Page No.
Date:

Q1. What is Object Oriented Programming?

Ans It is a type of programming in which emphasis is given on data and binding of data structure is done with methods that operate on data ~~is done~~.

Q2. Object & Classes.

Object :

An object is the primitive element of a program written in OOP language. Each object consists of a set of procedures and some data i.e data members and member function. It stores the data in variables and responds to messages received from other object by executing the procedure. An object encapsulate both data and functions so it is also known as an abstract data type.

The syntax for defining an object of a class is as follows;

```
class_name  object_name;
```

Class:

A class in C++ is a user defined data type or data structure declared with keyword class that has data and functions as its member whose access is governed by three access specifiers private, protected or public. It is used to encapsulate data and function together in a single unit. It act as a blue print for similar kinds of objects as it defines the properties & behaviour of objects.

Generally a class specification has two parts:

1. Class declaration.
2. Class function definition

Syntax -

```
Class        classname
{
  . specifier : (Variable declaration)
                    data type variable name;
Data member

      specifier ( function declaration).
                return type function name(Args)
                {
member function   function definition
                }
            };
```

For example -

```
class sum                          → class.
{
private : int a, b, sum=0;
public : void getnum()
         {
          cout<<" Enter two nos.";
          cin >> a>>b;
         }
         void showsum()
          {
          cout<< endl<<"Sum = "<< a+b;
          }
       };

Void main ()
{
 class A;         → object
 A. getnum();
 A. showsum();
 getch();
}
```

Advantages of OOPs.

Some key advantages of OOPs includes the following:

1. Elimination of reductant code through inheritance by extending existing classes.

2. Higher productivity and reduced development

time due to reusablity of existing modules.

3. Secure programs as data cannot be modified or accessed by any code outside the class, due to the principle of data hiding.

4. Real world objects in the problem domain can be easily mapped ~~or~~ objects in the program.

5. A program can be easily divided into parts based on objects.

6. A data centered design approach captures more details of a model in a form that can be easily implemented.

7. Programs designed using OOPs are expandable as they can be easily upgraded from small to large systems.

8. Message passing between object simplifies the ~~interferance~~ interface description with external system.

9. Software complexity becomes easily manageable

10. With polymorphism, behaviour of functions, operators, or objects may vary depending upon the circumstances.

11. Data abstraction and encapsulation hides

implementation details from the external world and provides it clearly defined interface.

12. OOP enables programmers to write easily extendable and maintainable program.

Comparision between OOP and POP

	OOPs	PP
1.	Emphasis is given on Data.	Emphasis is given on algorithms. & procedure.
2.	Real world is represented by Objects mimicking external entities	Real world is represented by logical entitees and control flow.
3.	Allows modelling of real life problem into objects with state and behaviour	Tries to fit real life problem into procedure.
4.	Data is encapsulated effectively by methods.	Data and procedure are separate in a module.
5.	Program modules are integrated parts of overall programs. Objects interact with each other by message passing.	Program modules are linked through parameter passing mechanism.

6.	Uses abstraction at class and object level.	Uses abstraction at procedure level.
7.	Object Oriented decomposition focuses on abstracted objects and their interaction	Algorithmic decomposition tends to focus on the sequence of events.
8.	Active and intelligent data structures encapsulates all passive procedure	Passive and dumb data structures used by active methods.
9.	Does support to virtual function, polymorphism operator overloading.	Doesnot support virtual function polymorphism, operator overloading, inheritance etc.
10.	Ex - C++, Small talk Java, Javascript.	Ex - C, cobol, Pascal etc.

Characteristics of Procedural Programming:

1. Puts much importance on single thing to be done.

2. Large problems are divided into smaller programs known as function.

3. Most of the function share global data.

4. Data moves openly around the system from function to function.

5. Function transfer data from one form to another.

6. Employs top-down approach in program designing.
Data and procedure are separate in a module.

7. In the cases of large programs bringing change is difficult and time consuming.
Real world is represented by logical entities & control flow.

8. Appropriate and effective techniques are unavailable to secure data of a function from others.

9. POP contains steps by steps procedure to execute.

10. ~~The problem get decomposed into small parts.~~
Uses abstraction at procedural level.

Token in C++

Tokens are the basic building block in C++ language. It is the smallest individual unit, the program is constructed using a combination of tokens. following are the tokens in C++.

1. Keywords
2. Variables
3. Constants
4. Strings
5. Special Characters
6. Operators.

Keywords - C++ has a set of reserved words, i.e a sequence of characters that have a fixed meaning. All the keywoords must be written in lowercase. There are 48 keywoards in C++.

Variable -
A variable is defined as meaningful name given to the data storage location in computer memory. C++ supports two basic kinds of Variables:
i. Numeric Variable
ii) Character variable

Constant - Constant are identifiers whose value does not change. It is an explicit data value specified by the programmer. C++ allows the programmer to specify contant of integer type, floating point type, character type & String Type.

