9. d) Distinguish between Dynamic binding and message passing.

Ans:

Difference between dynamic binding and message passing:

Dynamic binding	Message passing
① Dynamic binding is binding a call to a particular method at run-time. It is also referred to as late binding.	① Mesag passing is a form of communication on where objects exchange messages.
② Dynamic binding is needed when the compiler determines that there is more than one possible method that can be executed by a particular call.	② Message passing involves one object sending a message to another and receiving information.
③ It is associated with polymorphism and inheritance.	③ It involves specifying the name of the object, the name of the function (message) and the information to be sent.

What kinds of things can become objects in OOP?

Ans: 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 has to handle. They may also represent user-defined data such as vectors, time and lists. Each object contains data, and code to manipulate the data.

Describe inheritance as applied to OOP.

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

Each sub-class defines only those features that are unique to it. Without the use of classification, each class would have to explicitly include all of its features.

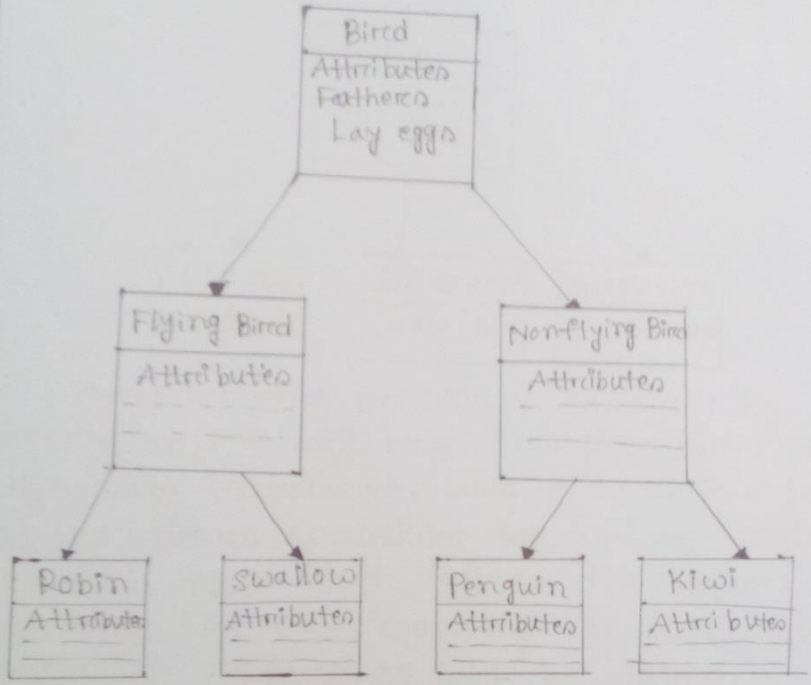


Figure: Propercty inheritance

12. What do you mean by dynamic binding? How it is useful in OOP?

Ans: 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 a given procedure call is not known until the time of the call at run-time. It is associated with poly-morphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

5. Can we assign a void pointer to an int type pointer? If not, why? How can we achive this?

Ans: We can not assign a void pointer to an int type pointer.

Because c++ does not allowed assign a void pointer to an int type pointer without using a cast pointer.

Example:

```
void * ptr 1;
char * ptr 2;
Ptr 2 = Ptr 1;
```

are not valid in c++.

A void pointer cannot be directly assigned to ~~at~~ other type pointers in c++.

To assign a void pointer to an int type pointer, we need to use a cast operator. in as shown below:

```
Ptr 2 = (char *) Ptr 1;
```

10. The const was taken from c++ and incorporated in ANSI c, although quite differently. Explain.

Ans: In both c and c++, any value declared as const cannot be modified by the program. However, there are some differences in implementation.

In c++, we can use const in a constant expression such as

```
const int size = 10;
char name[size];
```

This would be illegal in c.

c++ requires a const to be initialized. ANSI c does not requires an initializer, if none is given, it initializes the const to 0.

