Object ORIENTED Programming

Interview Preparation.

Ashutosh Jha.

↳ Object ORIENTED PROGRAMMING:-
((INTERVIEW) PREPRATION)

Ashutosh Jha.

↳ OOPS stands for object oriented proggramming language, it is a methadology to design a program with the help of classes and objects.

↳ OOPS allows us to create reusable codes.

↳ The main aim of the OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except this function.

* CLASSES AND OBJECTS:-

↳ CLASS → A class is like a blueprint of data member and functions.

Example:- Car is a class, it has some data member like speed., no of wheels, weight.

```
class CarClass {   // The class
  public:          // Access specifier
  double weight;   // (Variable)
  String type;     // (String variable)
  public double getweight() {  // getter.
     return weight;
  }
};
```

↳ It is a user defined data type which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

ii) Object → An object is an Instance of a class.

When a class is defined no memory is allocated but when it is instaintiated (i.e. Object is created) memory is allocated.

↳ In the above example car is a class, so when we create an object of class car that object has all the properties of the same.

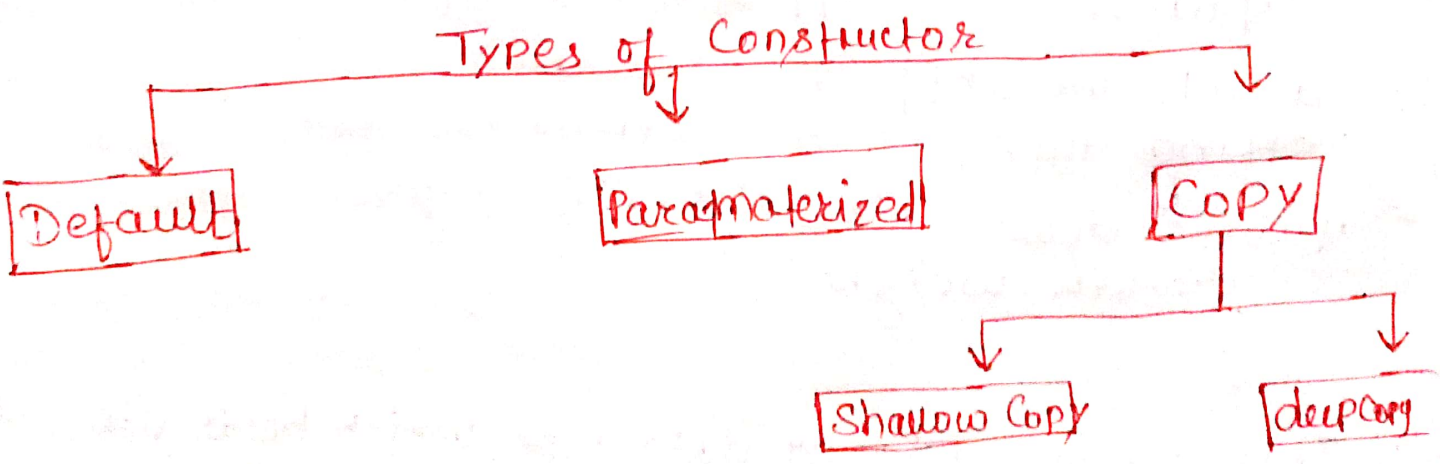
Ex- [Let's consider we create an object of class car as Car audi]

```
int main()
 Car audi;
 audi.getweight();
 audi.wheel = 4;
 return 0;
 }.
```

↳ Constructor and Destructor:-

Constructor is a special method which is invoked automatically at the time of object creation.

↳ It has no return type.
↳ Constructor has same name as class itself.
↳ if we do not specify, then C++ compiler generates a default constructor for us.



1. **Default Constructor** :- A default constructor doesn't have any parameter, it is invoked at the time of creating an object.

Input →

```
class Car{
  Public:
  Car(){ No Parameter
  Cout << " Car object created << " ";
  };
int main(){
  Car car;
  return 0;
  }
```

Output →
Car object created

2. **Parametrized Constructor** :- A parametrized constructor have any parameters, we can pass parameter (Arguments) while object creation.

Input →

```
class Car{
 public:
 string braud;
  Car(string x){
    braud = x;
    }};
 int main(){
  Car carObj1("BMW");
  Cout<<" Car with braud"<< Car Obj1. braud;
    return 0;
    }
```

Output → Car with braud BMW

3) **Copy Constructor** :- The copy constructor is used to create a new object by copying the members of an existing object.