Declaration of a constant

const data type var_name = value;

Identifiers : Identifiers are basically the names given to a program elements such as variable, array and function.

Operators : An operator is defined as a symbol that specifies the mathematical, logical or relational operation to be performed.

Special Charaters : (,), {, }, :, ; comes under special characters.

Operators -

- Arithmetic - to perform arithmetic operations
- Relational - to compare two values.
- Equality - to compare for strict equality or inequal
- Logical - to perform logical operations.
- Unary - operates over single operand : ++ --
- Conditional - performs operation at bit level.
- Bitwise -
- Assignment - Assigning values to the variable
- comma - separates operands when chained tog
- Sizeof - used to calculate size of date type.

Data types

- Integer, char, float, double void – Basic
- Pointers functions, References, Arrays – Derived.
- Structure, class, Union, Enumeration, Typedef – User define

Features of Object Oriented Programming

① ②
Object & Classes. P.T.O.

③ Method and Message Passing:

Two objects can communicate with each other through messages. An Object ask another object to invoke one of its methods by sending a message. In reply to the message, the receiver sends the results of the execution to the sender. The messages that are sent to other objects consists of three aspects –

- the reciever object
- the name of the method the reciever should invoke
- the parameter that must be ~~invoked~~ used by the method.

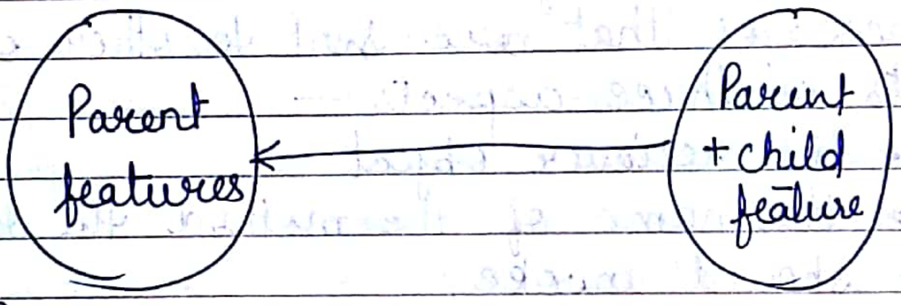
$$\begin{tikzcd} S.O \arrow[r, shift left] & R.O \arrow[l, shift left] \end{tikzcd}$$

A.getdata(2,4);

④ Inheritance

Inheritance is a concept of OOP in which a new class is created from an existing class. The new class often known as a sub-class, contain the attributes and methods of the parent class. ~~A sub-class not only~~.

Inheritance is the process by which Object of one class acquire properties of another class. It supports the concept of hierarchical classification.
The concept of inheritance provides the idea of reusablity. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from existing class. The new class will have the combined features of both the class. It allows the programmer to tailor the class in such a way that it does not introduce any undesirable side effects into the rest of the classes



Parent, base or super class.   child, derived or sub class.

⑧ Polymorphism:

Polymorphism is a concept that enables the programmer to assign different meaning or usage to a variable, function or an object in different context.
~~The~~ The Polymorphism can be applied to an operator, the process of making an operator

to exhibit different behaviour in different instances is known as Operator Overloading

Ex - a = 2 b = 3 c = a+b sum
a = oxford b = university c = a+b concatenation

Polymorphism can also be applied to a functions Using a single function name to perform different type of task is known as function overloading.

⑥ Containership :

The ability of a class to contain objects of one or more classes as member data, Containership is also called composition because the containers class in composed of contained class.

⑦ ~~Genericity~~ : Genericity.
To reduce code duplication and generate short simple code, C++ supports the use of generic codes or templates to define the same code for multiple data types. In other words a generic function can be invoked with arguments of any compatible type.

Generic programs, therefore act as a model of function or class that can be used to generat functions or classes. The generic programming is a technique of a programming in which a general code is written first. The code is instantiated only when need arises for specific

Types (provided as parameters)

(8) Reusablity :

Reusablity means developing codes that can be reused either in the same program or in different programs. In C++, reusablity is attained through inheritance, containership polymorphism and genericity.

~~Delagation~~:

(9) Data Abstraction and Encapsulation

Data Abstraction refers to the process by which data and function are defined in such a way that only essential details are revealed and the implementation details are hidden. Its main focus is to separate the interface and the implementation of a program.

Data encapsulation, also called data hiding is the technique of packing data and function into a single component (class) to hide implementation details of a class from the user. Encapsulation organises the data and methods into a structure that prevents data access by any function that is not specified in the class.

Add Dynamic Binding in features

Encapsulation defines three access level for data variables and member functions.

- lowel level data protection, accessed by any function belonging to any class.
- protected access level, accessed by that class or by any class that is inherited from it.
- highest level of data protection, accessed by the class in which it is declared.