The scoping of const values differs. A const in c++ defaults to the integral linkage and therefore it is local to the file where it is declared. In ANSI c, const values are global in nature. They are visible outside the file in which they are declared. However, they can be made local by declaring them as static. To give a const value an external linkage so that it can be referenced from another file, we must explicitly define it as an extern in c++.

Example: extern const total = 100;

How does a constant defined by const differ from the constant defined by the preprocessor statement #define?

Ans: In both c and c++, any value declared as constant can not be modified by the program in any way. There are some difference in implementation.

Declare variable as constant:

In c++: const int size = 10;
char name[size];

const allows us to create typed constants. As with long and short, if we use the const modifier alone, it defaults to int.

const size = 10;

means,

const int size = 10;

In c: # define pi 3.1416;

Use # define to create constants that have no type information.

Initialization:

In c++: C++ requires a const to be initialized.

In c: ~~Ansi~~ ANSI c does not require an initializer, if none is given, it initializes the constant to zero.

8.17. What are the advantage of using new operators as compared to the function malloc()?

Ans: The new operator offers the following advantages over the function malloc():

- It automatically computes the size of the data object. We need not use the operator sizeof.
- It automatically returns the correct pointer type, so that there is no need to use a type cast.
- It is possible to initialize the object while creating the memory space.
- Like any other operator, new and delete can be overloaded.

3.16. What is the application of the Scope resolution operator :: in c++?

Ans: The Scope resolution operator :: allows access to the global version of a variable. For example; ::cout means the global version of the variable cout (not the local variable cout declared in that block). The following program illustrates this feature:

```
# include <iostream>
Using namespace std;
int m = 10;  //global m
int main()
   { int m = 20; //m redeclared, local to main
```

```
{ int k=m;
  int m=30; // m declared again,
                local to inner block
  cout<<"we are in inner block \n";
  cout<<"k="<<k<<"\n";
  cout<<"m="<<m<<"\n";
  cout<<"::m="<<::m<<"\n";
}
  cout<<"\n we are in outer block \n";
  cout<<"m="<<m<<"\n";
  cout<<::m="<<::m<<"\n";
   return 0;
}
```

Output:

```
we are in inner block
  k=20
  m=30
 ∴ m=10
we are in outer block
  m=20
  ∵ m=10
```

∵ m will always refer to the global m.

19. How do the following statements differ?

a) char * const p;

b) char const * p;

Ans: c++ adds the concept of constant to pointers and pointers to a constant.

a) char * const p; // constant pointer

Example: char * const P= "Good";

we can not modify the address that p is initialized to.

b) char const * P; // Pointers to a constant.

Example: char const * P= &m;

P is declared as pointers to a constant. It can point to any variable of correct type, but the contents of what it points to can not be changed.

1.3 Describe the different types of writing prototype

Ans: Function prototype is a declaration statement in the calling program and is of the following form:

```
type Function-name(argument-list);
```

The argument-list contains the types and names of arguments that must be passed to the function.

Example: `float Volume (int x, float y, float z);`

Each argument variable must be declared independently inside the parentheses. That is

`float volume (int x, float y, z);` is illegal

In a function declaration the names of the arguments are dummy variables and therefore they are optional. That is —

`float volume(int, float, float);`

is acceptable. The variable names in the prototypes just act as place holders and therefore, if names are used, they don't have to match the names used in function call or function definition.

Function can be declared in an empty argument list:

11. Find errors, if any, in the following function prototype.

Ans: a) float average (x, y);

This function prototype is error.

In function declaration, the names of the arguments are dummy variable, and therefore they are optional. In the arguments list the data type of the arguments are necessary. But the above example prototype don't contain any type of the argument list.

b) int mul (int a, b);

This function prototype is error. Each argument variable must be declared independently inside the parenthesen. That is combined declaration like the above example is illegal.

c) int display (.....);

This function declaration is correct. This function prototype have an 'open' parameter list by the use of ellipses.