↳ Types of Copy Constructor

1) Shallow Copy
2) Deep Copy

1) **Shallow Copy** :- Whenever a copy constructor is called by its by default shallow copy.
The pointer points to the same copy of class. ie when we copy the data members of an object to the other, they point to the same Address.

2) **Deep Copy** :- Deep copy allocates memory for the copy dynamically and then copies the actual values, both the source and copy have separate memory locations.

Destructor → Destructor destructs the object of classes.

1) Destructor doesn't have a return type.
2) It can be defined only once in a class.
3) Destructor cannot be static.
4) Actually destructor doesn't destroy object, it is the last function that invoked before object destroy.

Access Specifiers

Types of Access specifiers

- Private
- Public
- Protected

a) **Public :-** When members are Accessed from outside the class. it is of type public.

```
class car{
 Public:
  int x;
};

int main(){
   car car1;
  car1.x = 60;
  cout<< car1.x;
    return 0;
    }.
```

b) **Private :-** When members are accessible only inside the class its of type private.

```
class car{
  private:
   int speed;

 public:
void setspeed(int spd)
     speed = spd;
   }
  int getspeed(){
    return speed;
      }
    };

int main(){
 Car myCar;
 myCar. setspeed(200);
   cout<< myCar. get-speed();
    return 0;
```

c) Protected – When members are accessible in inherited class and cannot be accessed from outside the class, however they can be accessed.

#) friend class → A friend class can access private and protected member of other class in which it is declared as friend.

#) It has access to the class protected and private members.

```
class Car{
  private:
  int speed;
  int dist;
  int time;
  // friend function
  friend int avgspeed(Car);
public:
  Car (int dist, int time){
    this -> dist = dist;
    this -> time = time;
      }
};
    // friend function defination
int avgspeed (Car car){
return car.dist/car.time;
    }
int main(){
  Car c(20,10);
  cout << "Car:" << avgspeed(c);
      return 0;
    }
```

Output = 2.

Virtual function

A member function from the base class that is redifined in a derived class is refered to as a virtual function in c++.

It is declared using virtual keyword.

This keyword

this → It is used when you want to refer to the current class.

This keyword is used to distinguish b/w a local variable and an instance variable.

Abstract CLASS

A class is abstracted in c++ by declaring at least one of its functions as a pure virtual function

```
class Car {
public:
  int weight;
virtual void speed ()=0;
  void getweight(){
    return weight;
    }
    };
class Audi : Car{
    int speed;
  void speed (int x){
      speed = x
      };
      =
```

Abstraction

Abstraction:- Abstraction means only displaying the essential information and hidding the details.

```
class Add
{
public:
int a;
int b;
void addData (int x, int y){
        a = x
        b = y
cout << "Addition of" << x << "and" << "is" << x+y;
    }
    };

int main(){
  Add a;
a.addData (10, 20);
  return 0;
  }
```

——x——

Advantage of Abstraction:-

1) Avoid Code duplication
2) Can change internal implementation of class independentally.

↳ Encapsulation :-

It binds together the data and functions just like capsules.

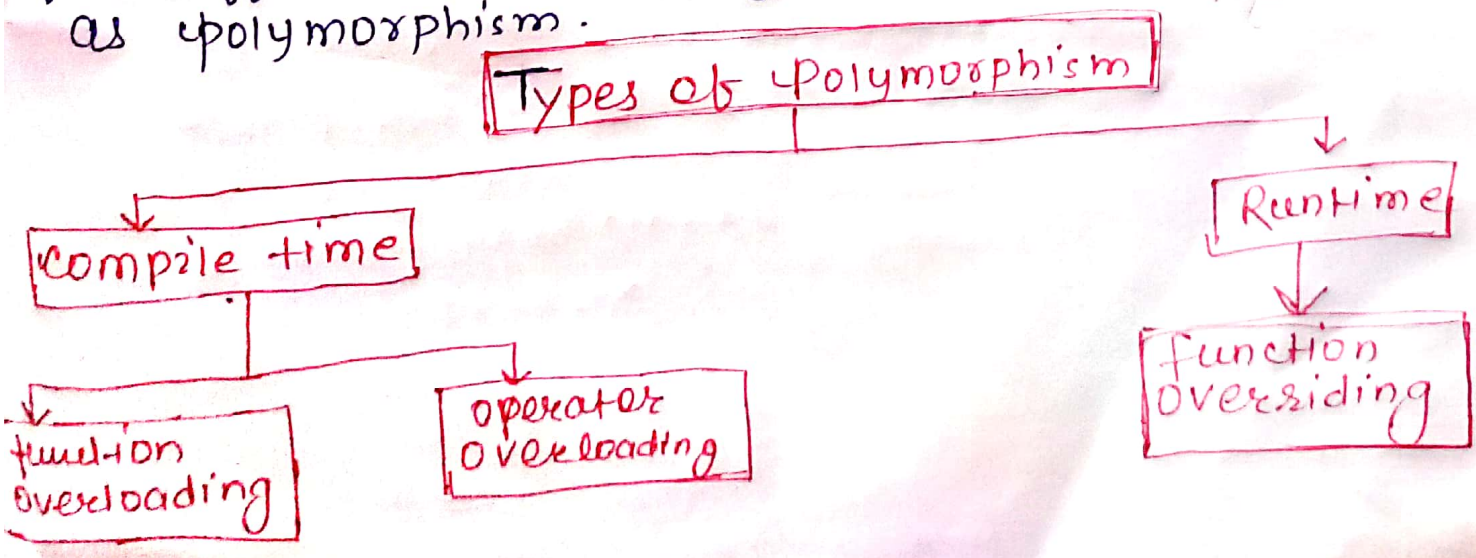
↳ It combines the code and data into a single entity or unit to protect the data from the outside world.

```
class Car {
private:
    int speed;
public:
    int getSpeed() {
        return speed;
    }
    void setSpeed(int speed) {
        this -> speed = speed;
    }
};
```