Type Conversion

Type conversion of variable refers to changing a variable of one data type into another. It is done when the expression has variables of different data types. To evaluate the expression the data type is promoted from lower to higher level where hierarchy of data types can be given as — double, float, long, int short and char.

```
Ex- float f;              float x;
    int i;                int y = 3;
    i = f;                x = y;

int type                  float type
demoting.                 promoting
```

Type Casting

Type casting is also known as forced conversion. In an arithmetic expression, it tells the compiler to represent the value of the expression in a certain format. It is done when the value of a higher data type has to be converted into the value of a lower data type.

Ex –
```
float salary = 10000.00;
int sal;
sal = int (salary);
```

feature

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call.
Dynamic binding or late binding means that the code associated with a given procedure call is not known until the time of the call at run time. A function call associated with a polymorphism reference depends on the dynamic type of that reference.

Page No.
Date:

Enumerated Data types:

It is a user defined data Type, in which each integer value is assigned with an identifier i.e It consist of a set of named integer constant.

```
enum enumeration_name { id1, id2 };

enum color { Red, Blue, black };
```

Page No.
Date:

Unit - 2.

Defining a member function

Member functions can be defined either inside the class or outside the class.

• Defining a function Inside the class

In this method, function declaration or prototype is replaced with definition inside the class. A function defined inside the class is treated as an inline function by default, provided they do not fall into the restricted category of inline function. It increase the execution speed of the program, it consumes more space.

```
class rectangle
 {
    private : float length;
              float breadth;
    public : void get_data ()
             {
             cout<<" Enter length & breadth ";
             cin>> length >> breadth;
             }
 };
```

In the above code get_data() is an inline function of the class rectangle. This function is used to read the values of private data members of the class from the users

Page No.
Date:

• Defining a function outside

A member function is defined outside a class by specifying a membership identity label in the function header. This identity label informs the compiler about the class to which the function belongs to.
We can define a function outside the class as follows.

```
return-type class name :: function name (Args)
 {
  function body
 }
```

The scope resolution operator identifies and specifies the context to which an identifier refers.

```
float rectangle :: area (void)
 {
   return length * breadth ;
 }
```

Array of class objects.

```
#include <iostream.h>
#include <conio.h>
class student
{
 private : int rollno, age;
          char sex;
 public : void getinfo ()
          {
```

Scanned by CamScanner

```
cout<<"Enter roll nos";
cin >> rollno;
cout<< endl <<"Enter age";
cin>> age;
cout << endl <<" Enter sex";
 cin>> sex;
 }
void display info ()
 {
   cout<< endl <<" rollno:"<< rollno;
   cout<< endl <<" Age:" << age;
   cout<< endl<<" sex:" << sex;
 }
};
void main ()
 {
  student s[100];
  int i,n;
  cout<<endl <<" How many student";
  cin>> n;
  for ( i=0; i<=n; i++)
   {
     s[i]. getinfo();
   }
  cout<< endl <<" Entered Info";
  for (i=0; i<n; i++)
   {
     s[i]. disinfo();
   }
  getch();
}
```

Syntax —
class-name array-name[size];

Pointers and classes.

Constructor

A constructor is a special member function of a class which is automatically invoked at the time of creation of an object to initialize or construct the values of data members of the object.

1. The name of the constructor is the same as that of the class to which it belongs.

2. A constructor must be declared in the public section

3. It should not be explicitly called because a constructor is automatically invoked when an object of a class created

4. A constructor can never return any value therefore unlike a normal function, a contructor does not have any return value.

5. A constructor cannot be inherited and virtual

6. Pointers and references do not work with constructor.

7. A constructor can not be declared as static volatile or const.

8. Like a normal function a constructor function can also be overloaded.

9. It can also have default arguments.

Syntax —

```
class classname
      {
        private : ____
        public  : class name ()
                  {
                  }
      };
```

Example.

```
#include <iostream.h>
# include < conio.h>
class demo
      {
       private :
       public: demo ()
               {
               int a,b;
               ~~clrscr~~ clrscr();
               cout<<"Enter values";
               cin >> a >> b;
               }};
```

```
void main ()
{
 demo d1, d2;
 getch();
}
```

Output
```
Enter values 2
3
Enter values 4
5.
```

Here constructor is used to initialize the objects, d1 and d2.

Types of Constructors

There are five types of constructors.

- Dummy
- Default
- Copy
- Dynamic
- Parameterized

Dummy Constructor

Dummy constructor also known as 'Do Nothing' constructor is a time mechanism which does not perform any action, when the program has been written without any constructor. Dummy constructor does not initialize any data member and thus, the variables acquire garbage value

Page No.
Date

Ex:-

```
# include <iostream.h>
# include <conio.h>
class numbers
      {
        private : ~~x~~ int x;
        public : void showdata ()
                 {
                  cout << endl << "x =  " << x;
                 }
       };
void main ()
 {
   Numbers n;
   n. showdata
   getch();
 }
```

Output
x = 3983 (garbage value) .

Default Constructor .

