        datatype    member2;
        .....
        .....
};
```

In above declaration,

**Storage_class** refers to the scope of structure variable. The storage class is optional **struct** is a keyword which shows the start of a structure.

**user_defined_name** is the structure name defined by the user.

The general format of declaring structure variables inside the structure is:

```
<storage_class>struct<user_defined_name>
{
        datatype member_1;
        datatype member_n;
}       variable_1, variable_2, variable_n;
```