Introduction to C++

- ✓ C++ is an object-Oriented programming language. Initially named 'C with Classes', C++ was developed by **Bjarne Stroustrup** at AT&T Bell Laboratories in New Jersey, USA, on the early eighties.
- ✓ C++ is an extension of C with a major addition of the class construct feature of SIMULA67.
- ✓ C++ is a superset of C. Most of what we already know about C applies to C++ also. Therefore, almost all C programs are also C++ programs with slight modifications.
- ✓ The most important facilities that C++ adds on to C are classes, objects, inheritance, function overloading, and operator overloading. These features enable us to create abstract data types, inherit properties from existing data types and support polymorphism, thus making C++ a truly object-Oriented Language.
- ✓ The Object-Oriented features in C++ allow programmers to build large programs with clarity, extensibility and ease of maintenance, incorporating the spirit and efficiency of C.

Features Object-Oriented Programming(OOPs)

Explain the features of OOPs.

The following are the some of the features to know extensively about the Object-Oriented Programming.

1) Objects
2) Classes
3) Encapsulation
4) Inheritance
5) Multiple inheritance
6) Polymorphism
7) Dynamic hiding
8) Message Passing
9) Data Abstraction
10) Extensibility
11) Persistence

12) Genericity

13) Data binding

1) **Object:**
   - ✓ Objects are the basic run-time entities in an Object-Oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program must handle.
   - ✓ Object is a collection of number of 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 contains data and code to manipulate the data.
   - ✓ Objects can interact without having known details of each other's data or code.
   - ✓ An object is considered to be a partitioned area of computer memory that stores data and set of operations that can access that data.
2) **Class :**
   - ✓ It is an abstract data type that contains data members and member functions that operates on data. It starts with the keyword class.
   - ✓ The entire set of data and code of an object can be made a user-defined data type with the help of a **CLASS**.
   - ✓ Objects are variables to type class. Once a class has been defined, we can create any number of objects belonging to that class.
3) **Encapsulation:**

   The wrapping of data and functions into a single unit is known as Encapsulation. The data is not accessible to the outside world and only those functions which are wrapped in the class can access it. These functions provides an interface between object's data and the program. This insulation of the data from direct access by the program is called data hiding.
4) **Inheritance:**
   - ✓ Inheritance is the process by which objects of one class acquire the properties of objects of another class.

- ✓ That is, deriving a new class from existing class. Here new class is called derived class where as existing class is called base class.
- ✓ 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.

5) Multiple Inheritance:

The mechanism by which a class is derived from more than one base class is known as multiple inheritance.

6) Polymorphism:

- ✓ Polymorphism is another important OOP concept.
- ✓ Polymorphism means the ability to take more than one form.
- ✓ For example, an operation may exhibit different behavior ion different instances. The behavior depends upon the types of data used in the operation.
- ✓ Function overloading and operator overloading are the examples of polymorphism.

7) Data hiding:

We can hide the information of class like we can hide data and member functions of a class by specifying private and protected members of class. Private and protected members are not accessed by outside the class, thus we can hide the data of a class.

8) Message Passing:

- ✓ An Object-Oriented program consists of a objects that communicate with each other.
- ✓ Objects communicates with one another by sending and receiving information much the same way as people pass messages to one another.
- ✓ A message for an object is a request for execution of a procedure, and therefore will invoke a function in the receiving object that generates the desired result.

- ✓ Message passing involves specifying the name of the object, the name of the function and the information to be sent.
- ✓ Objects have a life cycle. They can be created and destroyed.

9) Data Abstraction:

Abstraction refers to the act of representing essential features without including the background details or explanation. Since the classes use the concept of data abstraction, they are known as Abstract Data Types (ADTs).

10) **Extensibility:** It is a feature which allows the extension of the functionality of the existing software components. In C++, this is achieved through inheritance.

11) **Persistence:** The mechanism where the object outlives the program execution time and exists between executions of a program is known as persistence.

12) **Genericity:** It is a technique for defining software components that have more than one interpretation depending on the data type. In C++, genericity is achieved by function templates and class templates.

13) **Dynamic Binding:** Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with it given procedure call is not known until the time of the call at run-time. It is associated with Polymorphism and Inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Adding Comments in a program

- ✓ C++ introduces a new comment symbol "//"(double slash). Comments start with a double slash symbol and terminate at the end of the line. A comment may start anywhere in the line. The double slash comment is basically a single line comment.

  Ex: //this is an example

- ✓ The C comment symbols "/* , */are still valid and are more suitable for multiline comments.

Ex: /*this is an example of

C++ program */.

<u>Output Operator:</u>The standard output statement is "**Cout<<"Example for output statement"**. In the above statement the identifier COUT is a predefined object that represents the standard output stream in C++. Here, the standard output stream represents the screen. The operator "<<" is called the "insertion or put to operator". It inserts the contents of the variable on its right to the object on its left.

<u>Input Operator:</u> The standard Input statement is "**Cin>>num"**. The above input statement causes the program to wait for the user to type in a number. The number keyed in is placed in the variable num. The identifier CIN is a predefined object in C++ that corresponds to the standard input stream. Here this stream represents the keyboard. The operator ">>" is known as extraction or get from operator. It extracts the value from the keyboard assigns it to the variable on its right.

<u>Structure of C++ program</u>

<u>Explain the structure or layout of C++ Program</u>

- ✓ A typical C++ program would contain four sections as shown below. They are:
    1) Include files section
    2) Class declaration section
    3) Member function definition section
    4) Main() program section

INCLUDE FILES
CLASS DECLARATION
MEMBER FUNCTIONS DEFINITIONS
MAIN FUNCTION PROGRAM

Include files: In this section we can include all header files which supports C++ Program. For example #include<iostream.h>,#include<math.h>, #include<string.h> are the header files.

Class Declaration: In this section we will declare the class which holds data and member functions.

Member function definition: In this section we will define all the member functions which are declared in the class. Using Class name followed by :: (scope resolution operator) followed by function name we will define the member function.

Main() program:

- ✓ Each and every C++ Program execution starts from main().
- ✓ Each and every member function is called from main ()
- ✓ We will create object of the class in main ().

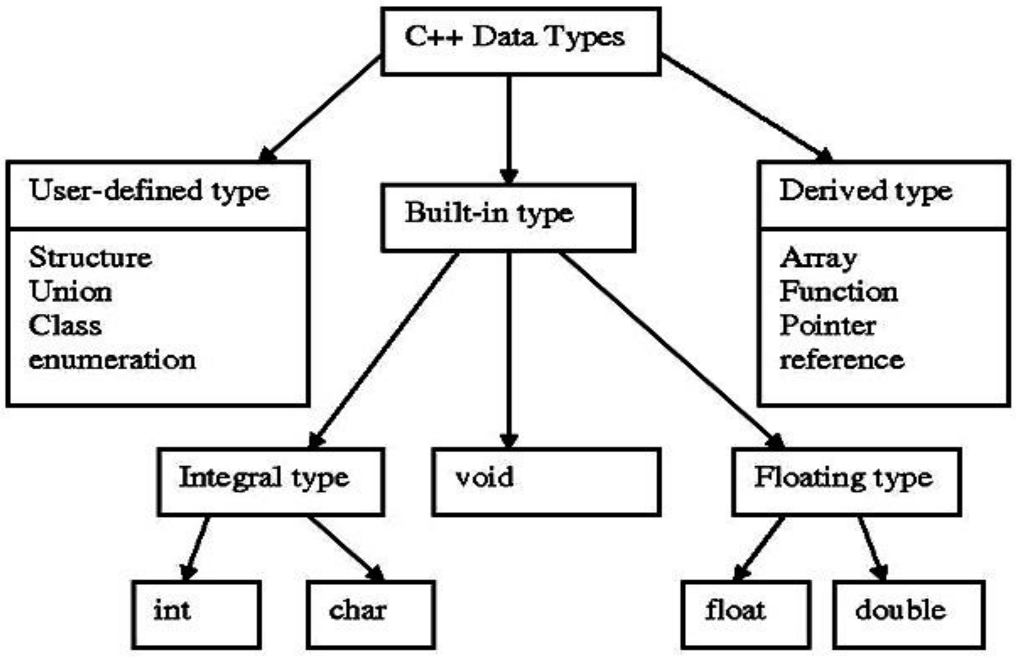
Sample Program:

```
#include<iostream.h>

void main()

{

cout<<"welcome"<<endl;

cout<<"C++"<<endl;

}
```

Data Types

Explain about data types with their sizes and range.