A constructor that does not take any arguments is called a default Constructor. The default constructor simply allocates storage for the data members of the obje

Ex

```
# include < iostream.h>
# include < conio.h>..
Class demo ()
{
```

```
private: int x;
public : demo ()
         {
           x = 2;  cout<<" x =  "<< x;
         }
};
void main ()
{
demo d1;
d1. getch();
}.
```

Output -
```
x = 2
```

Parameterized Constructor

A constructor That accepts one or more parameters or arguments is called as parameterized constructor

Ex-
```
#include <iostream.h>
#include <conio.h>
class demo
 {
 private: int x;
 public : demo ( int a)
            {
              x = a   cout<<" Value of x =  "<<x;
            }};
void main ()
{
demo d1(5);
getch(); }
```

Output :

x = 5.

Copy constructor

A copy constructor takes an object of the class as an argument and copies data values of member of one object into the values of another object. Since it takes only one argument, it is also known as a one argument constructor. The primary use of a copy constructor is to create a new object from an existing one by initialization. For this copy constructor takes a reference to an object of the same class as an args.

Ex-
```
# include <iostream.h>
# include <conio.h>.
class numbers
{
 private ~~x~~ : int x;
 public      : number(~~int~~ N2 &i)
                {
                 x = i.x;
                }
                numbers int n)
                {
                 x = n
                }
               void showdata ()
```

Page No.
Date:

```
cout << " x = " << x;
}
};
void main ()
{
 Number N1 (20);
 Number N2 (N1);
 N2.showdata ();
 Number N3 = N1;
 N3.showdata ();
 getch();
}.
```

N1 x = 20.

N2 x = i.x
x = 20.

N3 = x = 20.

Output -
```
x = 20
x = 20.
```

~~Why do we take contru~~

Why do copy constructor take objects by reference and not by value?

When an object is passed by value, the copy constructor is implicitly called to create a copy of the original arguments.
If the copy constructor had been designed to accept the object by value, then it would have resulted in infinite recursion. To avoid such a situation, C++ mandates the copy constructor's parameters to be passed by reference.

Page No.
Date:

Dynamic Constructor

Dynamic constructor are those constructors in which memory for data members is allocated dynamically.
It enables the program to allocate the right amount of memory to data members of the object during execution.
This is even more beneficial when the size of data members is not same each time the program is executed.
The memory allocated to the data members is released when the object is no longer required and when the object goes out of scope.

Ex.
```
#include <iostream.h>
#include <conio.h>
class array
 {
  private: int  *arr;
           int  n;
  public:  array()
           {
            n=0
           }
           array(int i);
           void showdata();
          }};
```

```
Array :: Array (int num)
```

```
~~Ott~~ n= num; ~~arr~~
arr = new int.(n);   Memory allocation for array
Cout<<"Enter the element";   dynamically
~~Cout<<"~~ for (int i =0; i<n ; i++)
        {
          cin >> arr [i];
        }

  }
void Array :: showdata ()
 {
  for (i=0; i<n; i++).
     {
      cout <<"   " << arr[i];
     }
void main ()
 {
  int size;
  Clrscr();
  Cout<<"Enter the size of array : ";
  cin >> size;
  Array arr (size);
  Arr. showdata ();
  getch ();
  }
```

~~Ext~~ Output :

```
Enter the size of array : 5
Element  1 2 3 4 5
1 2 3 4 5
```

Page No.
Date:

Constructor with default arguments.

While passing arguments to a constructor if any argument get missing, a default value ~~is assigned~~ to the ~~int~~ will be used to initialze the data member.

Ex:-
```
# include <iostream.h>
# include <conio.h>
class student
{
 private : int roll, marks;
 public  :  student ().
           {
           ~~ss~~ roll =0;
              marks=0;
           }
          student ( int r , int m =0)
          {
           roll = r;
           marks = m;
          }
        Void showdata()
         {
         cout << "roll nos =   " << roll;
         cout<<"marks =   " << marks;
        }};
void main ()
 {
student S1 ;
student S2 (10);
student S3 (10, 20);
S1. showdata ();
S2. showdata ();
S3. showdata ();
getch();
}.
```

Output -

```
Roll nos =   0
marks    =   0.
Roll no  = 10     marks = 0 ..
Roll no  = 10     marks = 20.
```

Constructor Overloading

Constructor can also be overloaded, when a class has multiple constructors, they are called as overloaded constructors. Some features of overloaded constructors are.

1. They have the same name as the class.
2. Overloaded constructor differ in their signature with respect to the no. and sequence of args.

When an object of class is created, the specific constructor is called.

```
# include <iostream.h>
# include <conio.h>
class demo
{
private : int a, b, ~~sum = 0~~;
Public : demo (int p, int q)
         {
         a ~~p~~= p;
         b = q;
         }
         demo (int o)
         {
          a =o;
          b =o;
         }
         void showdata ()
         {
         cout<<"sum = " << a+b;
         }
     };

Void main ()
{
 demo d1(2), d2(3,4);
 d1. showdata;
 d2. Showdata;
 getch();
}
```

Output ·

```
Sum = 4
Sum = 7
```

Destructor

A destructor is also a member function that is automatically invoked. The job of destructor is to destroy the object. It deallocates the memory dynamically allocated to the variables or perform other clean up operation.