↳ Polymorphism

↳ Poly means many and morph means means forms.

The ability to present the same interface for different underlaying forms is refered to as polymorphism.



↳ Compile time

1.) function overloading –

Distinct functions are called depending on the amount and data types of parameter when we have two functions with the same name but different parameters. Overloading of a function is what this is.

↳ The name of the person and the return types are same, but they different in the no. of arguments.

Example

```
void print (int i){
 cout<<" Here is int "<<i<<endl;
 }
void print (float i){
 cout<<" Here is float "<<i<<endl;
 },
  int main(){
   print(10);
   print(10.12);
 }.
```

2) Operator Overloading :-

overloading operators for our use is called operator overloading.

↳ some operators can't be overloaded like dot(.), scope resolution (::), ternary (?:)

Example

```
class Example {
private:
  int count;
public;
Example (int count){
this->count = count;
  }
void operator --(){
 count = count - 5;
 }
Void print(){
 cout << "count is :" << count;
 };
int main(){
 Example ex(8);
  --ex
  ex.print();
  return 0;
    }.
```

Runtime Polymorphism :-

1) Function overriding :- function overriding occurs when a method from a parent class is duplicated in the child class.

Example of function overriding :-

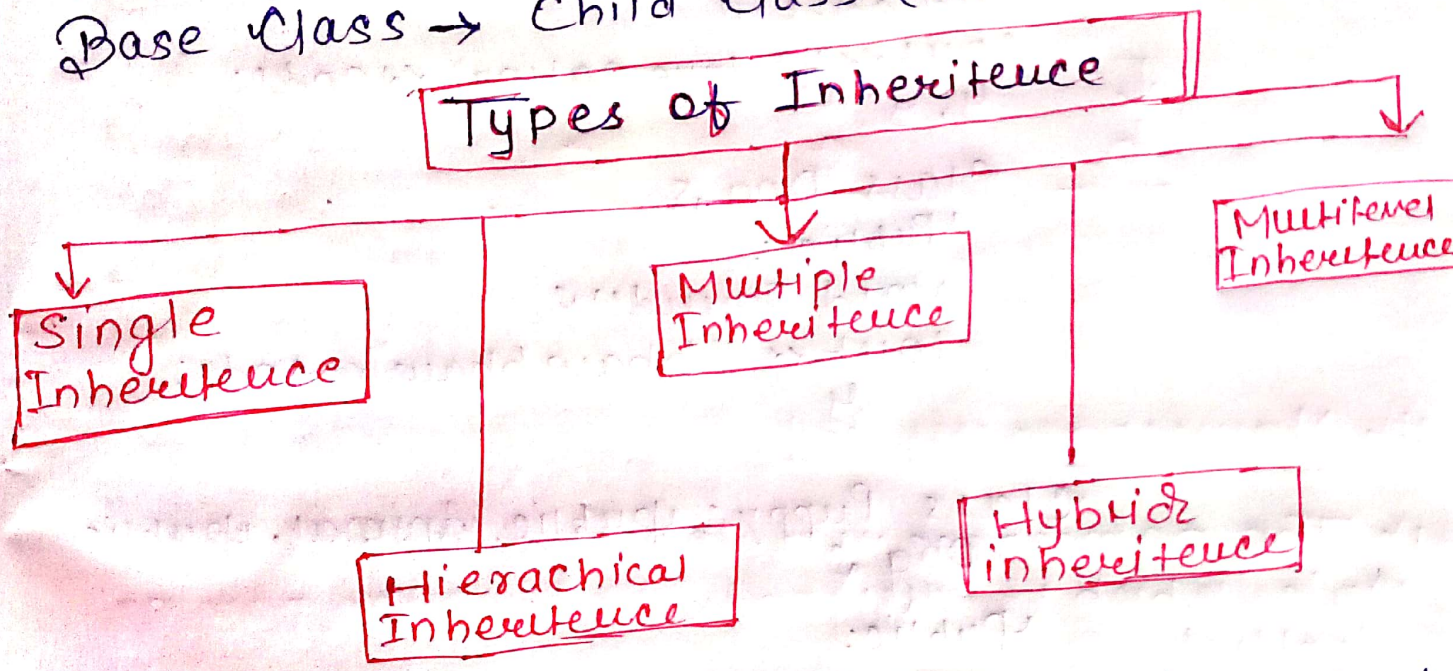
```
class Animal {
  public:
 void sound() {
  cout << "Animal sound" << endl;
        }; };

class Dog : public Animal {
    public:
   void sound() {
   cout << "bhaw bhaw" << endl;
       }
       };
 int main() {
  Animal animal;
  animal.sound();
   Dog dog;
   dog.sound();
   return 0;
   }
```