- ✓ The type of information stored in a variable is called data type.
- ✓ A particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.
- ✓ There are 3 different types of data types:
  1) User defined data type
  2) Built- in type data type
  3) Derived type



<u>**C++ DATA TYPE**</u>

<u>Primitive or Built-in Types:</u>

C++ offer the programmer a rich assortment of built-in data types. Following table lists down seven basic C++ data types:

Type	Size	Range
char	1byte	-127 to 127
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	2bytes	-32768 to 32767
unsigned int	2bytes	0 to 65,535
signed int	2bytes	-32768 to 32767

short int	2bytes	-32768 to 32767
unsigned short int	2 bytes	0 to 65,535
signed short int	2 bytes	-32768 to 32767
long int	4bytes	-2,147,483,647 to 2,147,483,647
signed long int	4bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295
float	4bytes	3.4e - 38 to 3.4e + 38
double	8bytes	1.7e- 308 to 1.7e-+308
long double	10bytes	304 e -4932 to 1.1 e+4932

Following is the example, which will produce correct size of various data types on your computer.

```
#include <iostream.h>

void main()

{
  cout << "Size of char : " << sizeof(char) << endl;
  cout << "Size of int : " << sizeof(int) << endl;
  cout << "Size of short int : " << sizeof(short int) << endl;
  cout << "Size of long int : " << sizeof(long int) << endl;
  cout << "Size of float : " << sizeof(float) << endl;
  cout << "Size of double : " << sizeof(double) << endl;
  cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;
}
```

CLASSES

Define a class. Explain about class specification.

Definition:

- ✓ A class is a collection of data members and member functions.
- ✓ A class is a way to bind the data and its associated functions together.
- ✓ C++ allows the data and functions into a single unit is called a class..
- ✓ When defining a class, we are creating a new ABSTRACT DATA TYPE that can be treated like any other built-in data type.

Class Specification:

- ✓ Generally a class specification has two parts:
  1) Class declaration
  2) Class function definition.

1) Class declaration: (How to declare or create a class?)

Class is a collection of data members and member functions.

Declaration of a class:

Syntax:

```
class  classname
{
Private:
Data members;
Member functions;
Public:
Data members;
Member functions;
};
```

Explanation:

- ✓ The keyword **class** is used to declare a class followed by class name.
- ✓ The body of a class is enclosed within braces and terminated by a semicolon.
- ✓ The class body contains the declaration of variables (data members) and functions (member functions).

- ✓ These functions and variables are collectively called MEMEBRS.
- ✓ The variables declared inside the class are known as data members and the functions are known as member functions.
- ✓ They are usually grouped under two sections, PRIVATE and PUBLIC to denote which of the members are private and which of the members are public.
- ✓ The keywords PRIVATE and PUBLIC are known as Access specifiers.
- ✓ The members that have been declared as PRIVATE can be accessed only from within the class. i.e, PUBLIC members can be accessed from outside the class also.
- ✓ The use of PRIVATE is optional. By default, the members of a class are PRIVATE.

Example:

```
class  Employee
{
Private:
            int eid;
            float salary;
 Public:
            void  getdata(int a, float b);
            void putdata();
};
```

Creating Objects or object creation

How to create an object?

- ✓ Objects are instances of classes.
- ✓ Once a class has been declared, we can create variables of that type by using the class name.
- ✓ We have two methods to create objects to a class

Method 1:

- ✓ We will create the objects in main() as follows:
- ✓ **Syntax:**

  classname   objectname;
- ✓ **Example:**

Employee e1;

- ✓ In the above example e1 is an object of a class Employee.
- ✓ We can create multiple objects at a time to a class;
- ✓ **Syntax:**

  classname objectname1, objectname2,………. Objectnamen;
- ✓ **Example:**

  Employee e1,e2,e3;

<u>Method2:</u>

- ✓ We can create the objects at time of class declaration itself
- ✓ That is, Objects can also be created when a class is declared by placing their names immediately after the closing brace as follows.
- ✓ **Syntax:**

```
class classname
{
Datamembers;
Member functions;
} objectnam1,objectname2,……objectnamen;
```
- ✓ **Example:**

```
Class Employee
{
int eid;
float salary;
} e1,e2,e3;
```
- ✓ In the above example we created 3 objects e1,e2,e3 to a class Employee at the time declaration of class.

<u>Defining Member functions (How to define member function?)</u>

- ✓ Class contains data members and member functions.
- ✓ We will declare member functions inside the class.
- ✓ Whereas we will define member functions either inside the class or outside the class

- ✓ That is, member functions can be defined in two places:
  1) Outside the class
  2) Inside the class

1) <u>Outside the Class:</u>

- ✓ Member functions that are declared inside a class have to be defined separately outside the class.
- ✓ To define the function outside the class we will use class name followed by :: (scope resolution operator) followed by function name,
- ✓ **<u>Syntax:</u>**

```
Return-type  class-name :: function-name (Parameters List)
{
        //Function body
}
```

The membership label **classname ::** tell the compiler that the function function-name belongs to the class classname. That means, The specified function belongs to specified class.

- ✓ **<u>Example:</u>**

```
void Employee  ::  getdata(int a, float b)
{
    eid = a;
    salary = b;
}
```

- ✓ Here in the above example getdata() function belongs to class Employee.

<u>Inside the class:</u>

- ✓ Another method of defining a member function is to replace the function declaration by the actual function definition inside the class.
- ✓ **<u>Example:</u>**

```
class  Employee
{
    int  eid;
    float  salary;
```

```
Public:
void  getdata(int a, float b)
{
eid = a;
salary = b;
}
void putdata()
{
cout<<eid<<endl;
cout<<salary<<endl;
};
```

- ✓ In the above example both functions getdata() and putdata() both are defined inside the class only.
- ✓ **NOTE:** When the function is defined inside a class , it is treated as an inline function.

<u>Accessing member functions (How to access member functions)</u>

- ✓ We can access member functions of a class using objects followed by '.' Operator (member accessing operator) followed by function name.
- ✓ In the main() we will call access member functions.
- ✓ **Syntax:**

```
object name  .  function-name (arguments);
```

- ✓ **Example:**

```
e1.getdata(10,20000.00);
```

- ✓ In the above example we are calling getdata() function using an object e1 with two arguments 10 and 20000.00.

Constructors

What is mean by constructor? Explain types of constructors.

- ✓ C++ provides a special member function called the CONSTRUCTOR which enables an object to initialize itself when it is created. This is known as automatic initialization of objects.
- ✓ A constructor is a 'special' member function whose task is to initialize the objects of its class.
- ✓ It is special because constructor name is the same as the class name.
- ✓ The constructor is invoked whenever an object of its associated class is created.
- ✓ It is called constructor because it construct the values of data members of the class.
- ✓ There are 3 different types constructors
    1) Default constructor
    2) Parameterized constructor
    3) Copy constructor

1) Default constructor:

- ✓ A constructor which does not take arguments or parameters is called default constructor.
- ✓ A constructor that accepts no parameters is called DEFAULT CONSTRUCTOR.
- ✓ Default constructor name and class name must be same.
- ✓ **Syntax to create default constructor:**

```
classname()
{
//constructor definition or constructor body
}
```

Example:

```
item()
{
m=0;
n=0;
}
```

- ✓ In the above example, constructor declaration and definition done at a time.
- ✓ We can declare and define a constructor separately like functions.
- ✓ Declaration we will do inside the class and constructor definition outside the class as follows.
- ✓ **Example:**

```
class item
{
int m,n;
public:
 item();   //constructor declaration
};
 item :: item ()           // constructor definition
{
m=0;
 n=0;
}
```

In the example the declaration not only creates the object of type class **item** but also initialized its data member's **m and n to zero**. There is no need to write any statement to invoke(call) the constructor function. If a normal member function is defined for zero initialization, we would need to invoke this function for each of the objects separately.