Important features

1. The name of the destructor is also the same as that of the class. However, the destructor 's name is preceded by '~' tilde symbol.

2. A destructor is called when the object goes out of scope.

3. It is also called when the programmer explicitly deletes an object using the delete operator.

4. It is declared in the public section

5. The order of invoking the destructor is the reverse of invoking the constructor.

6. Destructor does not have any arguments, so can not be overloaded.

9. The address of a destructor cannot be accessed in program.

10. An object with a destructor or constructor can not be used as member of union.

11. Destructor can not be inherited

12. Destructor can not be virtual

13. A class can have only one destructor.

Ex-
```
# include < iostream.h>
# include < conio.h>
class sample
 {
 private : int x;
 public : sample ( int n)
           {
             x = n;
             cout<<" Constructor called with
                     value of x = " << x;
           }
           ~ Sample ()
           {
           cout <<" Destructor called for
            Objects with value x = " <<x;
           }
 };
```

```
void main ()
{
Sample S1 (2), S2(3), S3(4);
}
```

Output.

```
Cont. called for the object with value = 2
  "      "     "   "    "     "    = 3
  "      "     "   "    "     "    = 4
Destructor Called for the object with value = 4
  "      "     "   "    "     "    = 3
  "      "     "   "    "     "    = 2.
```

this pointer

In C++, this pointer is used to represent the address of an object inside inside a member function.
For example, consider an object obj calling one of its member function m1() as ~~o~~ obj.m1();
Then this pointer will hold the address of object obj inside the member function m1(). It acts as an implicit arg. to all the member function.

Ex
```
# include <iostream.h>
# include < conio.h>
class demo
  {
   private : int i;
   public : void  m1 ( int a)
                {
                  this -> i = a;
                }
          };
```

this pointer stores the address of object obj and access i.

```
void main ()
{
 demo obj;
 obj. m1(5);
 getch();
}
```

Uses of this pointer

* **to return same object**

One of the important application of using this pointer is to return the object it points Ex-

return * this;

inside a member function will return the same object that calls the function.

* **Distinguish Data Members**

Another application of this pointer is distinguishing data members from local variables of the member function if they have same name.

Ex.
```
# include <iostream.h>
# include <conio.h>
Class sample
{
  ~~int~~ int a, b;
  public: void input (int a, int b)
          {
            this -> a = a;
            this -> b = b;
}
void output ()
{
cout <<" a = " <<a<< endl;
cout <<" b = " <<b<< endl;
}};
void main ()
{
sample x;
x. input (5,8);
x. output ();
getch ();
}
```

A class sample is created in the program with data members a and b and member function input () and output (). Input () function receives two integers parameters a and b which are of same name as data member of class sample. When input () is called, the data of object inside it is represented as this -> a and this -> while the local ~~members~~ variables of function is simply represented as a and b.

Access Specifiers.

C++ support the principle of data hiding. The access restriction to a class member is specified by access specifiers. These are used to set boundaries for availability of member. There are three specifiers.

specifier specifies the class members declared under label public will be available to everyone i.e they can be accessed by other classes too. Therefore there are chances they might get change. So the key members must not be declared public.

```
Syntax: class  class-name
          {
            public:   data members;
                          :
                      member functions;
                          :
          };
```

private: It is the highest level of data hiding. The members which are declared or defined under private can not be accessed outside the class in which it is declared, neither it can be accessed by other classes too.

```
          private:
Syntax :  data members;
                 :
              member function;
```

protected: It is somewhere similar to private but it makes class member inaccessible outside the class. But they can be accessed by any sub class of that class.

Syntax : protected : data members;
:
member functions;

Ex.
```
# include <iostream.h>
# include < conio . h>
class ~~de~~ circle
{
private :  int r, pi=3.14, area =0;
protected : void area()
            {
              area = pi * r * r ;
              ~~cout~~ << " ~~Area~~ =  " << ~~area~~;
            }
public  :  void getdata()
            {
              cout << endl << " Enter radius";
              cin >> r;
            }
            ~~void show~~ ()
            {
              ~~cout<<"Radius = "<<r;~~
              ~~cout<<" Area = "<< area;~~
            }
 };

class peri_circle : public circle
  {
  private : float per;
  public  : void per ()
            {
```

Page No.
Date:

```
per = 2 * pi x r;
}
void show ()
{
cout << "Area =  " << area;
cout << "Perimeter =  " << per;
}
};

void main()
{
per P1;
P1.getdata ();
P1. area ();
P1. per ();
p1. show ();
p1. r = 12;      // Error.
getch();
}
```

Visibility / class	Same	Derived	Any other	Friend class	Friend function
Private	Yes	No	No.	Yes	Yes
Protected	Yes	Yes	No	Yes	Yes
Public	Yes	Yes	Yes	Yes	Yes

Page No.
Date:

Friend function :

A friend function of a class is a non-member function of the class that can access its private and protected members.
To declare an external function as a friend of the class, it must include function prototype in the class definition.
Syntax :

```
        Class   class_name
          {
            friend  return_type function_name
                                        (Args);
```

Some important points to be considered.