Inheritence :-

Inheritence → The ability of creating a new class from an existing class.

↳ When a class inherit properties from another class its called Inheritence.

Base Class → Child class (derived class)



1) Single Inheritence :- A derived class is produced by this inheritence from a single base class.

```
class Animal {
  Public:
void eat(){
 cout << "I Love eating" << endl;
  }};

class Dog : public Animal {
  public:
   void sound(){
   cout << " bhau bhau " << endl;
    };
   int main(){
        Dog dog;
        dog.sound();
        dog.eat();
        return 0;
    }
```

↳ Multiple Inheritance :-

A derived class is produced by this inheritance from many base class.

Ex.

```
class Animal {
 public:
 void eat() {
 cout<<"I love eating"<<endl;
 };
 class Dog {
  public:
  void sound() {
  cout<<"bhau bhau"<<endl;
    }
    };
class Puppy : public Animal, public
  Dog {
 public:
  void breed () {
 cout<<"labrador"<<endl;
    }
    };
 int main() {
   Puppy puppy
   Puppy.sound()
   Puppy.eat();
   Puppy.breed();
   return 0;
    }.
```

Multilevel Inheritance

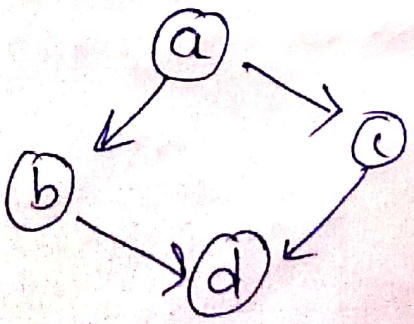
A derived class is produced from another derived class in this inheritance.

Ex —

```
class Animal {
 public:
 void eat() {
 cout << "I Love eating" << endl
    }
    };
 class Dog @ : public Animal {
  public:
   void sound() {
   cout << "bhau bhau" << endl;
   }, };
  class puppy : Public Dog {
      public:
      void weep() {
       cout << " weeping" << endl;
     }
     };
int main() {
  Puppy Puppy;
  Puppy. sound;
   Puppy. eat ();
   Puppy. weep();
  return 0;
    }.
```

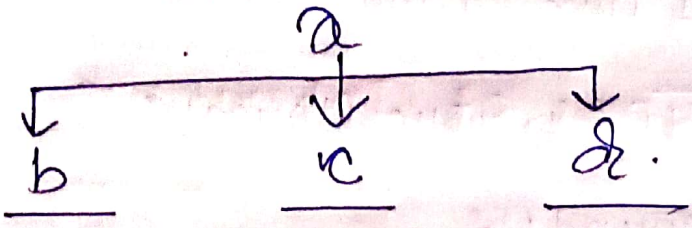
↳ Hybrid Inheritance

↳ This is a Result of combining multiple Inheritance.



4 Hierarchichal Inheritance

A single base class is used to construct several derived classes in this inheritance and additional child classes serve as the parent classes for multiple child classes.



#) Namespace in C++

1) The namespace is a logical divison of the code which is designed to stop the naming conflict.

2) Namespace defines the scope where the Identifiers such as variables, class, functions are declared.

3) The main purpose of using namespace in C++ is to Remove the Ambugity.