Program (Default Constructor):

```
#include<iostream.h>
class cube
{
public:
int side;
cube()                        //Default Constructor
{
side=10;
}
};
void main()
{
cube c1;
cout<<c1.side;
}
```

Parameterized Constructor:

- ✓ The constructors that can take arguments are called PARAMETERIZED CONSTRUCTORS.
- ✓ Constructor name and class name must be same.
- ✓ We must pass the initial values as arguments to the parameterized constructor function when an object is created.
- ✓ **Syntax to create parameterized constructor:**

```
classname(parameters list)
{
//constructor definition or constructor body
}
```

Example:

```
item(int a, int b)
{
m=a;
n=b;
}
```

- ✓ In the above example, constructor declaration and definition done at a time.
- ✓ We can declare and define a constructor separately like functions.
- ✓ Declaration we will do inside the class and constructor definition outside the class as follows.
- ✓ **Example:**

```
class item
{
```

```
int m,n;
 public:
  item(int x,int y);   //constructor declaration
};
 item :: item (int a,int b)   // constructor definition
{
m=a;
 n=b;
}
```

- ✓ In main (), we must we must pass the initial values as arguments to the parameterized constructor function when an object is created as follows:

  Item i1 = item (0,100) ; //explicit call

  (OR)

  Item i1(0,100); //implicit call

- ✓ When the constructor is parameterized, we must provide appropriate arguments for the constructor.

<u>Program (Parameterized Constructor):</u>

```
#include<iostream.h>
class cube
{
public:
int side;
cube(int x)             //Parameterized Constructor
{
side=x;
}
};
void main()
{
cube c1(10);
cout<<c1.side;
cube c2(20);
cout<<c2.side;
}
```

Copy Constructor:

- ✓ The constructors that can take arguments are called COPY CONSTRUCTOR; however, a constructor can accept a REFERENCE to its own class as a parameter.
- ✓ That means, we have to specify copy constructor argument is object of a class.
- ✓ Constructor name and class name must be same.
- ✓ **Syntax:**

```
classname(classname   &objectname)
{
//constructor definition or constructor body
}
```

- ✓ **Example:**

```
Sample(Sample &P)
{
a=p.a;
b=p.b;
c=p.c;
}
```

- ✓ In the above example, constructor declaration and definition done at a time.
- ✓ We can declare and define a constructor separately like functions.
- ✓ Declaration we will do inside the class and constructor definition outside the class as follows.
- ✓ **Example:**

```
        class Sample
        {
        int m,n;
```

```
public:
Sample(Sample &P)//constructor declaration

};
Sample:: Sample (int a,int b)   // constructor definition
{
a=p.a;
b=p.b;
c=p.c;
}
```

- ✓ A copy constructor is used to declare and initialize an object from another object.
- ✓ For example, the statement **Sample S2(S1);** would define the object **S2** and at the same time initialize it to the values of **S1**.
- ✓ We can write the above example as **Sample S2 = S1;**
- ✓ That means,

  Sample S2(S1);

  (OR)

  Sample S2 = S1;

  Both above statements are same

  <u>Program: (copy constructor)</u>

```
#include<iostream.h>
class Sample
{
public:
int a,b,c;
sample(int x,int y,int z)     //parameterized constructor
{
a=x;
b=y;
c=z;
}
Sample(Sample &p)  //Copy Constructor
{
a=p.a;
b=p.b;
c=p.c;
}
void display()
{
cout<<a;
cout<<b;
cout<<c;
}
};
void main()
{
Sample s1(10,20,30);
Sample s2(s1);
cout<<"object 1 details are:";
s1.display();
cout<<"object 2 details are:";
s2.display();
}
```

NOTE:

The constructor functions have some special characteristics:

- ✓ They should be declared in the public section
- ✓ They are invoked automatically when the objects are created.
- ✓ They do not have return types, not even void and therefore, they cannot return values
- ✓ They cannot be inherited, though a derived class can call the base class constructor.

Destructors(Explain about destructor)

- ✓ A destructor , is used to destroy the objects that have been created by a constructor.
- ✓ Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde (~)
- ✓ **Syntax:**

  ~classname();
- ✓ **Example:**

  ~sample ();
- ✓ A destructor never takes any argument nor does it return any value.
- ✓ It will be invoked implicitly by the compiler upon exit from the program.
- ✓ **Program:**

```
#include<iostream.h>
class test
{
public:
test();        //default constructor
~test();     //destructor
};
test :: test()
{
cout<<"object is created"<<endl;
}
test :: ~test()
{
cout<<"object is destroyed"<<endl;
}
void main()
{
test t;
cout<<"we are in main()"<<endl;
```

```
    cout<<""welcome";
    }
```

Output:

```
        object is created
        we are in main()
        welcome
        object is destroyed
```

FUNCTIONS:

1) What is mean by function and its advantages.

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main().**

Definition:

Function is a self contained block of statements that performs a particular task.

(OR)

Functions are the entities which are grouping a set of statements which do a specific job or set of jobs.

Advantages of functions:

1. The length of a source program can be reduced by the use of functions at appropriate places.
2. These are used to increase the execution speed.
3. A function may be used by many other programs.
4. We can easily identify the errors (OR) it is easy to locate and isolate a faulty function for further corrections.)
5. It facilitates top-down modular programming.

Types of functions:

2) Explain about user defined functions.

There are two different types of function

1. User defined functions
2. Built in functions

1) User- defined functions:

Definition: Function is a self contained block of statements that performs a particular task.

Components or elements of User defined functions: There are 3 components

1. Function declaration
2. Function definition
3. Function calling

1) Function declaration:

- ✓ Like variables, all functions in a C++ program must be declares, before they are invoked.
- ✓ We will declare a function within the class declaration.
- ✓ A function declaration also known as function prototype which consists of 4 parts.
  - **a)** Function type(return type)
  - **b)** Function name
  - **c)** Parameters list
  - **d)** Terminating semicolon

- ✓ **Syntax:**
  return-type function-name(parameterlist);
- ✓ **Example:**
  1. void display();
  2. int maximum(int a,int b);
  3. int minimum();
  4. void add(int a,int b);

Return (data) type: Each function must return on value after executing a program. So return type must be data type (like int, float, char….). If you specify return type as void this means the function does return any value.
Function name: We can specify a function name which is used to specify the name of the function.
Parameter list: The Parameter list or argument list contains valid variable names separated by commas. The list must be surrounded by parentheses. Note that there is semicolon follows the closing parenthesis.

<u>2) Function definition:</u>

- ✓ A function definition also known as function implementation
- ✓ In this we will specify actual task performed by function.
- ✓ We can define a function inside a class declaration or outside class declaration.
- ✓ It includes the following elements.
  a) Function name
  b) Return(data) type
  c) List of parameters
  d) Function body statements
- ✓ **<u>Syntax:</u>**

```
  return type    function name(argument list)
 {
   --------
   --------} body of the function
   --------
 }
```

- ✓ **Example:**

```
void add (int a,int b)
{
 int c=a+b;
 cout<<"the sum is:"<<c;
}
```

Return type or data type:
Each function must return on value after executing a program. So return type must be data type(like int, float, char….). If you specify return type as void this means the function does return any value.
Function name:A function must follow the same naming rules as variable names in C. Additional care must be taken to avoid duplicating library routine names or operating system commands.
Argument list:
The argument list contains valid variable names separated by commas. The list must be surrounded by parentheses. Note that no semicolon follows the closing parenthesis.
Function body:
The function body contains the declaration and statements for performing the required task and contains return statement also.

3) Function call:

- ✓ The last element is the function call, which when being needed a simple call will make the function to work.
- ✓ A function can be called by simply using object name followed by function name followed by a list of actual parameters (or arguments). Without object we cannot call function.

Syntax:
Object name.function-name (parameter-list);
Example:
s.add(x,y);
Once the function is called, the control goes to the actual implementation and executes the statements inside the function and the value is returned to the main function.