1. It is a normal external function that is given special privilege.

2. It is defined outside the class scope and cannot be called using : or -> operator.

3. It is not considered as the member of the class while the prototype is included in the class definition.

4. 'friend' declaration can be placed in public or private section.

5. 'friend' keyword is only used in declaration but not with definition.

- friend function do not require this pointer
- It can be member and friend of other class.

Ex-
```
# include <iostream.h>
# include <conio.h>
class cal
 {
  private : int a, b;
  public : void getdata ()
            {
             cout << "Enter two nos";
             cin >> a >> b;
            }
  friend
  ~~protected~~ : void avg ();

  protected : void sum ()
               {
                int sum=0;
                sum= a+b;
               }
 };

 void avg ()
  {
   float avg;
int sum = sum ();
  avg = sum/2;
  cout << endl << " Sum = "<< sum;
  Cout << " endl << " Avg = "<< avg;
 }
```

```
void main ()
{
cal  c1;
c1.getdata;
c1. cout<<" avg ();
getch ();
}
```

Friend Class.

A friend class is one which can access the private and protected members of another class in which its prototype declared. A friendship must be specified explicitly.

~~friend class name (forward declaration~~

```
class friend_class_name ; // forward Declaration
class  class_name
   {
     friend class(friend)_name ;
   } ;
```

Example -
```
# include <iostream.h>
# include <conio.h>
class B;
class A
{
  friend class B;
  private : int x;
  public : void getdata ()
         {
          cout<<" Enter x: ";
          Cin >> x;
}
void show ()
 {
cout<<" \n A's x = " << x;
 }
};

Class B
{
 private : int y;
 public : void getdata ()
          {
           cout<<" \n enter Y ";
           cin>> Y;
          }
         void showdata ()
          {
           cout<<" \n B's y = "<< y";
          }
 B       void swap (A &a)
          {
           int temp;
           temp = a.x;
           a.x  = y
            y   = temp;
          }
      };

Void main ()
   {
     A a;
     B b;
```

```
a. getdata ();
b. getdata ();
a. showdata ();
b. showdata ();
b. swap (a);
cout <<" After Swap";
a. showdata ();
b. showdata ();
getch ();
}
```

Output:

```
Enter x = 5
Enter y = 6.
A's x = 5
B'y y = 6.
After Swap.
A's x = 6.
B's ~~y~~ = 5
```

Page No.
Date:

Unit-3.

Inheritance:

Inheritance is the process by which object of one class acquires the properties of another class. The concept of inheritance provides the idea of reusablity. This means we can add additional features to an existing class without modifying it. This is possible by deriving a new class from an existing class.

Syntax
class child_name : public parent class

Learn from features

Types of Inheritance

1. Simple or Single inheritance
2. Multiple Inheritance
3. Multi level Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance
6. Multipath Inheritance

1. Simple or Single Inheritance

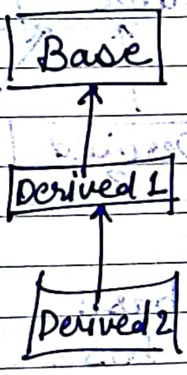
When a derived class inherits the features from a single base class, it is called as single or simple inheritance.

$$\begin{tikzcd}
\boxed{\text{Base}} \\
\boxed{\text{Derived}} \arrow[u]
\end{tikzcd}$$

class: child class : parent class -

Multi-level Inheritance

The technique of deriving a class from an already derived class is multi-level inheritance. In multi level inheritance, the number of levels can go upto any number based on the require -ment.



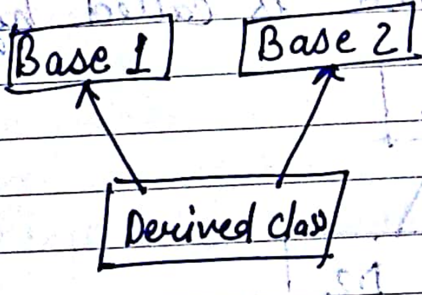
class derived 1: public base_class

Visiblity mode:

class derived 2: public derived 2

Multiple Inheritance

When a derieved class inherits features from more than one base class, it is called as Multiple Inheritance.