Output:

121.228
1.25713

1.8. When do we need to use default arguments in a function?

Ans: ① Default arguments are useful in situations where some arguments always have the same value.

② A function can be written with more parameters than are required for its most common application. Using default arguments, we can use only those arguments that are meaningful to a particular situation.

Example of prototype with default values:

```
float amount (float principal, int period, float rate=0.15);
```

The above prototype declares a default value of 0.15 to the argument rate.

```
Value = amount (5000, 7); // one argument missing
```

Passes the value of 5000 to principle and 7 to period, and then lets the function use default value of 0.15 for rate.

```
Value = amount (5000, 5, 0.12); // no missing argument
```

Passes an explicit value of 0.12 to rate.

5.1. What are the objects? How are they created?

Ans:

Objects: The class variables are known as objects.

Once a class has been declared, we can creat variables of that type by using the class name (like any other build in type variable). For example:

if a class named item is created, then; `item x;`

Creates a variable x of type item. x is called an object type item. We may also declare more than one object in one statement.

`item x, y, z;`

The declaration of an object is similar to that of a variable of any basic type. Objects can be defined as follows:

```
Class item
    {
    ------
    ------
    ------
    } x, y, z;
```

5.5 How is a member function of a class defined ?

Ans:

Member functions can be defined in two place.

1. Outside the class definition.
2. Inside the class definition.

Outside the class definition:

```
return-type class name :: function-name (argument
                                          declaration)
{
    function body;
}
```

The membership label class name:: tell the compiler that the function name belongs to the class class-name. That is, the scope of the function is restricted to the class name. The symbol :: is called Scope resolution operator

Inside the function:

```
return type function-name (argument declaration)
{
    function body;
}
```

This method of defining a member function is to replace the function declaration by the actual function definition inside the class. It is treated as an inline function. Only small functions are defined inside the

(55) Class definition.

Example:

```
Class item
  {
    int number;
    float cost;
   Public;
     Void getdata(int a, float b);
   // inline function
       Void putdata (void)
        {
          cout << number << "\n";
          cout << cost << "\n";
        }
  };
Void item:: getdata (int a, float b)
  { number = a;
    cost = b;
  }
```

6. Can we use the same function name for a member function of a class and an outside function in the same program file?

Ans:

Yes, we can use the same function name for a member function of a class and an outside function in the same program. The "membership label" will resolve their scope.

Explanation: The definition of a function is as follow;

```
return_type class_name:: function name (argument declaration)
{
function body;
}
```

The membership label class name:: tells the compiler that the function function-name belongs to the class class name. That is the scope of function is restricted to the class name specified in the header line. The symbol :: is called the scope resolution operator.

5.7. a) Describe the mechanism of accessing data members and member functions inside the main program.

Ans: The private data of a class can be accessed only through the member functions of that class. The following is the format for calling a member function:

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

For example, the function call statement

```
x.getdata(100, 75.5); // is valid
```

A variable declared as public can be accessed by the objects directly. Example:

if S1 is an object of sample, then

```
S1.read();   // won't work; objects cannot access
             // private members.
```

is illegal. However, the function read() can be called by the function update() to update the value of m.

```
void sample::update(void)
{
    read();   // simple call; no object used
}
```

5.7 b) Describe the mechanism of accessing data members and member functions inside a member function of the same class.

Ans: A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a

Private function using the dot operator ; Examples:

```
class sample
  {
    int m;
    void read (void); // private member function
  public:
    void update (void);
    void write (void);
  };
//-------------//
class ABC
  { int a;
  public:
    void setvalue (int i) { a = i; }
    friend void max (xyz, ABC);
  };
//-------------//
void max (xyz m, ABC n) // Definition of friend
  { if (m.x >= n.a)
    cout << m.x;
    else
     cout << n.a;
  }
  //-------------//
int main()
  { ABC abc;
    abc.setvalue(10);
  xyz xyz;
  xyz.setvalue (20);
  max (xyz, abc);
```

```
return 0;
}
```
output : 20

c) Describe the mechanism of accessing data members and member functions inside a member function of another class.