Program:
/* C++ program on functions */
```
#include<iostream.h>
class employee
{
public:
int  eid;
float  salary;
void  store(int a,float b)
{
eid=a;                    //function declaration & function definition of
salary=b;                   store() function
}
void display();          //This line specifies function declaration
};
void employee :: display()       // function definition of display()
{
cout<<"employee id is:"<<eid<<endl;
cout<<"employee salary is:"<<salary<<endl;
}
void main()
{
employee e1;                //object creation
e1.store(456,10000.00);//function call
e1.display();                        //function call
}
```

<u>Output:</u>

```
   employee id is: 456
   employee salary is: 10000.00
```

<u>PARAMETER PASSING</u>

<u>3) Explain about PARAMETER PASSING or explain various mechanisms to pass parameters to a function.</u>

- Parameter passing is a mechanism for communication of data and information between the calling (caller) and called function (callee)
- Parameters are two types:
  1) Actual parameters
  2) Formal parameters
- The formal parameters are those specified in the function definition and function declaration.
- The actual parameters are those specified in the function call.
- There are 3 techniques to pass parameter to functions:
  1) Pass by value
  2) Pass by address
  3) Pass by reference

➢ **<u>PASS BY VALUE</u>**

By default, functions pass arguments by value, which means that when the function is used, the contents of the actual parameters are copied into the formal parameters.

Pass by value mechanism does not change the contents of the argument in the calling function, if they are changed in the called function.

Program: (pass by value)

```
#include<iostream.h>
    void swap(int x,int y)
    {
    int t;
    cout<<"before swapping x and y="<<x<<endl<<y;
    t=x;
    x=y;
    y=t;
    cout<<"after swapping x and y="<<x<<endl<<y;
    }
    void main()
    {
    int a,b;
    cout<<"enter a and b values are "<<endl;
    cin>>a>>b;
    swap(a,b);
    }
```

output:

```
    enter a & b values
    10
    20
    before swapping  x & y+10,20
    after swapping  x & y 20,10
```

- **PASS BY ADDRESS:**

  C++ provides another means of passing values to a function known as pass-by address. Instead of passing value, the address of the variable is passed. In the function, the address of the argument is copied into a memory location instead of the value. The de-referencing operator (*) is used to access the variable in the called function.

```
    #include<iostream.h>
    void swap(int *x,int *y)
    {
    int t;
    cout<<"before swapping x and y="<<*x<<endl<<*y;
    t=*x;
    *x=*y;
    *y=t;
    cout<<"after swapping x and y="<<*x<<endl<<*y;
    }
    void main()
```

```
{
int a,b;
cout<<"enter a and b values are "<<endl;
cin>>a>>b;
swap(&a,&b);
}
```

- **PASS BY REFERENCE:**
  - ✓ In traditional C, a function call passes arguments by value. The called function creates a new set of variables and copies the values of arguments into them.
  - ✓ The function does not have access to the actual variables in the calling program and can only work on the copies of values. But, there may be situations where we would like to change the value of variables in the calling program.
  - ✓ Provision of the reference variables in C++ permits us to pass parameters to the function by reference.
  - ✓ When we pass arguments by reference, the formal arguments in the called function become aliases to the actual arguments in the calling functions.
  - ✓ This means that when the function is working with its own arguments, it is actually working on the original data. Consider the following function:

```
void swap (int & a, int & b)
  {
           int t = a ; // dynamic initialization
              a = b;
              b = c;
  }
```

Now, if m and n are two integer variables, then the function call

```
swap (m,n)
```

Will exchange the values of m and n using their aliases (reference variables) a and b.

Program:

```
#include<iostream.h>
void swap(int &x,int &y)
{
int t;
cout<<"before swapping x and y="<<x<<endl<<y;
t=x;
x=y;
y=t;
cout<<"after swapping x and y="<<x<<endl<<y;
}
void main()
{
int a,b;
cout<<"enter a and b values are "<<endl;
cin>>a>>b;
swap(a,b);
}
```

FRIEND FUNCTION:

4) Explain about friend functions

- ✓ In general we cannot access private and protected members of a class.
- ✓ But friend function is used to access private and protected members of a class.
- ✓ We will use friend key word to declare friend function.
- ✓ To work with friend functions we will follow 3 steps. They are:
  1) Friend function Declaration: we will declare friend function inside the class using friend keyword
  2) Friend function Definition: we will define friend function inside or outside a class
  3) Friend function Calling: we will call a friend function in main()

➢ **SYNTAX DECLARATION:**

 friend returntype function name(class name object name);

Example:

 friend void friendfunction1(sample s);

➢ **SYNTAX DEFINITION:**

 return type function name(classname objectname);

Example:

 void friendfunction1(sample s);

➢ **SYNTAX CALLING:**

 function name(classname objectname);

example:

 friendfunction1(s);

PROGRAM: (on friend functions)

```
class sample
{
private:
    int a;
    int b;
public:
    sample()
    {
    a=10;
    b=20;
    }
    friend void function1(sample s); // declaration
};
void function1(sample s)   //defintion
{
cout<<"the private data members are:"<<endl;
cout<<s.a<<endl;
cout<<s.b<<endl;
}
void main()
{
sample s;
function1(s); //calling
}
```

OPERATOR OVERLOADING:

5) What is mean by OPERATOR OVERLOADING? Explain various types

- ✓ Operator overloading is a concept through which the meaning of the operator is pre defined.
- ✓ It is a type of polymorphism in which an operator is overloaded to give a special meaning to it.
- ✓ C++ allows you to specify more than one definition for an operator , which is called operator overloading.
- ✓ **Operator overloading** allows the programmer to define how operators (such as +, -, ==, <,>,<=,>=and !) should interact with various data types. Because operators in C++ are implemented as functions.
- ✓ There are two different types of operator overloadings
  1. Unary operators overloading
  2. Binary operator overloading

6) Explain about Unary operating overloading:

- ✓ We can overload unary operators like unary +,unary -,++(increment),--(decrement)
- ✓ When an operator is overloaded the following steps will be involved.
  1) Create(or)declare member function to overload unary operators.
  2) Define a member function to overload unary operators.
  3) Calling a member function to overload unary operators.

➢ **FUNCTION DECLARATION:**

- **Syntax:**
  return type operator operatorsymbol();

- **Example:**
  Void operator-();

➢ **FUNCTION DEFINITION:**

Syntax:

Inside the class declaration:

```
return type   operator   operatorsymbol()
{
Body of the function
}
```

Example:

```
Void operator-()
{
------------
-----
}
```

OR

Syntax:

Outside the class declaration:

```
return type  class name :: operator operatorsymbol()
{
Body of the function
}
```

Example:

```
Void  sample :: operator-()
{
------------
-----
}
```

➢ **FUNCTION CALL:**

Syntax:

```
operatorsymbol  objectname;
```

Example:

```
-s;
```

Here s is an object name

➢ **Program: (unary operator overloading)**

Let us consider the unary minus operaor. A minus operator, when used as a unary, takes just one operand.

```
#include<iostream.h>
class space
(
private:
int x;
int y;
int z;
public:
void get data(int a,int b,int c);
void display();
void operator-(); // declaration
};
void sample :: get data(int a,int b,int c)
{
x=a;
y=b;
z=c;
}
void space :: display()
{
cout <<"x="<<x<<endl;
cout<<"y="<<y<<endl;
cout<<"z="<<z<<endl;
}
void space :: operator -()                    //defintion
{
x=-x;
y=-y;
z=-z;
}
void main()
{
space s;
cout<<"s="<<endl;
s.get data(10,-20,30);
s.display();
-s;                              //calling
cout<<"s="<<endl;
s.display();
}
```

7) Explain about BINARY OPERATOR OVERLOADING:

- ✓ We can overload binary operators like +,-,*,/,<,>,<=,>=
- ✓ There are three steps to overload binary operators.
  1. function declaration to overload binary operator
  2. function definition to overload binary operator
  3. Function call to overload binary operators.

➢ **FUNCTION DECLARATION:**

- **Syntax:**
  return type operator operator symbol(class name object name);
- **Example:**
  Void operator-(sample s);

➢ **FUNCTION DEFINITION:**

Inside the class:

Syntax:

```
return type   operator   operator symbol(class name  object name)
{
Body of the function
}
```

Example:

```
Void operator-(sample s)
{
------------
-----
}
(OR)
```

Outside the class:

Syntax:

```
return type  classname:: operator operatorsymbol
                                  (classname objectname )
{
Body of the function
}
```

Example:

```
Void sample :: operator-(sample s)
{
```

```
------------
-----
}
```

- **FUNCTION CALL:**

  Syntax:

  Object1 operatorsymbol object2;

  Example:

  S1 + S2

  Here S1 and S2 are objects.