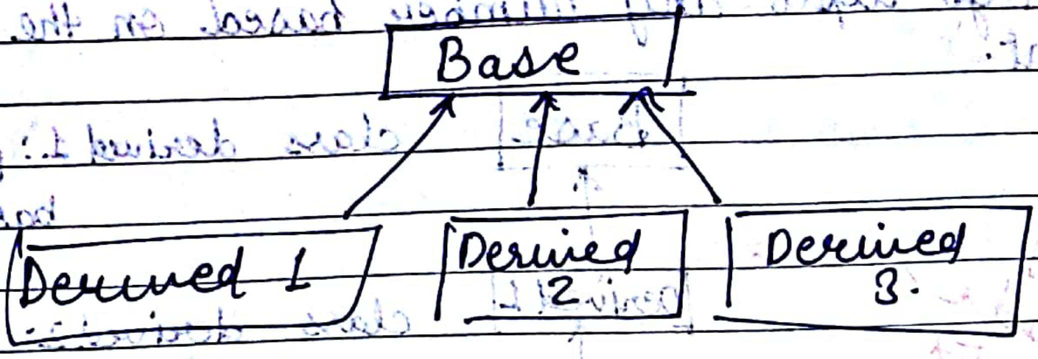
class derived: base1, base2

Define visiblity mode- such as

class derived: public base1, public base 2

Hierarchical Inheritance.

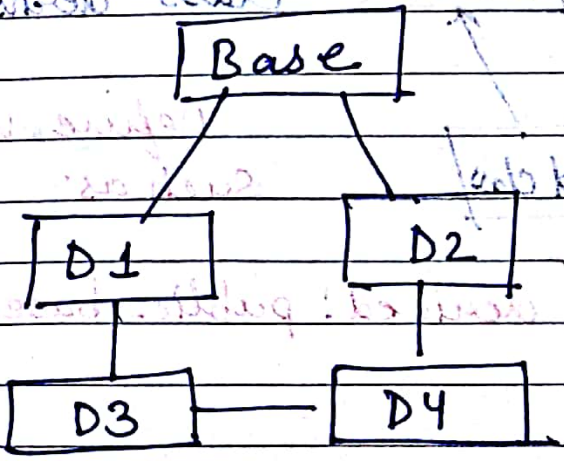
When a class is inherited by more than one class it is called hierarchical inheritance or we can say derivation of several child class from single base class is called as Hierarchical Inheritance.



Class derived1 : base.
class derived2 : base.
Class derived 3 : base.

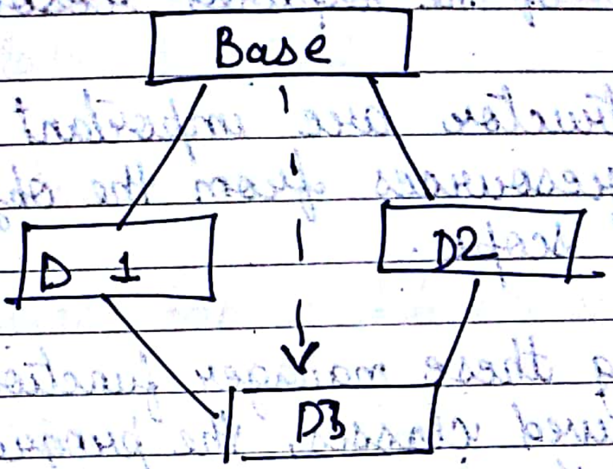
Hybrid Inheritance

Deriving a class that involves more than one form of inheritance is called hybrid inheritance.



Multi-path Inheritance

Deriving a class from other derived class, that are in turn derived from the same base class is called Multi path Inheritance.



Importance of Inheritance

1. Reusablity of ~~base~~ class code.
2. Enhance the reliablity of the code, as it is debugged and priorly tested code.
3. Reduces development cost and decreases the efforts and time spent on code maintenance.
4. Programmes need to focus only new or additio-al portion of the system.
5. Supports code extensiblity and allow function over-riding.
6. Facilitates class creation of class libraries.

Constructor and Destructor in derived class

A constructor plays an important role in initializing an object during creation and allocation of the required resources like memory.
While destructor are important for dealloca-ting the resources from the object when it goes out of scope.

While having these manager functions in base and/or derived classes, the programmer must keep the following in mind:

•1. In the base class constructor does not take any args, the derived class may not have a constructor.

•2 If the base class has a constructor with one or more args, then the base derived class must have a constructor function to pass the arg. to the base class constru

•3. It is the derived class's responsiblity to pass args to the base class Cz in the main () we will create objects only of the derived class ~~o~~ and not of base class.

•4 If both the derived class and ~~or~~ base class has constructors, the base class constructor is executed first and then the constructor in the derived class is executed

.5 The order of execution of destructor is just the reverse of constructor. First the destructor of derived class is called and then the ~~cons~~ destructor of base class is called.

Types of Base Class.

A base class can be defined into two types:

- Direct Base Class
- Indirect Base Class.

→ Direct Base Class

A base class is called a direct base class if it is mentioned in the base list. It appears directly as a base specifier in the declaration of its derived class.

```
Ex- class base A
    {
      -
      -
    }
    Class derived B : public base A
    {
      -
      -
    }
```

```
class base B
{
  —
}
Class derived C : public class A, public c/ B.
{
  :
  ;
}
```

Where both class A and B are direct base classes.