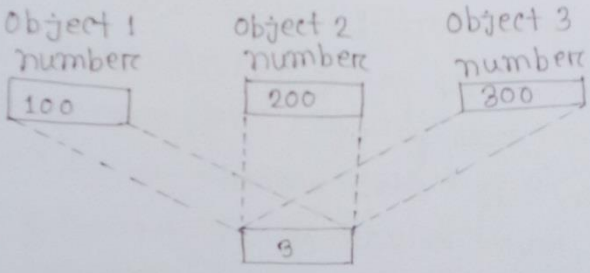
Ans: We can also declare all the member functions of one class as the friend functions of another class. In such cases, the class is called a friend class.

```
#include<iostream>
Using namespace std;
class ABC; // forward declaration.
//.....//
class xyz
{ int x;
  Public:
    void setvalue(int i) {x=i;}
    friend void max(xyz, ABC);
};
class xyz
{  int x;
   int y;
  Public:
   int z;
};
---
xyz P;
P.x=0; // error, x is private.
P.z=10; //ok, z is public.
---
```

5.8. When do we declare a members of a class static?

Ans: A static member variable has certain special characteristics. These are:

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.



- It is visible only within the class, but its life time in the entire program.
- Static variables are normally used to maintain values common to the entire class. Example:

```
#include<iostream>
Using namespace std;
Class item
{ static int count;
  int numbers;
 Public:
    void getdata(int a)
      { number=a;
      count++;
    }
```

```
void getcount(void)
 { cout<< "count : ";
   cout<<count<< "\n";
 }
};
int item :: count;
int main()
 {  item a,b,c; // count is initialized to zero.
    a.getcount(); // display count
    b. getcount();
    c. getcount();
    a. getdata(100); // getting data into object a.
    b. getdata(200); // getting data into object b.
    c. getdata(300); // getting data into object c.
cout<< "After reading data" << "\n";
   a. getcount(); // display count.
   b. getcount();
   c. getcount();
  return 0;
 }
```

The output:

```
count : 0
count : 0
count : 0
After reading data.
count : 3
count : 3
count : 3
```

6.9. What is friend function? What are the merits and demerits of using friend function?

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

```
class ABC
{
  ------
  ------
  Public:
  ------
  ------
  friend void xyz(void); //declaration
};
```

The function declaration should be preceeded by the keyword friend. The function is defined elsewhere in the program. A function can be declared as a friend in any number of classes. A friend function, although not a member function, has full access rights to the private members of the class.

Merits of using friend function:

- They provide a degree of freedom in the interface design options.
- Member functions & friend functions are equally priviliged. Friend functions allows a designer to select the syntax that it deemed most readable which lowers maintanance cost.
- Using friend function increasing versatility of overload operator.

- We can able to access the other class members in our class if, we use friend keyword. We can access the members without inheriting the class.

Demerits of using friend function:

- The major demerit of friend function is that they require an extra line of code when we want dynamic binding.
- Maximum size of the memory will occupied by objects according to the size of friend members. We can't do any run time polymorphism concepts in those numbers.
- Friend function use more obvious syntax for calling a function rather than what a member can do.
- To get the effect of a virtual friend, the friend functions should call a hidden (usually protected:) member function.

6
Constructors and destructors

1. What is a constructor? Is it mandatory to use constructors in a class?

Ans: Constructor: A constructor is a "special" member function whose task is to initialize the objects of its class. It is special because its 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 constructs the value of data members of the class.