Program:

Write a c++ program on binary operator overloading for complex numbers

```
#include<iostream.h>
class complex
{
float x;
float y;
public:
complex(float real,float imag)
{
x=real;
y=imag;
}
void display();
complex operator+(complex c);
};
complex  complex :: operator+(complex c)
{
complex temp;
temp.x=x+c.x;
temp.y=y+c.y;
return temp;
}
void complex :: display()
{
cout<<x<<"+j"<<y<<endl;
}
void main()
{
complex  c1,c2,c3;
c1=complex(2.5,3.2);
c2=complex(3.5,4.8);
c3=c1+c2;
cout<<"c1=";
c1.display()
cout<<"\n c2=";
c2.display();
cout<<"\n c3=";
c3.display();
}
```

BUILT IN FUNCTION OR LIBRARY FUNCTIONS:

- ✓ Library functions are pre-defined functions.
- ✓ Library functions are part of header file.
- ✓ In Library function, name of function cannot be changed

8)List various MATH LIBRARY FUNCTION:

- ✓ These functions are used to perform mathematical calculations in a program.
- ✓ There are various functions as follows:

Function name	Purpose	Example
sqrt(x)	square root of 'x'	sqrt(4)=2
sin(x)	trigonometric sine of 'x'	sin(30)=0.5
cos(x)	trigonometric cosine of 'x'	cos(60)=0.5
tan(x)	trigonometric tangent of 'x'	tan(45)=1
exp(x)	exponential function of 'x'	exp(2)=e^2
pow(x,y)	x raised to power y	pow(2,3)=2^3=8
log(x)	logarithmic function of x(natural i.e; base 'e')	log(2)=0.3010
log10(x)	logarithmic function of x(base 10)	log10(10)=1
ceil(x)	rounds x to the smallest integer not less than x	ceil(8.1)=9.0 ceil(-8.8)=-8
floor(x)	rounds x to the largest integer not greater than x	floor(8.2)=8.0 floor(-8.8)=-9.0

abs(x)	absolute value of x,if (i)if,x>0,then abs(x)=x (ii)if,x=0,then abs(x)=0 (iii)if,x<0,then abs(x)=-x	abs(10)=10 abs(0)=0 abs(-10)=-(-10)=10

PROGRAM:

```
#include<iostream.h>
Void main()
{
Cout<<"the built in functions are"<<endl;
Cout<<sqrt(4)<<endl;
Cout<<sin(30)<<endl;
Cout<<cos(60)<<endl;
Cout<<tan(45)<<endl;
Cout<<exp(2)<<endl;
Cout<<pow(2,3)<<endl;
Cout<<log(2)<<endl;
Cout<<log(10)<<endl;
Cout<<ceil(8.1)<<endl;
Cout<<floor(8.2)<<endl;
Cout<<abs(10)<<endl;
}
```

INHERITANCE

1) What is inheritance?

Introduction:

- Inheritance is a mechanism of deriving a new class from existing class. Here existing class is called base class and new class is called derived class or existing class is also called parent class or super class. New class is also called sub-class or child class.

  (OR)
- Inheritance is the capability of one class to acquire properties and characteristics from another class.
- The class whose properties are inherited by other class is called the parent or base or super class. And the class which inherits properties of other class is called child or derived or sub class
- **Advantage:** C++ strongly supports inheritance to achieve reusability. It is always nice when we reuse something that already exists rather than trying to create same thing again.
- **DERIVING OR DEFINING A DERIVED CLASS:-**

  Syntax:-
  ```
  class  <derived class name> : <access specifier>  <base class name>
  {
  //  Body of the derived class
  }
  ```
  Explanation:
  - ✓ The derivation of derived class from the base class indicated by the colon (:). (OR): (colon) indicates that derived class is derived from base class.
  - ✓ If the visibility mode is specified, it must be either PUBLIC or PRIVATE.
  - ✓ The visibility mode is optional. The default visibility mode is PRIVATE.
  - ✓ When a base class is privately inherited by a derived class, public members of the base class become private members of the publicly inherited, public members of the base class become public members of the derived class.
  - ✓ In both the cases the private members are inherited from base class to derived class.

  Example:-
  ```
  Class D: public B
  {
   //Data members and member functions
  }
  ```

- In above Example, D is derived class and B is base class and access specifier is public. So, all public members of the base class **A** become public members of the derived class **B**.

Types of inheritance

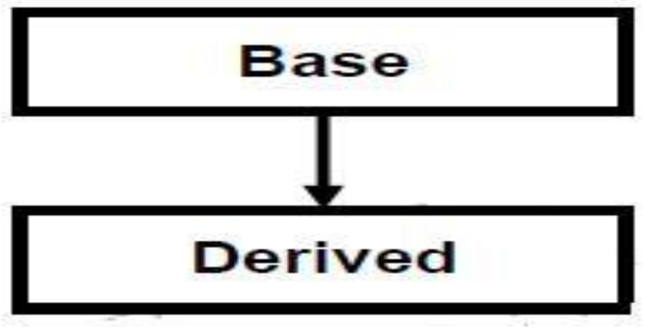
2) Explain about various types of inheritance.

In C++, we have 5 different types of inheritance. Namely,

1. Single Inheritance
2. Multiple Inheritance
3. Multi Level Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

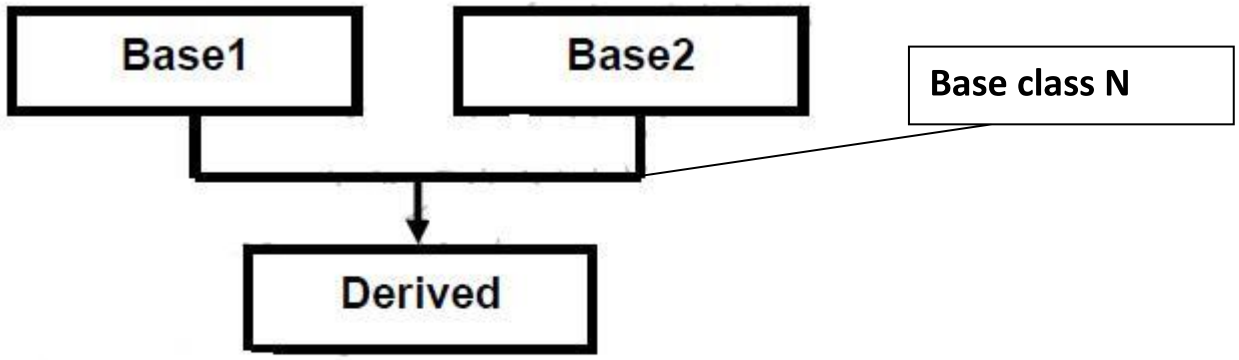
1) Single Inheritance:

when a single derived class is created from a single base class then the inheritance is called as single inheritance.



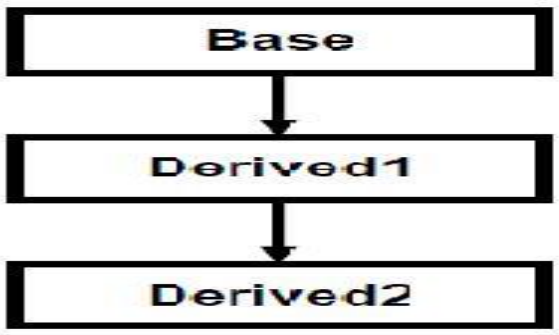
2) Multiple Inheritance

when a derived class is created from more than one base class then that inheritance is called as multiple inheritance.



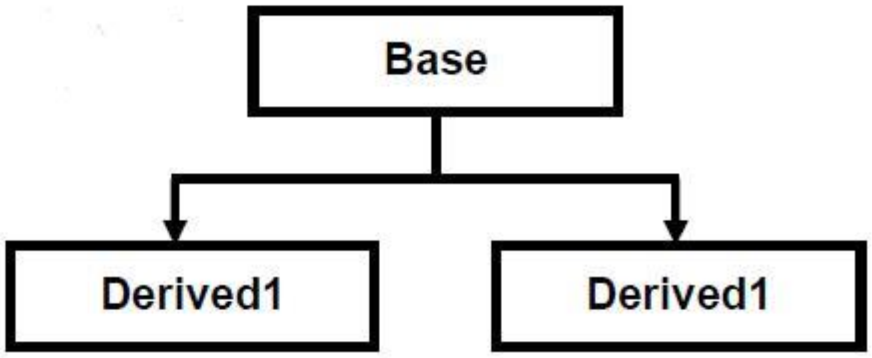
3) Multi Level Inheritance

when a derived class is created from another derived class, then that inheritance is called as multi level inheritance.