② Indirect base Class.

~~A class~~ An indirect base class is a base class that does not appear directly in the declaration of the derived class but is available to the derived class through on of its base classes. For a class all base classes that are not direct base classes are indirect base classes.

```
Class A
 {
  Public : int x
 }
Class B : public class A .
 {
  Public : int y;
 }
```

class C : public b
{
public : int z;
}

Unit 4.

File Handling - Handling data files

File - The information stored under a specific name on a storage ~~def~~ device.

Stream - It refers to a sequence of byte.

Text file - It is a file that stores info in ASCI characters. In text file each line of text is terminated with a special character as EOL

Binary file - It is a file that contain info. in the same format as it is held in the memory. no delimiters are used and no translation occured.

classes for file stream operations

ofstream - stream class to write on files.
ifstream - " to read from "
fstream - both read & ~~rd~~ write.

Opening a file

1) fopen() - creates a new file for use
Opens an existing file

2) fclose - to close an already open file

3) getc = Read a character from file

4) putc = Write a character to a file

5) fprint = write a set of data values to file

6) fscan = write reads a set of data values

7) getw = Read an integer value

8) putw = Write an integer value in file

9) fseek = set the position to desired point

10) ftell = tells the current position

11) rewind - sets posi to the begining

$\boxed{\text{fseek ftell rewind}}$

random access.

Templates

Templates are used to perform the same opration on different data types. It allows to develop reusable software components like functions, classes that supports different data types in a single work.
The template declared for a function is function template and those declared for a class is called class template.

Function template :
There are several function to be frequently used with different data types. The limitation of such functions is that they works only on a particular data types. A function template specifies how an individual function can be used with different data type.
A function template is prefixed with keyword TEMPLATE and a list of template type args.
Syntax :

```
template < class template name >
return type function name (args)
{
    function body
}
```

Ex. #
#

```
template <class t>
void swap ( t &a, t &b)
  {
     t c;
       c = a;
     a = b;
       b = c;
  }
void main ()
{
int i, j ;
float p,q;
char m,n;
cout<<"enter two integers";
cin>>a>>b;
swap(a,b);
cout<<"after swapping : <<i<<" "<<j<<en
Cout << "enter two real nos";
cin>>p>>q;
~~cout<<"~~ swap (p,q);
cout <<"after swapping <<p <<" "<< q<<end;
cout <<"enter two characters";
cin >>m >>n;
swap (m,n)
cout <<"after swapping "<<m<<" "<<n<<;
getch();
}
```

Class template

```
#
#
template < class t,> .
class sum
 {
  t a , t b;
  public : void sum ( t i, t j)
         {
           a = i;
           b = j;
         }
           void show sum ()
            {
            cout << a+b << endl;
            }};
void main ()
{ sum < int> ob1 ( 10,20);
  sum < float) ob2 ( 6.4, 9.3);
  Ob1 . Showsum ();
  Ob2 . showsum();
  getch();
}
```

Unusual Conditions

Exception Handling

Exceptional Handling is a feature in C++ that detect and report an exceptional condition so that appropriate action can be taken to deal with it

Exception handling are of two Type

1) Synchonous 10, array index unbound
2) Asynchronous disk failure, hordware half

Exception Handling deal only with Synchronous exceptions.

The steps performed in exceptional handling ~~can~~ comprise of two blocks:

1) Try block.
2) Catch block.

Try block: 1) Hit the exception by finding the problem.

2) Throw the exception by informing that an exception has occured

Catch block 1) Catch the exception by receiving info abt the exception.

2) Handle the exception by taking corrective action

```
try
  {
    throw exception;
  }
catch ( args )
  {
    -
    -
  }
```

```
{
return (x*x);
}
void main
{
int num
cout<<"Enter a nos";
cin>>num;
cout<<"square="<<num<<=<<
  sqr(num);
getch();
}
```

Page No.
Date:

Virtual function

It is a function in base class which is overidden in the derived class and which tell the compiler to perform late binding on this function.

```
class Base.
  {
   public : void show()
              {
           cout<<"Base";
              }
             virtual display()
              {
               cout <<"Base Class";
             } };
Class derived : public Base.
         {
      public: void show()
              {
                cout<<"Derived";
                }
                void display()
                   {
                    cout<<"derived class";
                   }
                 3}

Void main()
 {
  Base B;   Base *p;
```

Derived D;

```
p = &B;
p.show();
p.display();

p = &D;
p.show();
p.display();
getch();

}
```

Output :-

```
Base.
Base
Base
Derived.
```

Pure Virtual

If any function is only declared in base class, or The function that are not defined in base class are Called pure virtual function.

Memo Allo cation Dynam.

Ptr variable = new data type

```
int * arr.

  arr = new int

  delete array.
```