A constructor is declared and defined as follows:

```
class integer
  { int m,n;
    public:
      integer (void);  // constructor declared.
    ------
    ------
  };
integer :: integer (void) // constructor defined
  { m=0;
    n=0;
  }
```

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically. The declaration;

integer int1; //object int1 created.

not only creates the object int1 of type integer but also initialized the data member m and n to ~~zero~~ zero.

☐ No,

It is not mandatory to use the constructors in a class. If we don't provide a default constructor, the compiler will construct a default constructor that simply allocates memory for the class, but it will not initialize the members of the class.

If we do not provide a copy constructor, then the compiler will provide a copy constructor, that allocates memory for the class, and then copies the members data from class to class. This is a bad if the class contains pointers, because only the pointer will be copied and we would wind up deleting an object and then using it after deletion, with potentially devastating consequence.

So, yes, it is mandatory, from a good practice point of view, and just plain mandatory when the class has pointers, to always provide a default constructor and a copy constructor, along with the appropriate destructor.

6.2 How do we invoke a constructor function?

Ans:

A constructor is a special member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created.

A constructor is declared and defined as follows:

```
// class with constructor
class integer
{
    int m,n;
  Public:
    integer(void); // constructor declared.
    .....
    .....
};
integer:: integer(void) // constructor defined.
{ m=0;
  n=0;
}
```

When a class contains a constructor like the one defined above, it is guaranted that an object created by the class will be initialized automatically. For example, the declaration:

```
integer int1 ; // object int1 created.
```

It not only creats the object int1 of type integer but also initializes its data members

m and n to zero. There is no need to write any statement to invoke the constructor function (as we do with the normal member functions).

6.3 List some of the special properties of the constructor function.

Ans: The constructor functions have some special characteristics. These are as follows:

- 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, and they can not return values.
- They can not be inherited, though a derived class can call the base class constructor.
- Like other c++ functions, they can have default arguments.
- Constructors can not be virtual.
- We can not refer to their addresses.
- An object with a constructor cannot be used as a member of a union.
- They make implicit calls to the operators new and delete when memory allocation is required.

6.1 What is a parameterized constructor?

Ans: It may be necessary to initialize the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called parameterized constructors.

The constructor integer() may take arguments as shown bellow;

```
class integer
  { int m, n;
    Public:
    integer (int x, int y); // Parameterized
                               constructor.
    ------
    ------
  };
integer :: integer (int x, int y)
  { m = x;
    n = y;
  }
```

When a constructor has been parameterized, the declaration:

```
integer int1 = integer(0,100); // explicit call
```

This statement creates an integer object int1 and passes the values 0 and 100 to it.

```
integer int1 (0,100); // implicit call.
```

6.15. Can we have more than one constructors in a class? If yes, explain the need for such a situation.

Ans: C++ permits us to use more than one constructor in a class. For example:

```
class integer
{ int m,n;
  Public:
   integer() // constructor 1
     { m=0; n=0; } // constructor 2
   integer(int a, int b) // constructor 2
   { m=a; n=b; }
   integer(integer & i) // constructor 3
   { m=i.m; n=i.n; }
};
```

This declares three constructors for an integer object. The first constructor receives no arguments, the second receives two integer arguments and the third receives one integer object as argument. For example, the declaration:

```
integer I1;
```

would automatically invoke the first constructor and set both m and n of I1 to zero. The statement.

```
integer I2 (20,40);
```

would call the second constructor which will initialize the data members m and n of I2 to 20 and 40 respectively.

The statement:

integer I3(I2);

would invoke the third constructor which copies the values of I2 into I3.

When more than one constructor function is defined in a class, we say the constructor is overloaded.

Necessity of using more than one constructors in a class: constructor overloading that is more than one constructors in a class are used to increase the flexibility of a class by having more number of constructor for a single class. By have more than one way of initializing objects can be done using more than one constructors in a class.

6.6. What do you mean by dynamic initialization of objects? Why do we need to do this?

Ans: Dynamic initialization of objects: objects can be initialized dynamically. That is to say, the initial value of an object may be provided during run-time.