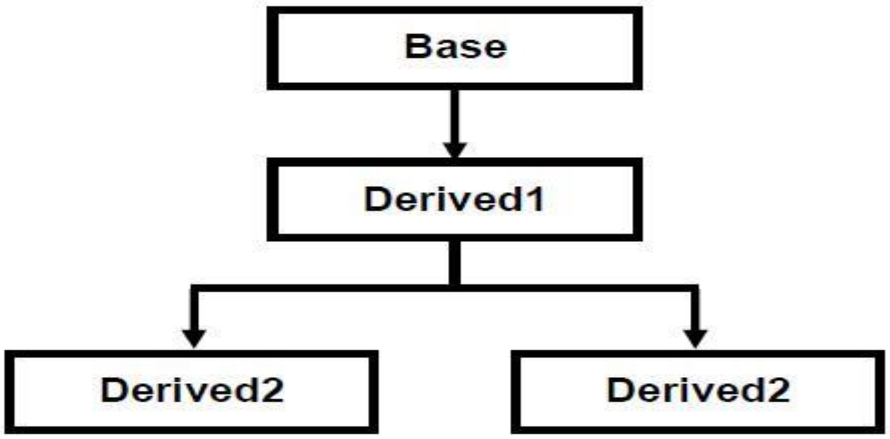
4) Hierarchical Inheritance

when more than one derived class are created from a single base class, then that inheritance is called as hierarchical inheritance.



5) Hybrid Inheritance

Any combination of single, hierarchical and multi level inheritances is called as hybrid inheritance.



Programs:

1)Write a C++ Program on <u>Single Inheritance</u> using <u>Public</u> access specifier

```
#include<iostream.h>
//base class
class B
{
int a;
public:
int b;
void get_ab()
{
a=5;
b=10;
}
int get_a()
{
 return a;
}
void show_a()
{
cout<<"a="<<a<<endl;
}
};
class D : public B
{
int c;
public:
void mul()
{
c=b*get_a();
}
void display()
{
cout<<"a="<<get_a()<<endl;
cout<<"b="<<b<<endl;
cout<<"c="<<c<<endl;
}
};
```

```
void main()
{
D d;
d.get_ab();
d.mul();
d.display();
d.show_a();
}
```

2) Write a C++ Program on Single Inheritance using Private access specifier

```
#include<iostream.h>
//base class
class B
{
int a;
public:
int b;
void get_ab()
{
a=5;
b=10;
}
int get_a()
{
 return a;
}
void show_a()
{
cout<<"a="<<a<<endl;
}
};
//derived class
class D : private B
{
int c;
public:
void mul()
{
get_ab();
c=b*get_a();
}
void display()
{
cout<<"a="<<get_a()<<endl; //or show_a();
cout<<"b="<<b<<endl;
cout<<"c="<<c<<endl;
}
};
void main()
{
D d;
d.mul();
d.display();
}
```

3) Write a C++ Program on <u>multiple inheritance.</u>

```
#include<iostream.h>
class student
{
protected:
int rno,m1,m2;
public:
void getdata()
{
cout<<"Enter the Roll no :";
cin>>rno;
cout<<"Enter the two subject marks:";
cin>>m1>>m2;
}
};
class sports
{
protected:
int sm;              // sm = Sports mark
public:
void getsm()
{
cout<<"\nEnter the sports mark :";
cin>>sm;
}
};
class statement:public student,public sports
{
int tot,avg;
public:
void display()
{
tot=(m1+m2+sm);
avg=tot/3;
cout<<"\nRoll No: "<<rno;
cout<<"\nTotal: "<<tot;
cout<<"\nAverage: "<<avg;
}
};
void main()
{
statement obj;
obj.getdata();
obj.getsm();
obj.display();
}
```

Output:

```
        Enter the Roll no: 100
        Enter two marks
        90
        80
        Enter the Sports Mark: 90
        Roll No: 100
        Total    : 260
        Average: 86.66
```

Explanation: (ALGORITHM)

```
Step 1: Start the program.
Step 2: Declare the base class student.
Step 3: Declare and define the function getdata() to get the student details.
Step 4: Declare the other class sports.
Step 5: Declare and define the function getsm() to read the sports mark.
Step 6: Create the class statement derived from student and sports.
Step 7: Declare and define the function display() to find out the total and average.
Step 8: Declare the derived class object,call the functions getdata(),getsm() and display().
Step 9: Stop the program.
```

4) Write a C++ Program on <u>Multilevel inheritance</u>

```
#include<iostream.h>
class Student
{
protected:
int rno;
public:
void get_number(int a)
{
rno=a;
}
void put_number()
{
cout<<"Roll number is "<<rno<<"\n";
}
};
class Test:public Student
{
protected:
float sub1,sub2;
public:
void get_marks(float a,float b)
{
sub1=a;
sub2=b;
}
void put_marks()
{
cout<<"marks in sub1= "<<sub1<<"\n";
cout<<"marks in sub2= "<<sub2<<"\n";
}
};
class Result:public Test
{
float total;
public:
void display()
{
total=sub1+sub2;
put_number();
put_marks();
cout<<"total = "<<total<<endl;
}
};
void main()
{
Result obj;
obj.get_number(14);
obj.get_marks(75.0,59.5);
obj.display();
}
```

<u>Output:</u>

```
Roll number is 14
Marks in sub1=75.0
Marks in sub2=59.5
Total=134.5
```

5)Write a C++ Program on <u>Hierarchical Inheritance</u>

```
#include<iostream.h>
class A  //Base Class
{
public:
int a,b;
void getnumber()
{
cout<<"\n\nEnter Number:\t";
cin>>a;
}
};
class B : public A  //Derived Class 1
{
public:
void square()
{
getnumber();  //Call Base class property
cout<<"\n\n\tSquare of the number:\t"<<(a*a);
}
};
```

```
class C :public A //Derived Class 2
{
public:
void cube()
{
getnumber(); //Call Base class property
cout<<"\n\n\tCube of the number :\t"<<(a*a*a);
}
};
void main()
{
B b1;        //b1 is object of Derived class1
b1.square();  //call member function of classB
C c1;        //c1 is object of Derived class2
c1.cube();    //call member function of classC
}
```

6) Write a C++ Program on <u>Hybrid Inheritance</u>

```
#include<iostream.h>
class student
{
protected:
int rollno;
public:
void get_number(int a)
{
rollno=a;
}
void put_number()
{
cout << "Roll Number Is:"<<rollno <<"\n";
}
};
class marks : public student
{
protected:
int sub1;
int sub2;
public:
```

```
void get_marks(int x,int y)
{
sub1 = x;
sub2 = y;
}
void put_marks(void)
{
cout << "Subject 1:" << sub1 << "\n";
cout << "Subject 2:" << sub2 << "\n";
}
};
class extra
{
protected:
float e;
public:
void get_extra(float s)
{
e=s;
}
void put_extra(void)
{
cout << "Extra Score::" << e << "\n";}
};
class result : public marks, public extra
{
protected:
float tot;
public:
void display(void)
{
tot=sub1+sub2+e;
put_number();
put_marks();
put_extra();
cout << "Total:"<< tot;
}
};
void main()
{
```

```
result std1;
std1.get_number(10);
std1.get_marks(10,20);
std1.get_extra(33.12);
std1.display();
}
```

Polymorphism

Poly means many; morphism means forms i.e. polymorphism means one thing we can express in many forms. **Example:** function overloading, operator overloading and function overriding.

3) Explain about FUNCTION OVERLOADING with example

- ✓ It is a type of polymorphism.
- ✓ We can use same function name to more than one member function.
- ✓ Overloading refers to the use of the same thing for different purpose. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as function polymorphism.
- ✓ Using the concept of function overloading, we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the arguments list in the function call.
- ✓ Functions are identified which has same function name by the compiler based on following two things;
  1. Different number of arguments.
  2. Different types of arguments.

Example:

```
Void   print()
{
........
........
}
Void   print(int i)
{
......
......
}
Void   print(int i,float j)
{
```

```
.....
.....
}
Void   print(float a, float b)
{
.....
.....
}
Void   print(float a, float b, int c)
{
.......
.......
}
```

Explanation:

In above example , we used function name as **print** for five functions with different arguments like function 1 has zero arguments ,function 2 has one argument ,function 3 and function 4 has two arguments ,function 5 has three arguments.,function 3 and function 4 has two arguments ,function 5 has three arguments.

Programs on function overloading:

1) Write a c++ program of function overloading to calculate volume of cube , cuboid ,cylinder .

```
#include<iostream.h>
Class    functionoverloading
{
Public:
int   volume(int a)                //for cube
{
return (a*a*a);
}
int  volume(int a, int b, int c)           //for cubiod
{
return(a*b*c)
}
Int    volume (int r, int h)                //for cylinder
{
return(3.14*r*r*h)
}
};
```

```
void  main()
{
Cout<<"volume of cube is:"<<volume(10)<<endl;
Cout<<"volume of cuboid is:"<<volume(10,20,30)<<endl;
Cout<<"volume of cylinder is:"<<volume(10,20)<<endl;
}
```

Similarly ,

2) C++ program to calculate area of square, triangle , rectangle.
3) C++ program to calculate perimeter of square, rectangle, triangle.
   These two programs refer in class notes

4) Explain about FUNCTION OVERRIDING with example

- ✓ Function overriding means two or more functions have same function name with same arguments i.e. number of arguments and type of arguments are same.
- ✓ If we inherit a class into the derived class and provide a definition for one of the base class's function again inside derived class then that is said to be function overriding.
- ✓ **Requirements for function overriding:**
  - To handle function overriding we have to use inheritance. That means we need inheritance i.e. we require a derived class and base class.
  - Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

Example:

Program on function overriding

```
#include<iostream.h>
Class   base
{
int  a;
public :
void  input()
{
Cout<<"enter  a:";
Cin>>a;
}
Void  show()
{
Cout<<"a="<<a<<endl;
}
};
Class  derived : public base
{
int  a;
public :
void input()
{
cout<<"enter a" ;
cin>>a;
}
Void   show()
{
Cout<<"a="<<a<<endl;
}
};
Void   main()
{
Cout<<"base class functions are executed"<<endl;
Base  b1 ;
 b1.input() ;
b1.show() ;
cout<<"base class functions are overridden in derived class"<<endl;
derived  d1 ;
d1.input() ;
d1.show() ;
}
```

5) Explain about VIRTUAL FUNCTIONS with example

Introduction:

- ✓ When the situation comes like both base class and derived class having same function name and with same arguments then the compiler will get confused like which function to execute first.
- ✓ There must be a provision to use the member function of both base class and derived classes using the same interface.
- ✓ This problem can be achieved by a new c++ concept called virtual functions.
- ✓ Virtual Functions provides a solution to invoke the exact version of the member function, which has to be decided at runtime. They are the means by which function of the base class can be overridden by the functions of the derived class.

- ✓ The keyword **VIRTUAL** provides a mechanism for defining the virtual functions. When declaring the base class member functions, the keyword VIRUTAL is used with those functions, which are to be bound dynamically.
- ✓ Virtual functions should be defined in the public section of a class when such a declaration is made; it allows to decide which function to be used at runtime, based on the type of object, pointed to by the base pointer, rather than the type of the pointer.

Syntax to declare virtual functions:

```
Class   classname
{
public:
          virtual   returntype   functionname (arguments)
          {
          }
};
```

Example:

```
Class   base
{
Public :
Virtual   void  show()
{
Cout<<"\n show base";
}
};
```

Here base is a class name and show() is a virtual function name

Rules for Virtual Functions:

- ✓ Virtual function must be a member of a class.
- ✓ They cannot be static(fixed) members
- ✓ They are accessed using object pointers.
- ✓ They should be declared in the public section of a class.
- ✓ They can be friend function to another class.
- ✓ Its prototype in a base class and derived class must be identical for the virtual function to work properly.
- ✓ The class cannot have virtual constructors, but can contain virtual destructor.

- ✓ When a virtual function in a base class is created, there must be definition of the virtual function in the base class even if base class version of the function is never actually called.

Program on virtual functions

```
#include<iostream.h>
Class   base                    //base class
{
Public :
     Void  display()                  //display() is normal member function
     {
     Cout<<”\n display base”;
     }
     Virtual   void  show()           //show() is virtual member function
     {
     Cout<<”\n show base”;
     }
};
Class  derived : public  base       //derived class
{
Public :
     Void  display()
     {
     Cout<<”\n display derived”;
     }
     Void  show()
     {
     Cout<<”\n show derived”;
     }
};
void   main()
{
     Base  b;                   //base class object
     Derived  d;                //derived class object
     Base    *bptr;             //base pointer for base class
     Cout<<”\n  bptr points to base \n”;
     bptr =&b;                  //storing base class object in to bptr
     bptr ->display();
     bptr ->show();
     Cout<<”\n \n  bptr points to derived \n “;
     bptr=&d;                   //storing derived class object in to bptr
     bptr -> display();
     bptr -> show();
}
```

Output:

```
bptr points to base
Display base
Show base
bptr points to derived
Display derived
Show base
```

Pure Virtual Function

6) Explain about pure virtual function with example

Introduction:

Sometimes implementation of all functions cannot be provided in a base class because we don't know the implementation. Such a class is called abstract class. We cannot create objects of abstract classes.

For **example**, let Shape be a base class. We cannot provide implementation of function draw () in Shape, but we know every derived class must have implementation of draw ().

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration.

What is a Pure Virtual Function? (Definition)

A Pure Virtual Function is a Virtual function with no body.

(Or)

A pure virtual function is a virtual function for which we don't have implementation of the function.

(Or)

A pure virtual function is a virtual function which doesn't have function definition.

Declaration of Pure Virtual Function:

Since pure virtual function has no body, the programmer must add the notation =0

for declaration of the pure virtual function in the base class.

Syntax to declare pure virtual function:

```
class  classname
{
public:
virtual void virtualfunctioname ( ) = 0      //This denotes the pure virtual
                                               function in C++
};
```

Example:

```
 class Test
 {
   public:
   virtual void show() = 0;            // Pure Virtual Function
  };
```

Here Test is called abstract class, because show() does not contain definition of the function (i.e. function body). We will implement pure virtual functions in derived class.

Program On Pure virtual functions

```
#include<iostream.h>
class Base             //Base class
{
 public:
int x;
 void  getX(int a)
{
x=a;
}
virtual void fun() = 0;      //Here fun() is pure virtual function
};

// This class inherits from Base and implements fun()
class Derived: public Base
{
public:
void fun()
{
cout << "pure virtual function is called";
}
};
```

```
void main()
{
   Derived d;
   d.fun();
 }
```

Output:

```
Pure virtual function is called
```

Note:

- ✓ Any class contains at least one pure virtual function then that class is called abstract class .
- ✓ Always declare pure virtual function in public section.
- ✓ We are going to implement pure virtual function in derived class.
- ✓ An abstract class can have constructors.

Unit IV

What is a Template ?

A template in C++ is a Generic Framework to work with different data types. Templates in C++ allows us to create or define generic Classes and functons. It supports a different data type in a single frame work. We can construct, template functions and template classes to perform the same operations on different data types.

Class template:

Similar to functions, classes can also be declared to operate on different data types. Such classes are called class templates. The class templates model a generic class which supports similar operations for different data types

General syntax:

```
template <class T>
Class classname
{
        //…………………….
        //class member specification with
        //anonymous type T wherever appropriate
       //…………………
};
```

The prefix template <class T> specifies that a template is being declared and a data type name T will be used in the declaration. Any call to the template function and classes need to be associated with a data type in a class.

Syntax for creating objects using the class – templates:

classname <datatype> objname(arglist);

Example for class templates:

```
#include<iostream.h>
const size=3;
template<class T>
class vector
{
T *v;
public:
vector()
{
        v=new T[size];
        for(int i=0;i<size;i++)
        v[i]=0;
}
vector(T *a)
{
        for(int i=0;i<size;i++)
        v[i]=a[i];
}
T operator *(vector &y)
{
        T sum=0;
        for(int i=0;i<size;i++)
        sum+=this->v[i]*y.v[i];
        return sum;
}
};
int main()
{
        int x[3]={1,2,3};
        int y[3]={4,5,6};
        vector <int> v1;
        vector <int> v2;
        v1=x;
        v2=y;
        int R=v1*v2;
        cout<<"R="<<R<<"\n";
        return 0;
}
```

Class templates with multiple parameters

A template can have multiple arguments i.e, in addition to the data type arguments T we can also use other arguments such as built in data type, constant expressions, function names, strings.