The following program illustrates how to use the class variables for holding account details of a commercial bank and how to construct these variable at run time using dynamic initialization:

```
// long-term fixed deposit system.
#include<iostream>
using namespace std;
class Fixed-deposit
{ long int p_amount; // Principal amount
  int years;         // Period of investment
```

```
float Rate;     // Interest rate
float R_value;  // Return value of amount.
Public:
Fixed_deposit() { }
Fixed_deposit(long int p, int y, float r=0.12);
Fixed_deposit(long int p, int y, int r);
Void display(void);
};
Fixed_deposit::Fixed_deposit(long int p, int y, float r)
{   P_amount = p;
    Years = y;
    Rate = r;
    R_value = P_amount;
    for(int i=1; i<=y; i++)
    R_value = R_value * (1.0 + r);
}
Fixed_deposit::Fixed_deposit(long int p, int y, int r)
{   P_amount = p;
    Years = y;
    Rate = r;
    R_value = P_amount;
    for(int i=1; i<=y; i++)
    R_value = R_value * (1.0 + float(r)/100);
}
Void Fixed_deposit::display(void)
{ cout << "\n"
       << "Principal Amount = " << P_amount << "\n"
       << "Return Value = " << R_value << "\n"; }
```

```
int main()
{ Fixed deposit FD1, FD2, FD3; //deposit created.
  long int P;                  // Principal amount.
  int y; // investment period. years.
  float rr; // interest rate, decimal form.
  int R; // interest rate, Percent form.
 Cout << "Enter amount, Period, interest rate (in
                                    Percent)" << "\n";

 cin >> P >> y >> rr;
 FD2 = Fixed_deposit (P, y, rr);
 cout << "Enter amount and period" << "\n";
 cin >> P >> y;
 FD3 = Fixed_diposit (P, y);
 cout << "\n Deposit 1";
 FD1.display ();
 cout << "\n Deposit 2";
 FD2.display ();
 cout << "\n Deposit 3";
 FD3.display ();
 return 0;
}
```

The output would be:

```
Enter amount, period, interest rate (in percent)
 10000    3  18
Enter amount, period, interest rate (in decimal fo
 10000    3  0.18
Enter amount an period
 10000    3
```

Deposite 1

Principal Amount = 10000
Return value = 16430.3

Deposit 2

Principal Amount = 10000
Return value = 16430.3

Deposit 3

Principal Amount = 10000
Return value = 14049.3

The program use three overcloaded constructors. The parameter values of these constructors are provided at runtime.

Necessity of using dynamic initialization:

- Dynamic initialization can provide various initialization formats, using overloaded constructors.
- This provides the flexibility of using different formate of data at run time dependi upon the situation.

6.8. Distinguish between the following two statements:

time T2(T1);
time T2 = T1;

T1 and T2 are objects of time class.

Ans: The statement

time T2(T1);

would define the object T2 and at the same time initialize it to the values of T1. Another form of this statement is

time T2 = T1;

The process of initializing through a copy constructor is known as copy initialization. The statement,

T2 = T1;

will not invoke the copy constructor. However as T1 and T2 are objects, this statement is legal and simply assigned the values of T1 and T2, member-by-member. This is the task of the overloaded assignment operator (=).

6.9. Describe the importance of destructors.

Ans: A destructor, as the name implies, is used to destroy the objects that have been created by a constructors. The destructor is a member function whose name is the same as the class name but is preceded by a tilde. The destructor for the class integer can be defined as shown below:

```
~integer(){}
```

A destructor never takes any argument nor does it return any value. Destructor can also be defined as follows:

```
matrix::~matrix()
{ for(int i=0; i<d1; i++)
    delete p[i];
  delete p;
}
```

The main use of destructors is to release dynamic allocated memory.

Destructors are used to free memory, release resources and to perform other clean up.

Destructors are automatically named when an object is destroyed.