General Syntax:

```
        template<class T1,classT2,………..>
        class classname
        {
                ………………
                …………….
                …………….
        };
```

The argumetns must be supplied whenever a template class created.

Creating objects with multiple arguments in a template class:

```
        Classname< datatype,datatype,….> objname(arglist);
```

Example program for class template with multiple parameters:

```
#include<iostream.h>
template<class t1,class t2>
class test
{
        t1 a;
        t2 b;
        public:
        test(t1 x,t2 y)
        {
                a=x;
                b=y;
        }
        void show()
        {
                cout<<a<<"and"<<b<<"\n";
        }
};
int main()
{
        test <float,int> test1(1.23,123);
        test <int,char> test2(100,'w');
        test1.show();
        test2.show();
        return 0;
}
```

Function Templates:

There are several functions which have to be used frequently with different data types. The limitation of such functions is that they operate only on a particular data type. It can be overcome by defining that function as a function template (or) generic function. The C++ template feature enables substitution of a single piece of code for all overloaded functions with a single template function.

```
template <class T>
returntuype  function_name (arguments of type T)
{
---------------//body of function with type T
---------------//wherever appropriate
--------------
}
```

General syntax :

A function template is pre-defined with the keyword template and a list of template type arguments. These template type arguments are called generic data types, because their exact representation is now known in the declaration of the function template.

Example program for function templates:

```
#include<iostream.h>
template<class T>
void bubble(T a[],int n)
{
for(int i=0;i<n-1;i++)
for(int j=n-1;i<j;j--)
if(a[j]<a[j-1])
{
        swap(a[j],a[j-1]);
}
}
template<class X>
void swap(X &a,X &b)
{
        X temp=a;
        a=b;
        b=temp;
}
int main()
{
int x[5]={10,50,30,40,20};
float y[5]={1.1,4.4,3.3,5.5,2.2};
bubble(x,5);
bubble(y,5);
cout<<"sorted x-array:";
for(int i=0;i<5;i++)
cout<<x[i]<<"  ";
cout<<endl;
cout<<"sorted y-array:";
for(int j=0;j<5;j++)
cout<<y[j]<<"  ";
cout<<endl;
return 0;
}
```

Function template with multiple parameters

Like template classes we can use more than one generic data type in the template statement using a comma separated list in the template functions.

General syntax:

```
        template<class T1,class T2,………>
        returntuype  function_name (arguments of type T1,T2,…..)
        {
        ---------------
        ---------------//body of function
        --------------
        }
```

Example program for function template with multiple parameters:

```
#include<iostream.h>
#include<string.h>
Template<class T1, class T2>
void display(t1 x,t2 y)
{
        cout<<x<<” ”<<y<<”\n”;
}
```

```
int main( )
{
        display(1999,”EBG”);
display(12.34,1234);
return 0;
}
```

Overloading of Template functions

A template function may be overloaded either by template functions or ordinary functions of its name. In such cases, the overloading resolution is accomplished as:

- Call an ordinary function that has an exact match.
- Call a template function that could be created with an exact match.
- Try normal overloading resolution to ordinary functions and call the one that matches.

An error is generated if no match is found. Note that no automatic conversions are applied to arguments on the template functions.

Example program for overloading of function templates

```
#include<iostream.h>
#include<string.h>
template<class T>
void display(T x)
{
        cout<<”Template display:”<<x<<”\n”;
 }
void display(int x)
{
        cout<<”Explicit display:”<<x<<”\n”;
}
int main( )
{
display(100);
display(12.34);
display(‘C’);
return 0;
}
```

Member function template

General syntax:

```
  template<class T>
  returntype classname <T>::functionname(arglist)
  {
```

```
                //…………..
                //……………function body
                //………..
 }
```

Exception Handling

2) Explain exception Handling mechanism.

Introduction:

- The two most common types of bugs are logic errors and syntactic errors.
- The logic errors occur due to poor understanding of the problem and solution procedure.
- The syntactic errors arise due to poor understanding of the language. We can detect these errors by using debugging and testing procedures. We often come across some peculiar problems other than logic or syntax errors. They are known as exceptions.
- **Exceptions** are run time anomalies or unusual conditions such as division by zero, access to an array outside of its bounds, or running out of memory or disk space. When a program encounters an exceptional condition, it is important to identify and deal effectively with exception.
- **Basics of Exception Handling:**
  Exceptions are of two kinds**.**
  Synchronous exceptions: Errors such as “out of range index” and “over flow” belong to the synchronous type of exceptions.
  Asynchronous exceptions: The errors that are caused by events beyond the control of the program (such as keyboard interrupts) are called Asynchronous exceptions.
- The purpose of the exception handling mechanism is to detect and report an error so that correct action can be taken. For this, a separate error handling code that performs the following:

1. Find the problem. (Hit the exception).
2. Inform that an error has occurred. (Throw the exception)
3. Receive the error information. (Catch the exception)
4. Take corrective actions. (Handle the exception)

Exception handling mechanism:

C++ exception handling mechanism is built up on three blocks(keywords).

1. **try**
2. **throw**
3. **catch**

try : The key word ‘**try’** is used to preface a block of statements which may generate exceptions or which may raise exceptions. This block of statements is known as try block.
throw: When an exception is detected, it is thrown using a **throw** statement in the try block.
Catch: A catch block defined by the keyword catch ‘catches’ the exception ‘**thrown**’ by the throw statement in the try block, and handles it appropriately.

Catch block

Catches and handles the exception

try block

Detects and throws
an exception

Try block throwing exception

The catch block that catches an exception must immediately follow the try block that throws the exception. The general form of these two blocks are as follows:

```
try
{
....................
throw exception;
..........
}
catch(type arg)
{
...........
.........
}
```

When the try block throws an exception, the program control leaves the try block and enters the a catch statement of the catch block.

```
#include<iostream.h>
void main()
```

```
{
        int a,b;
        cout <<"Enter values for a and b";
        cin >> a;
        cin >> b;
        int x = a – b;
        try
        {
        if(x != 0) {
                cout <<"Result(a/x) =" << a /x << endl;
                }

        else
                {
                        throw(x);
                }
        }
        catch(int i)
        {
                cout <<"Exception caught: x = "<<x << endl;
        }
}
```

Multiple catch statements

It is possible that a program segment has more than one condition to throw an exception. In such cases, we can associate more than one catch statement with a try as shown below:

```
try
{
        //try block
}
catch(type1 arg)
{
        //catch block1
}
catch(type2 arg)
{
        //catch block2
}
```

```
                catch(type3 arg)
                {
                        //catch block3
                }

                ……………..
                ………………
                catch(typen arg)
                {
                        //catch blockn
        }
```

Example program for multiple catch statements:

```
#include<iostream.h>
#include<conio.h>
void test(int x)
{
  try
  {
          if(x>0)
            throw x;
      else
            throw 'x';
  }
   catch(int x)
  {
          cout<<"Catch a integer and that integer is:"<<x;
  }
   catch(char x)
  {
          cout<<"Catch a character and that character is:"<<x;
  }
}
 void main()
{
  clrscr();
  cout<<"Testing multiple catches\n:";
  test(10);
  test(0);
  getch();
}
```

Catch all Exceptions

In some situations, we may not be able to anticipate all possible types of exceptions and therefore may not be able to design independent catch handlers to catch them. In such circumstances, we may use a single catch statement to catch all exceptions instead of a certain type alone. The catch statement can defined as follows:

```
catch(…)
{

        //statements for processing
        //all exceptions
}
```

Example program for catching all exceptions

```
#include<iostream.h>
void  test(int  x)
{
        try
        {
                if(x= =0) throw x;
                if(x= = -1) throw ‘x’;
                if(x= =1) throw 1.0;
        }
catch(...)
{
        Cout<<”caught an exception\n”;
}
}
int main()
{
        cout<<”Testing Generic catch\n”;
        test(-1);
        test(0);
        test(1);
        return 0;
}
```